Procedural Generation of Three-Dimensional Game Levels with Interior Architecture

by

William Hamilton

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Master of Information Technology
in
Digital Media

Carleton University
Ottawa, Ontario

Abstract

Procedural Content Generation can help game developers by automating the creation of content that is costly and tedious to develop by hand. We present a system to generate 3D game levels based on interior spaces such as buildings and dungeons. Our system generates individual rooms by using a shape grammar to recursively subdivide rectangular blocks. Multiple rooms can be joined to form a larger level layout. Our system searches for a good layout based on a fitness function specified by a human designer. We tested three search algorithms for this purpose and found that in almost all cases an evolutionary algorithm produced the best results. Once a level layout has been selected, architectural details are added to the layout by placing modular meshes such as columns and wall segments. The final level can be loaded in *Unreal Engine 4*, a widely used game engine.

Preface

Looking back on all I have done, I cannot help but feel like one who set out with a flimsy piece of chain and the naïve intention of subduing a terrible whirlwind. Far from capturing it, I was seized by its great force and borne hither and thither before at last being deposited far from where I began. All that I have managed to grasp seems like a thin wisp of cloud, insignificant compared to the mighty forms of darkly intimated heavenly beings I beheld in the clutches of the storm. I have striven in this thesis to diligently describe my thin wisp of cloud. Nonetheless, I hope it offers its readers some suggestion of the greater visions, though it be but a sketch of future possibilities.

I would be remiss not to mention a few of the people who have helped me the most over the past two and a half years. This thesis could not possibly have been written without the guidance of my supervisor, Dr. Chris Joslin. For his unwavering confidence in me and his many insightful suggestions, I am deeply thankful. I always left his office feeling more optimistic and sure of my direction than I did when I walked in.

I must also offer thanks to my friend Jon Newhall, for helping me edit several sections of my thesis and for sending me surprisingly good advice about academic writing. Jon helped motivate me to work on my thesis every week by introducing me to great board games and then refusing to play them with me unless I worked at least as many hours as I said I would.

My brother Richard helped proofread my thesis and caught several subtle errors. Aside from this, he helped introduce me to the fascinating area of procedural level generation, and was always a joy to talk to about my research because of his sincere interest in the topic.

My parents have been an unwavering source of support and encouragement not only over the past two and a half years but for the entire span of my existence.

Finally, I must thank Kate, my constant companion and the love of my life.

— Billy

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

CGA — Computer-generated architecture

EA — Evolutionary algorithm

LOD — Levels of detail

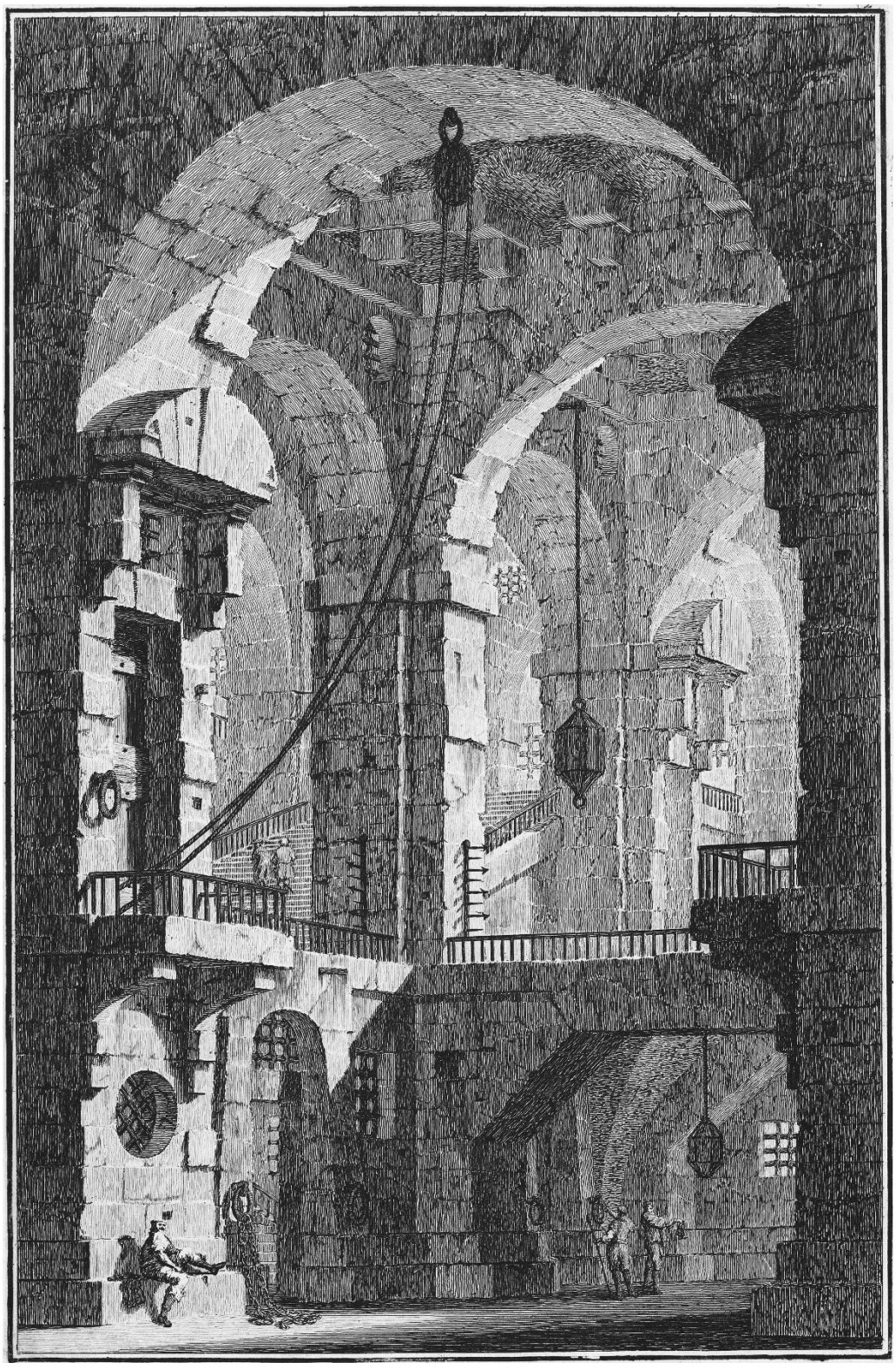
MCTS — Monte Carlo tree search

PCG — Procedural Content Generation

RDFS — Randomized depth-first search

RPG — Role-playing game

UCT — Upper Confidence Bound for Trees



Carcere oscura, by Giovanni Battista Piranesi, *c*. 1750 [Piranesi 1750]

1 Introduction

The video game industry is one of Canada's fastest-growing industries. In 2017, the industry employed 21,700 people and added $3.7 billion to Canada's GDP [ESAC 2018]. However, a major challenge facing the game industry is the continually increasing cost of game development. Since 1990, the inflation-adjusted cost of developing a top-quality game has increased by a factor of ten every decade [Koster 2018]. Today, budgets of $50 to 100 million (before marketing) are not unusual for top-end games. As technology has improved, consumers have come to expect game content of an ever-higher quality in ever-greater quantities. The creation of this content, which includes models, animation, quests, and level maps, is time-consuming and labour-intensive, a problem known as the content bottleneck [Amato 2017].

One way to address this bottleneck is Procedural Content Generation (PCG), which is the use of algorithms to produce game content. Although PCG has been used since the early days of the game industry, over the past decade it has become a topic of increasing interest both in the game industry and the academic community. The use of PCG offers several benefits. In terms of gameplay, PCG can provide new and different content each time a game is played, allowing the player to explore endless variations of a game, or to experience a version specially tailored to his or her choices and needs. From a technical point of view, PCG can help reduce the amount of memory and disk space required to store game content. Most importantly for industry applications, PCG can vastly reduce the costs of game development by allowing routine design tasks to be completed automatically.

The best-known examples of PCG systems are those that generate content at runtime on the end user's computer, which appear in games like *Minecraft* (2011). *Minecraft* began as a one-man hobby project, but its effective combination of creative sandbox gameplay with massive procedurally generated worlds allowed it to become one of the best-selling games of all time. Of course, not all games have been equally successful at harnessing PCG to drive gameplay. *No*

Man's Sky (2016) promised to immerse gamers in a procedurally generated galaxy of eighteen quintillion planets, each with its own plants and animals, but upon its release, the game received backlash from players who found its worlds repetitive and uninteresting [Mailberg 2016]. Despite achieving many of its ambitious technical goals, *No Man's Sky* failed to live up to people's expectations. This high-profile disappointment brought into focus some of the major limitations of existing PCG technology. Procedurally generated content tends to be generic or bland, lacking in structure or purpose, aesthetically empty, and devoid of creative insight [Togelius 2013]. In short, it feels computer-generated.

On its own, procedurally generated content cannot replace hand-crafted content in most applications. An alternative approach, which avoids many of the pitfalls of runtime PCG, is to generate content behind the scenes during a game's development. For example, the Academy Award-winning PCG software *SpeedTree* has been used to generate realistic models of trees for dozens of successful games. Unlike the worlds of *No Man's Sky*, which demand the player's attention, these trees pass unnoticed as they do not stand out from the hand-crafted content around them. To the best of our knowledge, there has never been any consumer backlash against *SpeedTree*.

The great advantage of development-time PCG is that it allows the power and efficiency of PCG algorithms to be combined with the skills and sensibilities of human designers. Content can be generated according to a designer's specifications and incorporated into the final game at the designer's discretion. Poor content can be rejected and re-generated, or manually tweaked and improved. Development-time generation also enables the use of many methods that would not be practical at runtime, such as algorithms with long running times or which require extensive human input. The downside of development-time PCG is that it does not provide a new experience each time the game is played. This means it cannot deliver the unique gameplay of a game like *Minecraft*. However, for the purposes of most games, a limited amount of refined high-quality content is more valuable than an endless supply of random and generic content. While runtime PCG can be a fun toy for the player, development-time PCG is a practical tool for the designer.

This thesis focuses on generating 3D game levels using development-time PCG. Game levels are the virtual worlds in which games take place. Today, the levels in most high-budget games consist of large and detailed 3D environments, which are difficult and time-consuming to

design. Although exact budget breakdowns for major games are not publicly available, we can get some sense of the amount of work involved in level design by looking at game credits and development timelines. For example, the game *Assassin's Creed Origins* (2017) spent three years in development and credits a total of 79 people as world designers, level designers, and lead level designers. Based on online reports from players [HowLongToBeat], the game takes an average of 79 hours to completely explore, suggesting that, as a very rough approximation, a level that takes one hour to play takes three man-years to design. Evidently, level design is an important part of the content bottleneck, and as such is an excellent target for PCG. However, prior research in level generation has focused primarily on two-dimensional levels, which are much less difficult and expensive to create.

We focus in particular on the generation of levels based on interior architectural spaces, such as buildings and dungeons. These spaces are ubiquitous in games. In the long term, the goal of this research is to create a tool that can generate systems of rooms and corridors as easily as *SpeedTree* generates trees. We do not expect that such a tool would generate perfectly finished and polished levels, but that it would serve as a useful labour-saving device, leaving designers with more time to spend refining the generated levels and adding original creative details.

In the present work, our aim is to address the technical needs of such a tool. That is, we aim to describe the algorithms required to generate 3D game levels. To prove the viability of our methods, we present a level generator prototype that we developed, and show the results that it can produce.

1.1 Requirements

To fulfill our goals, our level generator must have several properties. First of all, it must generate levels with believable architectural features such as columns, arched ceilings, doorways, and balconies – not just a series of open cubes. Ideally, the generator should be able to generate levels based on actual architectural traditions, such as gothic architecture from medieval Europe. However, the methods used in the generator should be general enough to be applied to many different styles of architecture.

The generator must also be controllable. This is an essential feature if it is to be useful as a development-time tool for designers. It should be possible for the designer to control the high-level properties of the generated level, including the length of the level and the area it occupies. It should also be possible for the designer to control topological features of the level that are likely to affect gameplay, such as whether the level has many dead ends or branching paths.

Furthermore, the generator must be able to produce truly three-dimensional levels. That is, it should be generate levels with vertical features such as staircases, ledges, and multiple storeys. We do not simply wish to take a 2D level generator and instantiate each room as a 3D space.

We also do not wish to build a generator that simply combines prefabricated rooms. Such a generator is of limited use because it is not likely that game designers will wish to use the exact same room more than once in a game. Instead, we wish the generator to build each room up from basic components such as its walls, floor, and ceiling. The finished levels must be produced in a format that can be loaded and run in a game engine such as *Unreal Engine 4*.

Piranesi's etching of the *carcere oscura* ("dark prison"), reproduced on page x, exemplifies the type of environment we wish to generate. The image shows a complex interior space with detailed architecture and many three-dimensional elements such as stairs and ledges. A level generator capable of automatically producing game environments with these features would represent a significant advancement in the state of the art.

1.2 Proposed Solution

Our proposed solution is to break the problem of generating 3D levels into three subproblems. First, there is the generation of individual room designs. We address this subproblem using a shape grammar, which can effectively represent rules about the structure of 3D architecture. The second subproblem is the generation of a level layout. By a level layout, we refer to the arrangement of multiple rooms into a good level structure. We address this subproblem by using a search algorithm to test many possible layouts. The layouts can be evaluated and compared using a fitness function. The third subproblem is the placement of individual meshes to create the finer architectural details of a level. We address this problem using a case-based system,

which iterates over the interior spaces in a level and places meshes based on the types of spaces and the adjacencies that exist between them.

1.3 Contributions

The original contributions of this thesis are the following:

- A new method for generating interior architectural spaces (rooms) using a shape grammar.
- A comparison of three search algorithms for 3D level layout, namely a randomized depth-first search, a Monte Carlo tree search, and an evolutionary algorithm. We show that the evolutionary algorithm produces the best results in most cases.
- A new method for placing 3D meshes in a level, based on an abstract level layout.
- An explanation of how these three systems can be combined to form a complete level generator.

1.4 Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background information about game levels, formal grammars, and evolutionary algorithms. Chapter 3 describes previous work on game level generation, as well as some related work on computer-generated architecture. Chapter 4 describes our new method for level generation. In Chapter 5, we evaluate our method and its results. Chapter 6 presents our conclusions and suggestions for future work.

2 Background

This chapter provides a basic introduction to a few concepts that will be referred to repeatedly throughout the remainder of the thesis. Section 2.1 describes game levels, including the types of game levels, how they can be represented, and how they are traditionally designed. Section 2.2 describes formal grammars, a system for specifying the structure of content. Section 2.3 describes evolutionary algorithms, a form of algorithm inspired by the natural process of evolution.

2.1 Game Levels

A game level consists of a space in which gameplay takes place, a population of game objects within that space, and a mission that the player must accomplish. Game levels classically feature missions based on topology, requiring the player either to "get to the other side" or "visit every location" [Koster 2005]. In most games, the player must complete a series of levels to win the game. Early levels are used to introduce key mechanics and teach the player to play, while later levels present new variations and successively harder challenges. Thus, level design is often central to a game's progression system.

2.1.1 Types of Game Levels

A game's levels must reflect the game's mechanics, dynamics, and aesthetic goals. As such, game levels vary drastically between different games, and a level generator must be built with a particular target game in mind. That said, games within the same genre generally feature loosely similar levels. This subsection describes several types of level and specific games that have been the focus of level generation work. The specific procedural generation methods that have been applied to these games are described more fully in chapter 3.

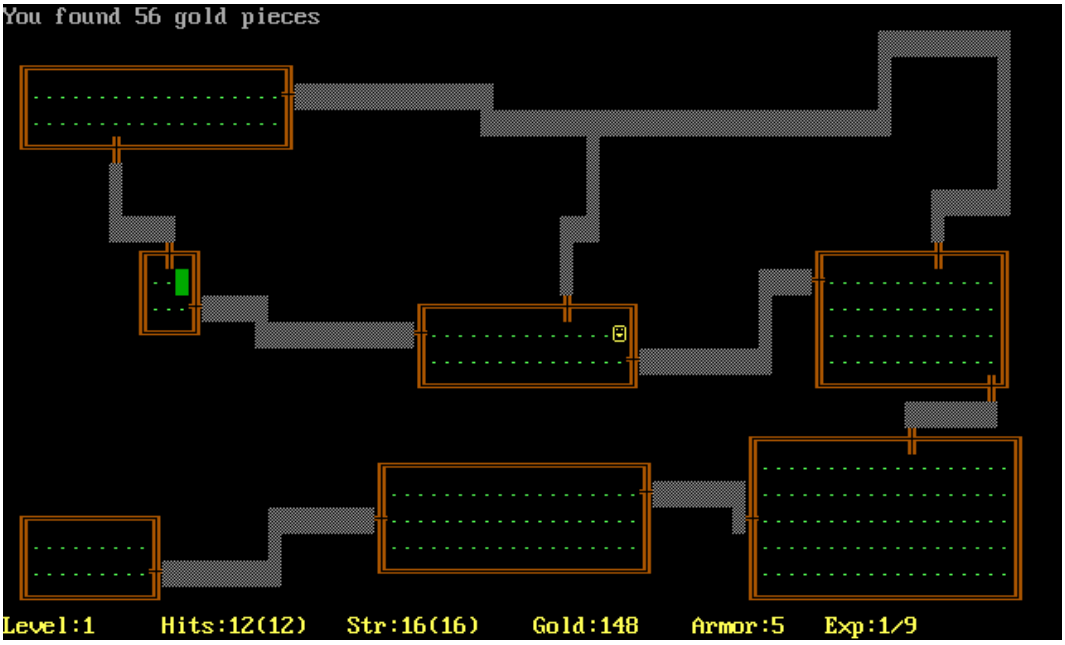


Figure 2-1. A dungeon level from *Rogue* (1980).

Dungeons. A *dungeon* is an indoor or underground game space consisting of multiple interconnected rooms and passages. These rooms and passages contain challenges for players to overcome, such as monsters, traps, or puzzles. The term originated with the tabletop roleplaying game *Dungeons & Dragons* (1974), in which the first dungeons were literally situated in the basements and subbasements of abandoned castles, but the term dungeon is now taken to encompass any game environment that offers similar gameplay features.

People have been interested in procedurally generating dungeons for nearly as long as the concept of a dungeon has existed. The earliest dungeon generation algorithms were carried out by hand using dice and cards. An overview of this analogue history of procedural generation is provided by Smith [Smith 2015]. Two major genres of video games featuring dungeons are roguelike games and action-adventure games.

Roguelike games. *Rogue* (1980) was one of the first computer games to include a procedural dungeon generator. The generator was relatively simple and predictable, as it always placed rooms in a three-by-three grid. Nonetheless, *Rogue* inspired an entire genre of "roguelike" computer RPGs, which predominantly feature procedural dungeon generation. Some of the most famous and popular roguelike games are *NetHack*, *Angband*, and *Dungeon Crawl: Stone Soup*. These classic roguelikes are continuously developed by hobbyists and

feature only primitive graphics, but several major commercial games have also taken inspiration from the roguelike genre. *Diablo* (1996) and its sequels feature procedurally generated isometric dungeons inspired by *Angband*. The Japanese company Chunsoft has found commercial success combining the roguelike formula with licensed franchises to create games such as *Pokémon Super Mystery Dungeon* (2015). Roguelike games are so closely associated with procedural level generation that games in other genres with randomly generated levels are often known as "rogue-lites."

A dungeon level in a roguelike game consists of a square grid of tiles. Each tile may contain a wall, open floor, or some other feature such as a staircase to the next level. Monsters and treasure are randomly distributed throughout the level. The player must explore the level, collecting treasure while killing or avoiding monsters. When the player dies, the game starts over and a new random dungeon is generated. This mechanic is known as *permadeath*, because the old character and dungeon are permanently lost. Roguelike dungeon levels are not generally designed to provide a highly structured sequence of tasks or challenges. On the contrary, the unpredictability of each level's design is a major part of the roguelike genre's appeal. Little academic research has focussed on level generation for roguelike games, as they are generally well served by simple algorithms. Nonetheless, roguelike games have proven useful as test-beds for level generation techniques [Liapis 2014; Smith 2014].

Action-adventure games. The action-adventure genre is a very broad genre of games featuring elements of exploration, puzzle-solving, and combat. Unlike roguelike games, action-adventure games normally feature carefully designed dungeons. These dungeons provide game designers with a structured and unobtrusive way to control a player's progression through a game [van der Linden 2014]. For example, by placing the only key to a room behind a monster, the designers can ensure players will encounter the monster before entering the room. Puzzles that prevent a player from entering a given area of a dungeon until a task in another area has been completed are known generally as *lock-and-key puzzles*, whether or not they literally involve locks and keys. Many action-adventure games feature sophisticated dungeons with complex logical and physical connections between different areas.

The dungeons of Nintendo's *Legend of Zelda* series are often used by researchers as examples of good dungeon design [Dormans 2010; Togelius 2013; Summerville 2015]. Early *Zelda* games, such as *The Legend of Zelda: A Link to the Past* (1991), feature two-dimensional

Figure 2-2. Dungeon rooms from *The Legend of Zelda: A Link to the Past* (1991) [VGMaps].

dungeons viewed from a top-down perspective. This type of dungeon has been a frequent topic of PCG research [Sorenson 2010; van der Linden 2013; Karavolos 2015; Lavender 2015]. Later *Zelda* games, such as *The Legend of Zelda: Ocarina of Time* (1998), feature three-dimensional dungeon levels. Such levels are naturally more complex to design and have not yet been the subject of much work in PCG.

Platformer games. Platformer games require the player to navigate a dangerous environment by jumping over pits and other obstacles. Traditionally, platformer games feature two-dimensional levels viewed from the side, although 3D platformers also exist. The player's ability to navigate a level is constrained by gravity and by the movement and jumping capabilities of the player avatar. For this reason, platformer levels are more difficult to generate than top-down dungeons [Compton 2006].

Perhaps the most iconic platformer game is *Super Mario Bros.* (1985). *Mario* and similar "side-scroller" games feature linear levels much wider than they are tall. The player traverses each level from the left to right, overcoming obstacles in the order they are encountered. This type of level has been another popular subject of PCG research [Sorenson 2010; Mawhorter 2010; Smith 2011a; Kerssemakers 2012]. The open-source *Mario* clone *Infinite Mario Bros* (2008) has been used as a test-bed for much research, including for the Mario AI Championship [Shaker 2011]. Side-scroller levels can be abstracted as a linear sequence of challenges, so they lend themselves well to studies of game difficulty and progression. Other platformer games, such as *Prince of Persia* (1989) and *Sonic the Hedgehog* (1991), feature non-linear levels that require exploration in two-dimensions. Such levels can be considered a type of dungeon, as the player's progression through the level is controlled by the physical connections between different areas. Several systems have been developed to generate non-linear platformer levels [Mourato 2011; Koens 2015]. Perhaps the most famous is the level generator in the popular open-source platformer *Spelunky* (2008).

Like 2D platformers, 3D platformers may feature either linear or nonlinear levels. A classic example of the former is *Crash Bandicoot* (1996), while a classic example of the latter is *Super Mario 64* (1996). Level generation for 3D platformer games does not yet appear to have been the subject of significant research.

First-person shooters. First-person shooter games feature 3D levels that the player must navigate while using guns or similar weapons to fight enemies. The main focus of gameplay is combat, rather than exploration or puzzle-solving. An important consideration in level design for first-person shooters is the presence of features that affect combat, such as cover and choke points. Hullet and Whitehead provide a list of design patterns commonly appearing in first-person shooter levels [Hullet 2012]. First-person shooter games are popular for both single-player and multiplayer play, with many games supporting both these game modes. Single-player levels are designed to provide a series of challenging fights against computer opponents, while multi-player levels are designed to provide a balanced contest between two teams. Some work [Adams 2002; Cardamone 2011; Bhojan 2014] has been done on PCG of shooter levels, but not with complex 3D gameplay features.

Strategy games. Real-time and turn-based strategy games require players to control the actions of many followers, typically from a top-down perspective. Multiple human or computer-

controlled factions vie for territory and resources on a limited map, which may represent a geographical region, an entire planet, or in some cases an entire galaxy. Unlike dungeon levels, strategy game maps are usually very open, allowing player forces to move freely except where constrained by terrain features such as mountains or oceans. Many commercial strategy games have featured level generators, such as the best-selling *Age of Empires* and *Civilization* series. A major function of level generation in these games is to provide an unpredictable distribution of terrain and resources each time the game is played, forcing players to explore the map while developing their positions.

Map balance is an important consideration in level generation for strategy games. Each faction's starting position should offer comparable advantages in terms of resources and favourable terrain. This topic and several related concerns have been explored in studies of level generation for strategy games [Togelius 2010; Liapis 2013].

Sandbox games. Sandbox games are based on play and experimentation without a specific goal. Most sandbox games feature large and complex virtual worlds for the player to explore and interact with. *Minecraft* and *No Man's Sky* both fall into this genre. *Dwarf Fortress* (2006) is a well-known sandbox game that combines an extremely complex simulation of a procedurally generated world with strategy and roguelike elements.

2.1.2 Level Representations

It is possible to represent a game level in many different ways, but not all representations are equally advantageous for level generation. Many level generation algorithms are strictly dependent on a particular representation. For example, most roguelike level generators depend on a tile-based representation. Even general-purpose algorithms, such as evolutionary algorithms, may produce different outputs or show vastly different performance depending on the level representation chosen. The situation is complicated by the fact that the representation most conducive to level generation may not be the representation most suitable for use during actual gameplay. The choice of level representation is thus an essential topic in the area of level generation. This subsection describes a few of the most common representations used for game levels, as well as the advantages and disadvantages of each representation for level generation.

Tile representation. In a tile representation, a level is a 2D array of tiles. In the simplest form, each tile may be either solid or open. More complex variations use tiles of many

Figure 2-3. Tiles used to represent level features in the roguelike *Dungeon Crawl: Stone Soup* (2016).

types for different level features. Alternatively, each tile may store a density value, with values below a certain threshold representing open space. This density-based approach is useful for generating organic caves or terrain. Tile representations are very natural for 2D games, including both side-view platformers and top-down roguelikes and action-adventure games. Tile representations are also very convenient for level generation. A major reason is that each tile can be analyzed both spatially (in terms of its position), and topologically (in terms of its connections to the eight neighbouring tiles). Spatial operations such as line drawing can easily be applied to a level with a tile representation, as can graph operations such as breadth-first search.

Voxel representation. Voxels ("volume elements") are the elements of a 3D array. As such, a voxel representation is analogous to a tile representation, but describes space in three dimensions instead of two. The major downside of a voxel representation is that it demands a large amount of memory. This generally means that only a small area of the game world can be stored at a time. Voxel representations are most suitable for levels generated from deterministic mathematical functions, such that any part of the level can easily be re-generated if it has been discarded from memory. The levels of *Minecraft* are a good example of this technique. Voxels can be displayed as visible cubes, as in *Minecraft*. Alternatively, high-resolution voxel geometry can be converted into a polygon mesh representation for display. The "marching cubes" algorithm of Lorensen and Cline [Lorensen 1987] can be used for this purpose.

Mesh representation. A three-dimensional level can be represented using one or more polygon meshes. In early 3D games, it was normal for every part of a level to be modelled individually, but this approach became increasingly difficult in the early 2000s as improving

Figure 2-4. Modular mesh kit for dwarven dungeons in *Skyrim* [Burgess 2013].

technology led to a demand for large and highly detailed game environments. The now-standard paradigm of modular level design [Perry 2002] emerged as a practical way to meet this demand. In modular level design, the designer builds a level by piecing together multiple meshes representing prefabricated pieces of architecture such as wall, floor, and ceiling segments, stairs, columns, windows, and statues. Each mesh is modelled by an artist and then reused repeatedly across one or more levels. Meshes are normally designed to be laid out on a grid, so that they can easily be joined in many combinations. A mesh representation consists of a list of meshes and an affine transformation (translation, rotation, and scaling) for each mesh.

Burgess [Burgess 2013] describes the modular level design practices used by Bethesda Game Studio to develop large open-world RPGs such as *Fallout 3* (2008) and *The Elder Scrolls V: Skyrim* (2011). For *Skyrim*, a series of seven different kits were developed to represent different architectural styles, such as Nord crypts and dwarven dungeons. A danger of the modular approach is that players may suffer "art fatigue" as they begin to recognize reused components. In order to forestall this, designers often combine kits in ways not originally envisioned by the kit artists. For example, one *Skyrim* dungeon combined components from the dwarven kit with components from an ice cave kit. The kit artists worked closely with the level designers to improve the usability of the kits. Burgess emphasizes that one benefit of the modular approach is that it allows a small number of artists to support a larger number of level designers. The *Skyrim* team included eight level designers and only two kit artists.

Although a mesh representation is convenient for human level designers, and for rendering a finished level, it is less convenient for procedural level generation. Unlike tiles or voxels, meshes do not have a one-to-one correspondence with positions on a grid. A list of meshes cannot be readily interpreted as a graph of level connectivity. The traversable spaces of the level lie in the areas between the meshes, and the meshes may be arbitrarily shaped, so determining which spaces are accessible from one another requires extensive collision checking.

2.1.3 A Human Approach to Level Design

As a point of comparison to the procedural generation techniques discussed later in this thesis, it is worth providing a brief summary of the traditional human approach to level design.

Co [Co 2006] describes the creative process of level design based on his own experience as a level designer at several game studios including Valve, Blizzard, and KnowWonder. The process begins with pre-production. Producers decide how many levels the game will contain, and determine parameters for each level such as length, setting, and difficulty. Each level is assigned to a level designer, who brainstorms a list of ideas for the level based on the parameters provided. Often, the designer chooses an abstract narrative to provide the overall structure of the level. For example, a level might be inspired by a factory production line or a yin-yang symbol. The designer identifies key landmarks that will help the player navigate the level, and gathers visual references such as photographs, books, and films.

Based on all these ideas, the designer composes a written description of the level, describing areas in the order they will be experienced by the player. This description is revised and refined several times. Next, the designer works out the level layout by testing many different arrangements on paper. Once a good layout is found, details such as obstacles and rewards are added to the diagram. Each room in the diagram is keyed to a written description. The level draft is discussed with other team members and feedback is incorporated.

When the draft has been approved by the project leads, production begins. The level is implemented in a level editor. First, a rough level template is built using simple geometric shapes. Required gameplay objects such as moving platforms and triggers are added. The level is then repeatedly tested and iteratively improved. Once the template level works well, a series of quality passes are performed. Placeholder assets are replaced. Architectural details such as door and window frames are added, along with decorative props and decals (flat images on

walls). Imperfections such as damaged columns and misaligned tiles are introduced. Lighting, audio, and particle effects are among the last features to be added.

Eventually, the game enters formal alpha and beta testing. Bugs and issues with the level are fixed as they are discovered. Last-minute balance changes may be made, such as adding pickups or adjusting the number of enemies. A technical pass is performed on the level to optimize its performance. Finally, the game is shipped and the level data is archived. The level designer prepares for the next project or starts looking for a new job.

2.2 Formal Grammars

A formal grammar is a system of replacement rules that describes the structure of some type of content. Formal grammars originated with the work of Noam Chomsky in linguistics, but have been widely applied in math and computer science. In this thesis, we are concerned only with their application to procedural content generation.

Formal grammars traditionally operate on strings, but other variants exist such as graph grammars, which operate on graphs. For the sake of brevity, we hereafter refer to formal grammars simply as *grammars*. This section first explains the key concepts of grammars in terms of strings, then describes several types of grammar that have proven useful for procedural content generation. The application of grammars to game level generation and computer-generated architecture are covered in Chapter 3.

2.2.1 Key Concepts

A grammar consists of an alphabet and a set of production rules. The alphabet is the set of symbols that are valid within the grammar. These symbols can be combined to form strings. For example, a grammar's alphabet could contain the symbols $\{a, b, c\}$. The production rules describe string replacements that are valid within the grammar. A production rule consists of two strings separated by an arrow (→). Each rule indicates that it is valid to replace the string on its left-hand side with the string on its right-hand side. For example, the rule $a \rightarrow bc$ indicates that it is valid to replace a with bc.

Given some starting string, it is possible to generate a new string by applying a production rule. For example, if we start with the string ca and apply the rule $a \rightarrow bc$, we can

Algorithm 2-1. Example of a production process for a context-free grammar

alphabet {*a*, *b*, *c*, *d*}
rule1 $a \rightarrow bc$
rule2 $b \rightarrow acd$
axiom *a*

production process:

	a	(axiom)
→	*bc*	(by rule 1)
→	*acdc*	(by rule 2)
→	*bccdc*	(by rule 1)
→	*acdccdc*	(by rule 2)
	…	

obtain the new string *cbc*. More complex content can be generated using a production process with multiple iterations. In the first iteration, a production rule is applied to a starting string known as the *axiom*. In each subsequent iteration, we apply a production rule to the string obtained from the previous iteration. The process terminates when no more string replacements can be applied, *i.e.*, when an iteration results in a string in which no substring matches the left-hand side of any rule in the grammar. Alternatively, the process may be terminated after a fixed number of iterations or when the output fulfills some condition.

An example of a production process is shown in Algorithm 2-1.

2.2.2 Types of Grammars

This section introduces several different types of grammars, with an emphasis on those that are useful for procedural generation.

Context-free grammars. The grammar shown in Algorithm 2-1 can be classified as a context-free grammar. A context-free grammar is a grammar in which the left-hand side of each rule consists of a single symbol. The symbols in the alphabet of a context-free grammar can be classified according to their role in the grammar's production rules. A *non-terminal* symbol is one that forms the left-hand side of at least one rule. A *terminal* symbol is one that does not appear on the left-hand side of any rule. Once produced, a terminal symbol will never be replaced, and will appear unchanged in the final content. In Example 1, *a* and *b* are non-terminal symbols, while *c* and *d* are terminal symbols.

The production process of a context-free grammar can be conceived of as a tree. Each symbol is a node of the tree, with the axiom forming the root. The children of each node are the

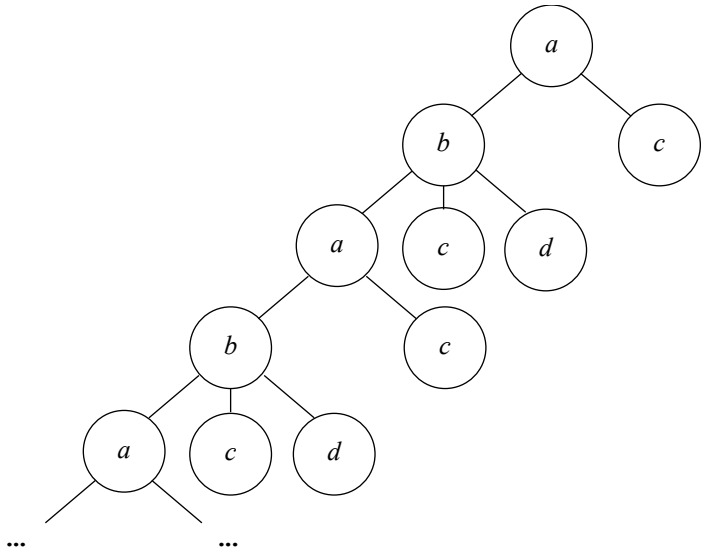


Figure 2-5. The production process of a context-free grammar, depicted as a tree.

symbols that replace it. Terminal symbols are leaf nodes, while non-terminal symbols are inner nodes. Figure 2-5 shows how the production process from Example 1 can be depicted as a tree.

L-systems. Lindenmayer systems, commonly known as L-systems, are a type of context-free grammar originally developed by the Hungarian botanist Astrid Lindenmayer. L-systems use a parallel production process, where all applicable string replacements are performed in each iteration. L-systems have proven very useful for procedural content generation. They are especially well-suited to generating branching and recursively self-similar structures, such as those found in plants. The generation of plant models using L-systems is described in detail by Prusinkiewicz and Lindenmayer [Prusinkiewicz 2012].

Attributed grammars. An attributed grammar is a form of context-free grammar in which the symbols (or other manipulated objects) have attributes. Attributed grammars were introduced by Knuth [Knuth 1968] as a way of adding semantic information to context-free grammars. In an attributed grammar, each instance of a symbol may store one or more attribute values. The values of attributes may be determined either by inheritance or by synthesis. These two processes are most easily understood when the grammar's production process is viewed as a tree. Inheritance is a top-down process where the attribute value for a symbol is a function of the attributes of its ancestors. Attribute values percolate down the tree as it is generated. By contrast, synthesis is a bottom-up process where the attribute value for a symbol is a function of

the attributes of the symbol's descendants. Synthesis requires the production process to be completed before values of attributes can be propagated up from the leaves of the tree to the root.

Graph grammars. A graph grammar is a grammar that operates on graphs. A rule may replace a node of a graph with multiple nodes and edges, or replace a specific configuration of multiple nodes and edges with a new configuration. Graph grammar systems are difficult to design because they must specify how to connect a new subgraph to the remainder of the existing graph, typically based on the edges that formerly existed between the replaced subgraph and the rest of the graph. Various solutions to this problem have been developed [Fahmy 1992].

Shape grammars. Shape grammars are grammars that operate on geometric shapes. They can be used to analyze structure in art, architecture, and design. The original shape grammars of Stiny and Gips [Stiny 1972] were designed to operate directly on patterns of lines. In this conception, a particular arrangement of line segments in space is called a shape. Each rule has a shape on the left-hand side and a new shape on the right-hand side. Shape grammars of this type have been developed to describe architectural systems such as the villas designed by the sixteenth-century Italian architect Andrea Palladio [Stiny 1978; Benros 2012]. However, these systems are not suitable for procedural generation as they require a human designer to specify which rules to apply. The rules in these systems are usually designed to permit all good designs, without necessarily excluding bad designs.

Performing replacements directly on a set of line segments leads to certain problems, especially for a computerized generation process. The most fundamental problem is that the shapes manipulated by a shape grammar lack semantic content. For example, a shape grammar cannot distinguish between a square wall and a square window. During a production process, a given arrangement of line segments might arise by coincidence and trigger a rule inappropriately.

An elegant solution to this problem is to reformulate shape grammars as set grammars [Stiny 1982]. In a set-based shape grammar, replacements are performed on a set of abstract symbols, not directly on geometric shapes. Each symbol can be used as a stand-in for a particular shape. For example, the symbol *Wall* might represent an arrangement of line segments in a square. A rule can only replace a labelled shape as a whole, so the lines in different shapes cannot accidentally come together to trigger a rule. An additional benefit of this approach is that far less computation is required to detect a matching symbol than to detect a matching arrangements of lines [Wonka 2003]. The set-based approach to shape grammars has proven

very effective for the procedural generation of models of exterior building architecture. These methods are covered in section 3.2.

2.3 Evolutionary Algorithms

How do you make better rats? You start with a bunch of rats, then kill the bad ones and let the good ones have babies. Repeat. We can apply the same principle to computer-generated content.

Evolutionary algorithms are algorithms inspired by the process of evolution in nature. Possible solutions to a problem are compared using a fitness function and only the best solutions are kept. These survivors form the basis for the next generation of solutions. New solutions are created by applying operations to modify the survivors. Over time, the population improves. Evolutionary algorithms have been used to solve many types of problems, but this section introduces them in terms of level generation.

An evolutionary algorithm for level generation [Togelius 2010] works as follows. The algorithm starts with a set of candidate levels called the *population*. These levels are compared using a mathematical function called the *fitness function*, which measures the quality of each level. A few of the best-scoring levels are kept while the others are discarded. A new population of candidate levels is then created by copying and modifying the surviving levels. This process is repeated, with each repetition being called a *generation*. Over the course of many generations, the process creates a population of levels that score highly on the fitness function.

The effectiveness of an evolutionary algorithm for procedural level generation depends on many factors. Perhaps the most important is the choice of a fitness function. The fitness function must capture all the features that make a level good or bad, which is no simple task.

Another important choice is how the game level is represented. Depending on the representation, a given level design might be easy, hard, or impossible to describe. The representation determines the shape of the search space and the likelihood of evolving a particular design.

Closely related to the choice of representation is the question of how the new levels in each generation are derived from the survivors of the previous generation. *Genetic algorithms* [Whitley 1994], the most popular type of evolutionary algorithm, use mutation and crossover

operations inspired by nature. In the context of level generation, mutation involves making small, random changes to an individual level, while crossover involves combining parts of two separate levels. The details of these operations depend on the level representation. For a genetic algorithm to succeed, the level representation and the crossover operation must be designed such that complex structures in the design of a level can survive between multiple generations. Various other parameters can also affect the output of an evolutionary algorithm, including the size of the population, the number of generations simulated, and the proportion of the population allowed to survive each generation.

The running times of evolutionary algorithms can typically be measured in minutes or hours. As such, evolutionary systems are not usually ideal for runtime content generation on the end user's computer. Fortunately, these long running times are less of a problem for development-time generation.

3 Related Work

This chapter provides an overview of previous research on procedural level generation and computer-generated architecture.

3.1 Procedural Level Generation

Level generation methods are sometimes classified into two types: constructive methods and search methods [Togelius 2010]. A constructive method is one that generates a level directly by following a series of rules and instructions. Grammars are a standard example of a constructive technique. By contrast, a search method is one that generates many levels and evaluates them using a fitness function to select the best one. Evolutionary algorithms are an example. In practice, many of the most interesting and effective level generators are hybrid systems with both constructive and search-based components. Compton and Mateas [Compton 2017] consider search methods to be "consumers of generative pipelines" because they require some underlying constructive method to generate the content they search.

We have chosen not to separate constructive and search methods in our survey. Instead, we have divided level generation methods into a number of loose categories based on their shared properties and underlying ideas. Section 3.1.1 briefly describes a number of different classic methods. The following subsections then each describe a group of related techniques. Hybrid methods using search techniques such as evolutionary algorithms are generally categorized based on the non-search-based methods they also employ. Several evolutionary algorithms that do not fit into any other group are covered in Section 3.1.8.

3.1.1 Classic Methods

We begin our survey of level generation with an assortment of classic methods. Most of these methods have a long history of use for PCG and are attested in sources dating back to antiquity (*i.e.*, roguelike games from the 1980s). As such, the references cited in this section are generally not to the originators of the techniques, but to easily accessible recent sources that summarize them. The main features these methods have in common is that they are easy to implement and easy to understand. Many of them also rely upon a tile representation of a game level. These classic techniques are frequently reused in the more complex systems described in later sections.

Fixed layout. The simplest type of level generator is one that always produces a level with the same characteristic layout. For example, the level generator in *Rogue* always places rooms in a three-by-three grid. Individual levels are distinguished by minor variations in features such as the sizes of rooms. Fixed-layout generators are commonly used because they are simple and easy to implement. However, they tend to produce repetitive results. Some games, such as *Worms: Armageddon* (1999), include multiple fixed layout algorithms to increase the variety of possible levels. Nonetheless, the output of each algorithm quickly becomes familiar to the player.

Random room placement. In random room placement [Baron 2017], rooms are placed at random locations in a level. Typically, rooms are randomly sized rectangles. Rooms that overlap with other rooms many be culled, re-placed elsewhere, or left in place to create irregularly shaped rooms. After all rooms have been placed, hallways can be added between rooms using a pathfinding algorithm.

Agent algorithms. In an agent algorithm [Shaker 2015], an agent moves around the level space making modifications to the level as it goes. The policy of the agent determines its behaviour and thus the shape the level takes. A typical design is an agent that continually moves forward digging a tunnel, with a small chance at each step of turning or placing a room.

Mazes. Maze generation algorithms [Beyer 2017], such as recursive backtracking, Kruskal's algorithm, and Prim's algorithm, can be used to generate a maze from a graph. They can easily be applied to tile-based levels, since a grid of tiles can be trivially reduced to a graph. Generally the goal of a maze generation algorithm is to create a "perfect maze" where there is only one path to the exit and all other paths lead to dead ends. Maze algorithms can also be

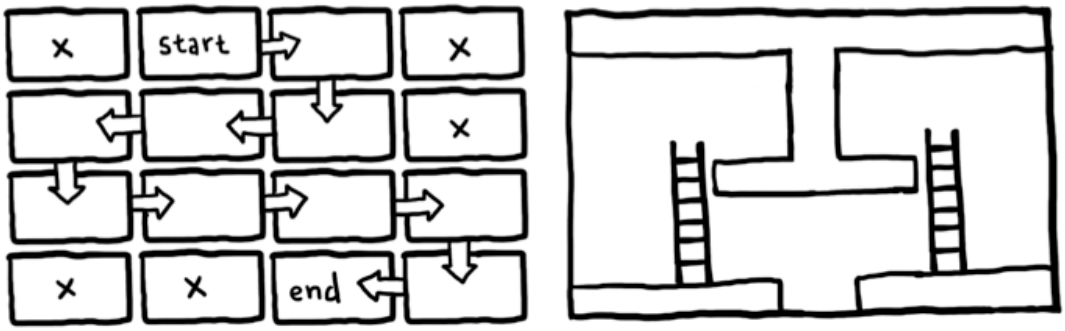


Figure 3-1. Diagrams of the process used to generate levels in *Spelunky* [Yu 2011]. First a path is generated on a room grid (left), and then each room is instantiated using an appropriate template (right).

adapted to create braided mazes (with loops instead of dead ends) or unicursal mazes (which consist of a single winding path).

Cellular automata. Cellular automata [Johnson 2010] can be used to generate organic, cave-like levels in a tile representation. Map tiles are treated as cells and simulated over several time steps. At each time step, every tile may simultaneously change from solid to open or vice versa, according to some policy. A simple policy is that a tile becomes solid if at least five adjacent or diagonally adjacent tiles are solid; otherwise, it becomes open. The interaction of many individual tiles results in the emergence of larger structures.

Space partitioning. A space partitioning method [Shaker 2015] creates different areas in a level by recursively dividing the level space. For example, in binary space partitioning, areas of the level are divided into two sections (either vertically or horizontally). The divisions may be placed at random points so that not all areas are the same size and shape. Once the areas are a suitable size, each area is converted to a room and connections are added between adjacent regions.

Room grid. A room grid is a spatial partition method where a level is divided into a grid of evenly sized rooms. The main advantage of using a room grid for level generation is that the connections between rooms can be determined at a high level and then used to generate the details of individual rooms. One well-known game that uses this method is *Spelunky* (2008), which generates levels on a 4-by-4 grid of rooms [Yu 2011]. First, a path is generated through the room grid from the start room to the end room. Each room on the path is then generated using a template from a library. Templates are selected based on the required connections to adjacent rooms.

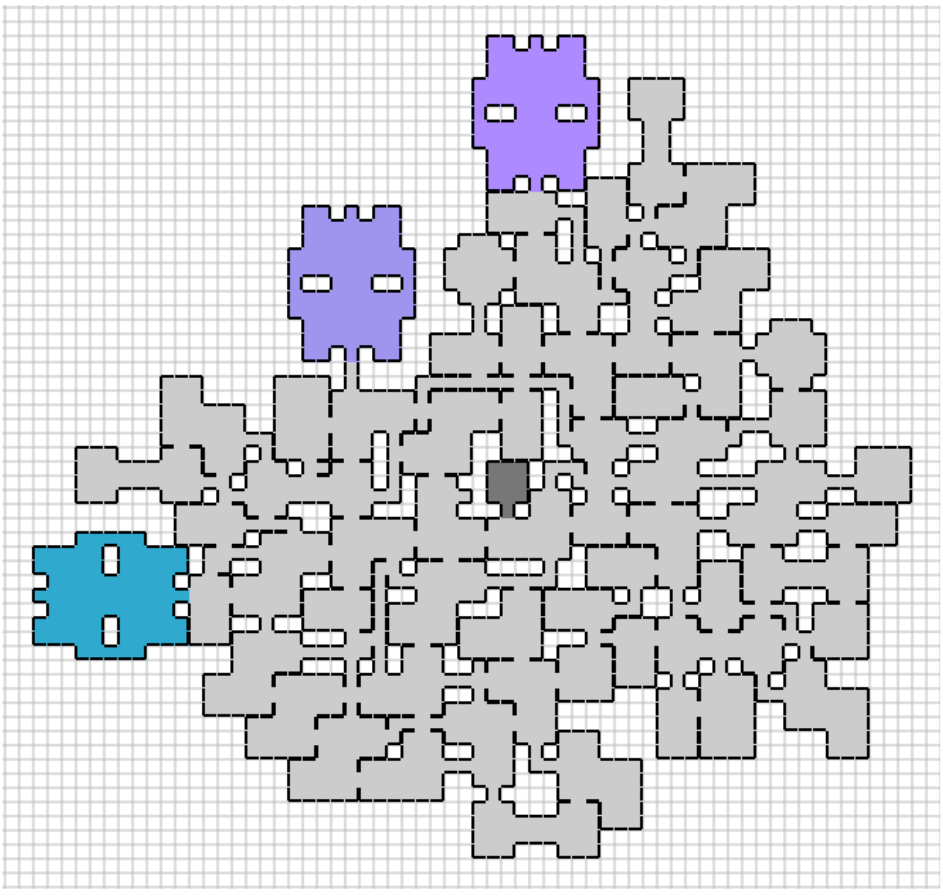


Figure 3-2. Dungeon map generated by Valtchanov and Brown [Valtchanov 2012].

Noise. Noise functions [Beyer 2017] such as Perlin noise, Simplex noise, and Worley noise can be used generate arrays of values in pseudorandom patterns with various properties. These noise functions can be combined and interpreted in various ways to generate random shapes for mountains, caves, and other level features [Tippetts 2011]. For example, *No Man's Sky* uses a mix of different noise functions to generate terrain for alien planets [Compton 2017].

3.1.2 Tree Extension

One appealingly simple way to generate a level is to start with a single room and repeatedly add another room by connecting it to an existing one. We can refer to such a method as a tree extension method because the result is many rooms connected in a tree structure. Each room extends some branch of the tree. If a tree structure is not desired, additional connections between rooms can be added after the fact.

Figure 3-3. Levels for *In Verbis Virtus*, generated by Ferrari [Ferrari 2013].

Valtchanov and Brown [Valtchanov 2012] combined tree extension with a genetic algorithm to generate 2D dungeon levels. Their system uses a library of hand-crafted rooms annotated with possible connection points. A level is represented as a tree showing which child rooms should be joined to each parent room. This representation allows subtrees to be transplanted between multiple trees without disrupting the relative positions of rooms in the subtree.

The authors' genetic algorithm evolves these trees using mutation and crossover operations. Crossover consists of replacing a subtree with a subtree from another tree, and mutations consist of randomly adding, removing, and changing rooms in a tree. When a new tree is generated, rooms that cannot be successfully placed are pruned from the tree, allowing only the valid parts of each level to survive to the next generation. Aside from the connections specified by the tree, additional connections are added when rooms incidentally align so as to allow them.

The authors' fitness function rewards efficient hallways, clusters of connected rooms, minimal wasted space, variety in room selection, and successful placement of required special event rooms. The generator was tested with various parameters to demonstrate its versatility. One experiment involved generating small dungeons very quickly, for use in a runtime generation system. However, levels were only generated as abstract room layouts, never converted to a detailed format suitable for in-game use.

Ferrari [Ferrari 2013] used a tree extension algorithm to generate levels for the first-person adventure game *In Verbis Virtus* (2015). A level is built of three types of rooms: large rectangular rooms, straight hallway segments, and curved hallway segments. After the tree extension phase is complete, additional connections are added to join pairs of disconnected rooms. This is accomplished by adding a short wedge-shaped hallway section to each room, and then placing a straight hallway between the two wedges. Finally, dead-end areas are removed from the level.

Although *In Verbis Virtus* is a 3D game, Ferrari's system only generates levels with 2D layouts. The levels are converted to cave-like 3D environments by placing many rock meshes around the boundary of each room. Additional decorative meshes such as columns and light sources are added, but the method used to place them is not described. Ferrari suggests more detailed architecture could be generated using an L-system or some other type of grammar, but leaves this task for future work.

3.1.3 Occupancy-Regulated Extension

Mawhorter and Mateas [Mawhorter 2010] introduced occupancy-related extension, a level generation method that is similar to tree extension but operates on small overlapping chunks

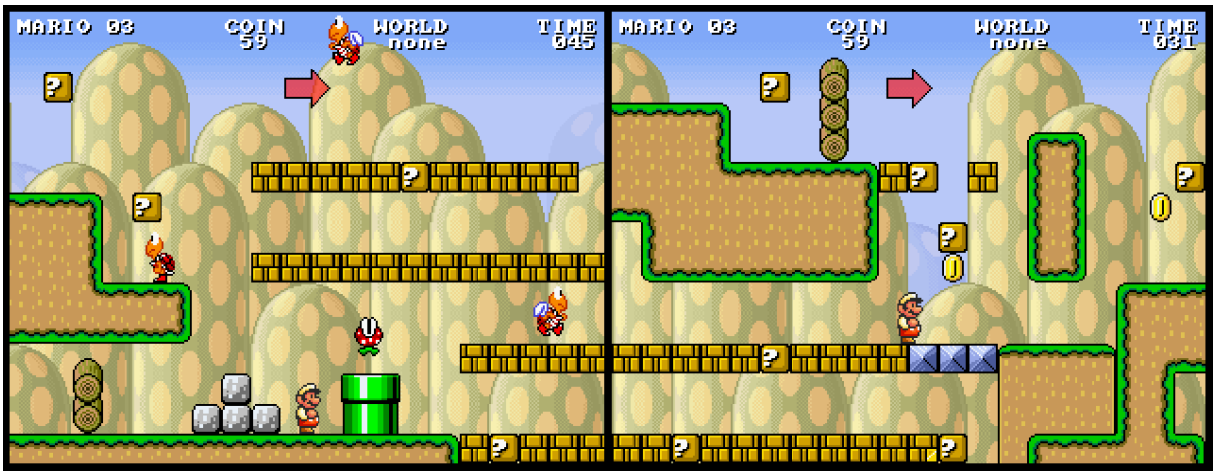


Figure 3-4. Two segments of a level generated by Mawhorter and Mateas [Mawhorter 2010].

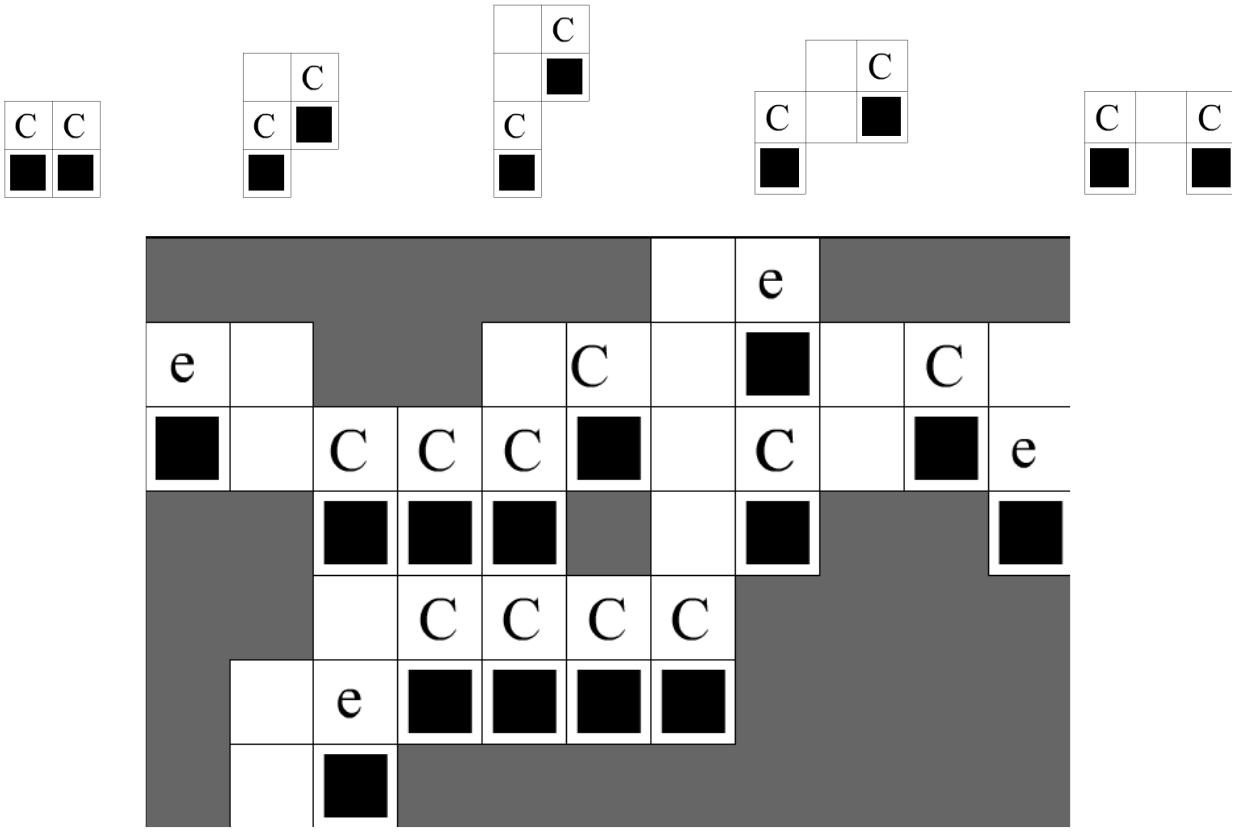


Figure 3-5. The set of valid level chunks (top) and a room generated from these chunks to join four exit tiles (bottom) in Koens' system [Koens 2015]. Anchor tiles are marked "C" while exit tiles are marked "e".

rather than distinct rooms. They used this method to generates levels for *Super Mario Brothers*. In their system, a chunk consists of several tiles including one or more "anchor" tiles. Each chunk promises that it is possible for the player to navigate between the anchor tiles. For example, a chunk could consist of two platforms separated by a gap small enough for Mario to jump over, with an anchor on each side. To build a complete level, multiple chunks from a library are joined together at their anchors. Two chunks can be joined if none of their tiles conflict when they are overlapped with their anchors at the same position. If the chunks in the library are correctly designed, the resulting level will be possible to navigate.

A similar technique was used by Koens [Koens 2015] to generate 2D platformer levels with a maze-like structure. His system starts by breaking the level down into a grid of rooms and marking the exit tiles which connect adjacent rooms. Each room is then filled in using an algorithm that combines occupancy regulated extension with a depth-first search. The goal is to create a navigable path between all the room's exits. The algorithm starts by placing a single chunk with an anchor at one of the exit tiles. Additional chunks are then added by chaining together the anchors of compatible chunks. The generator continues adding chunks until either all the exit tiles of the room have been connected, or it builds itself into a corner and is unable to add more chunks. In this case, it backtracks by removing the most recently placed chunk and testing alternate placements, effectively searching the tree of possible chunk arrangements in a depth-first manner.

A problem with Koens' algorithm is that it may spend a long time exhaustively searching an area of the search space that does not contain a solution. Koens addresses this problem by resetting the search algorithm if it fails to find a solution within a time limit. Using this method, complete levels are generated at runtime in a matter of seconds.

3.1.4 Graph Grammars

Graph grammars have attracted some attention as a method of generating level structures. A level can be represented as a graph in which the nodes represent traversable spaces (such as rooms) and the edges represent connections between rooms (such as hallways). Such a graph can then be manipulated using a graph grammar. The advantage of this approach is that a graph grammar can control the topology of the generated level more readily than a tree expansion method. Features such as looping paths are easily created by rules of the grammar. The

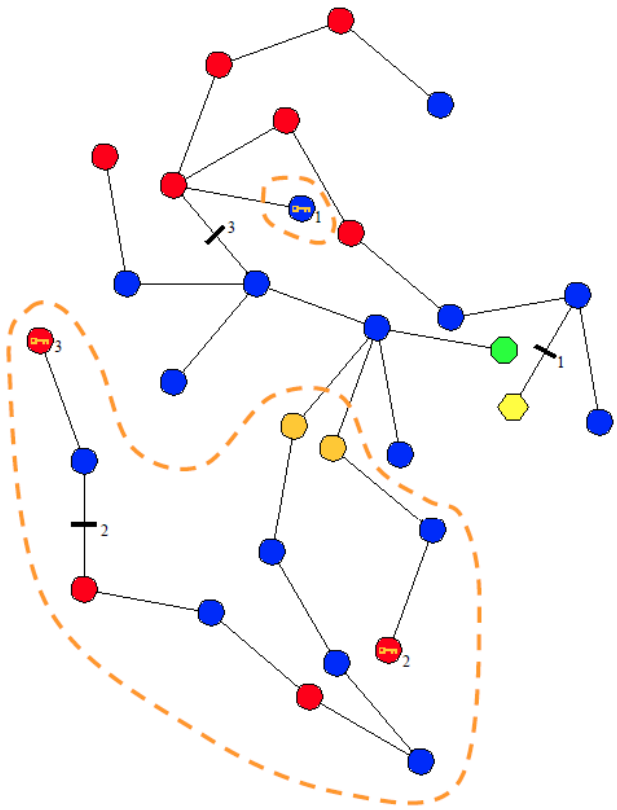


Figure 3-6. Level graph generated by Adams [Adams 2002].

disadvantage is that the output of a graph grammar is only a graph, not a finished level. Some other method must be used to convert the graph into a physical arrangement of rooms in space.

Early work on level generation with graph grammars was carried out by Adams [Adams 2002], who generated graphs of levels for the first-person shooter *Star Wars: Dark Forces* (1995). The nodes of each graph are annotated with information about the obstacles and rewards found in each room of the level. However, levels were not converted from a graph to a playable format. Adams' generator combines a graph grammar with a search approach. During each iteration of the production process, the generator uses a randomized hill-climbing algorithm to select which rule to apply. The algorithm tests all the rules that can be applied to the current graph, discards those which do not improve the level's fitness score, and applies a randomly selected rule from among those remaining. A level's fitness score is based on an abstract simulation of gameplay. A level is considered fun if the simulated player runs low on health and energy but does not die.

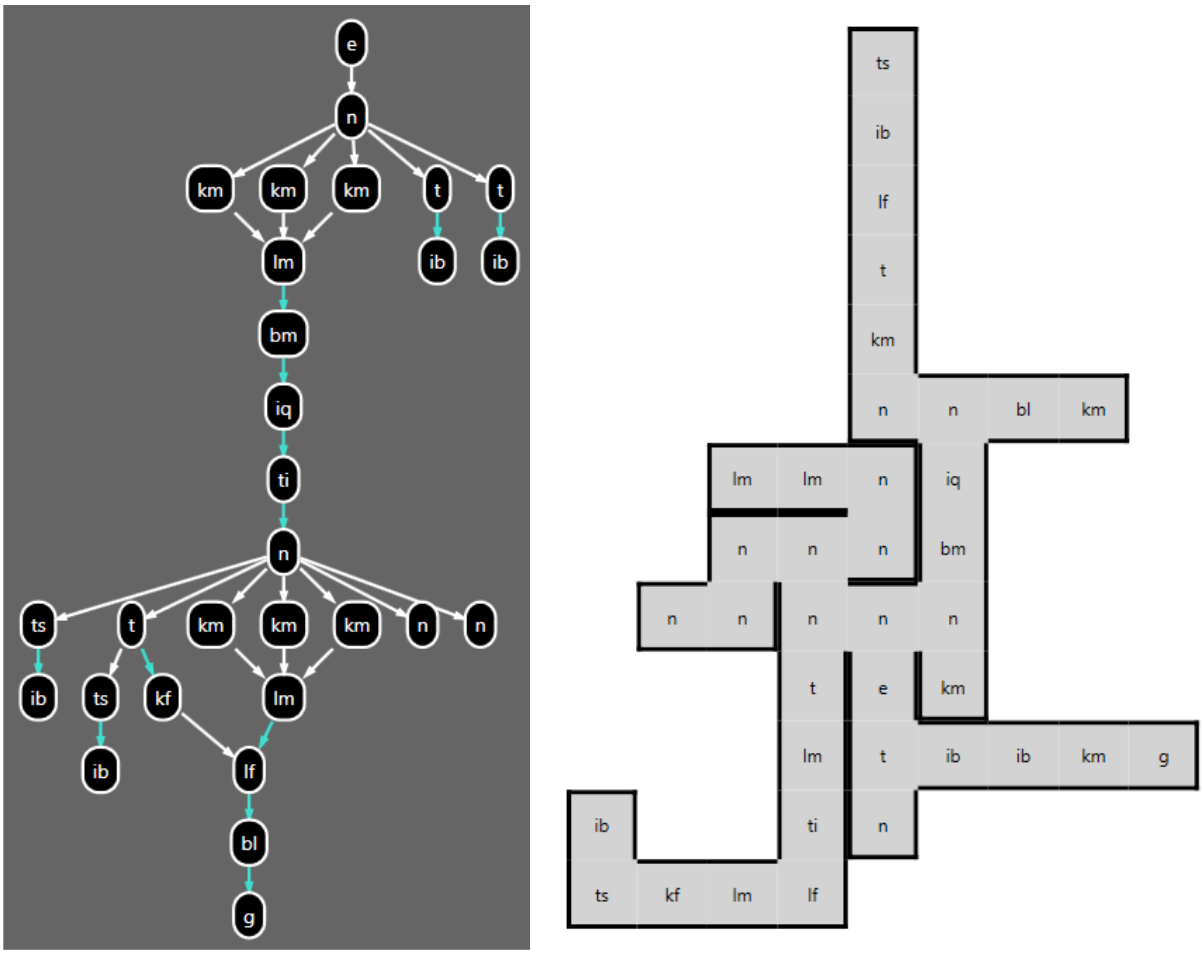


Figure 3-7. A mission graph and the corresponding level layout generated by Lavender [Lavender 2015].

Dormans [Dormans 2010] developed an alternative approach to level generation based studying the structure of dungeons in *The Legend of Zelda: The Twilight Princess* (2006). He observed that each dungeon contains two separate structures: a mission, and the space in which the mission takes place. Dormans' approach is to first generate a mission using a graph grammar, and then generate a level based on the mission. The mission is a directed graph, where the vertices represent actions for the player to take, and the edges represent the order in which these actions can be performed. The generated level must support the action of the mission; for example, if the mission requires the player to unlock a door before fighting an enemy, the design of the level must prevent the player from reaching the monster without opening the door. This does not imply, however, that the topology of the level map must be isomorphic to that of the mission graph.

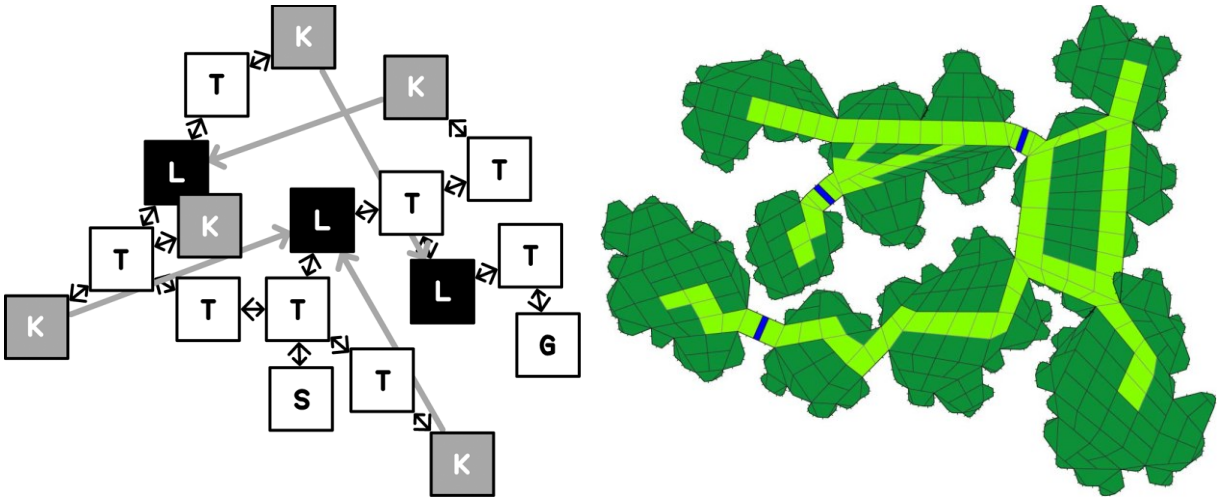


Figure 3-8. A mission graph and the corresponding level generated by Dormans and Bakkes [Dormans 2011].

In Dormans' original paper, a graph traversal algorithm and a shape grammar are used to generate a level layout from the mission graph. Each type of node in the mission graph corresponds to one or more rules in the shape grammar. As the mission graph is traversed, the corresponding rules are applied to the level map. The rules themselves implement a tree extension algorithm. Dormans' graph traversal algorithm was subsequently refined by Lavender [Lavender 2015], who used it to generate playable 2D levels for an open-source clone of *The Legend of Zelda: A Link to the Past* (1991). She identified several cases in which the level fails to produce a valid layout. The underlying problem is that the tree extension algorithm fills space randomly, so it may build itself into a corner and be unable to place rooms required by the mission graph.

In another follow-up paper, Dormans and Bakkes [Dormans 2011] introduced an alternate method for generating a 2D level map from a mission graph. In this new method, the mission graph is used directly as a graph of the level topology. Tasks in the mission graph become rooms in the level, and relationships between tasks become connections between rooms. Rooms are laid out in space using a graph layout method in which the edges of the graph are treated as spring constraints and overlapping nodes of the graph repel one another. This method is not entirely reliable as it sometimes produces overlapping edges. The authors get around this problem by replacing the offending connections with teleporters. Once a layout has been decided, rooms are instantiated as quadrilaterals and a shape grammar is used to provide embellishment to the edges of each room. Lavender [Lavender 2015] has correctly noted that

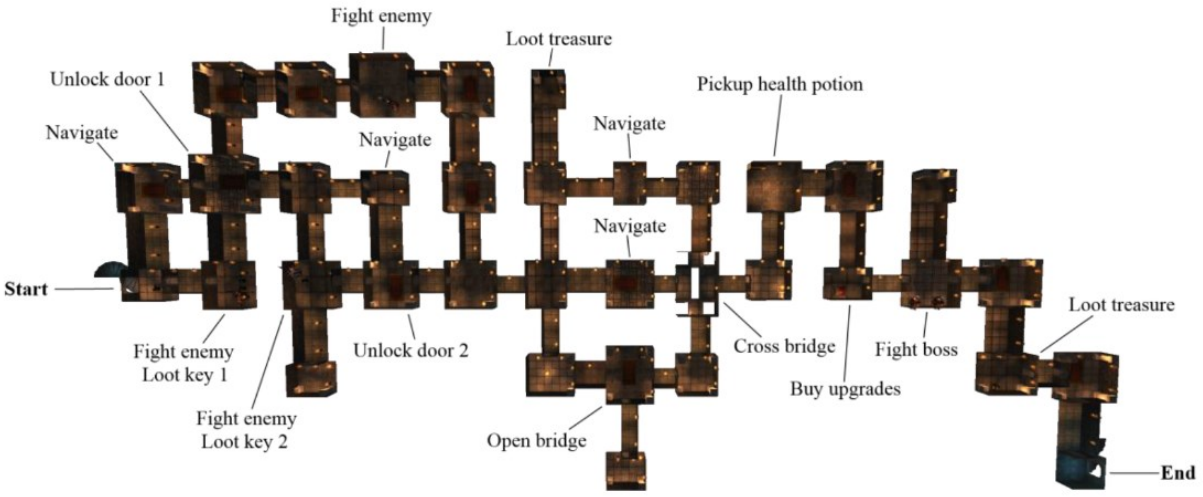


Figure 3-9. *Dwarf Quest* level generated by van der Linden [Van der Linden 2013].

this graph layout method undermines the original premise of Dormans' research, since it cannot produce a level with separate mission and space structures like a *Zelda* dungeon.

Van der Linden [van der Linden 2013] developed a system that generates levels for the turn-based RPG *Dwarf Quest* (2013). Levels in *Dwarf Quest* consist of small rectangular rooms laid out on a 2D grid. Much like Dormans and Bakkes' system, Van der Linden's generator first generates a mission graph using a graph grammar and then converts the mission graph to a level map. However, van der Linden's system introduces two notable innovations. The first is an interface that allows a human designer to specify the rules of the graph grammar for mission generation, and constraints on how the rules may be applied. For example, a certain rule might only be used when the difficulty parameter is above a certain level.

The second innovation is a layout solver which can convert a graph to a series of rooms on a 2D grid. First, the graph is pre-processed to remove crossed edges and to ensure that no single node has more than four edges. Next, a sophisticated constraint-solving algorithm places a room in the dungeon grid for each node in the graph. The algorithm deduces which rooms must have greater or lesser *y*-coordinates than one another, then systematically assigns rooms to appropriate positions in the grid. Additional empty rooms must sometimes be added to ensure the proper connectivity of rooms in the dungeon. Finally, the level is post-processed to shorten long hallways and bring rooms closer together.

As in the work of Dormans and Bakkes, the level topography is copied from the mission graph. This technique sometimes has an odd effect on the levels generated – for example, if the

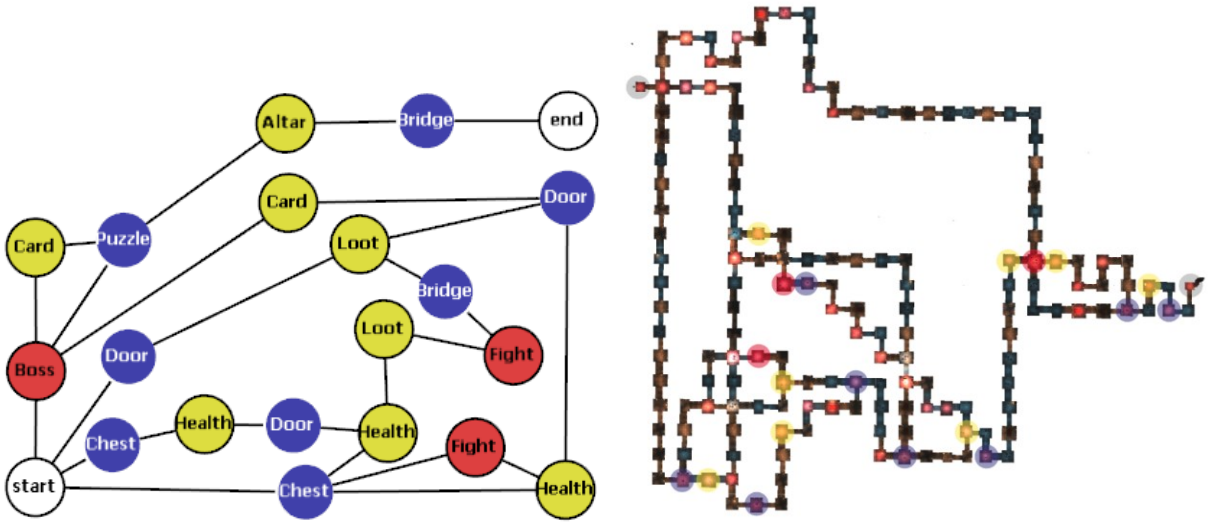


Figure 3-10. Mission graph evolved by Karavolos, Liapis, and Yannakakis [Karavolos 2016] and the corresponding level produced using van der Linden's layout solver.

player is supposed to perform the "navigate" action before finding a door, an extra room is added for them to "navigate" in. A human designer might instead place the door in an existing area of the dungeon, forcing the player to navigate previously explored areas.

Karavolos and others have tested several variants on van der Linden's system. Karavolos, Bouwer, and Bidarra [Karavolos 2015] developed a version in which a human designer provides the initial mission graph and can edit the resulting level at certain stages in the generation process. This makes the generator more controllable. Karavolos, Liapis, and Yannakakis [Karavolos 2016] further modified this variant by replacing the human designer with an evolutionary algorithm. The algorithm evolves an abstract mission graph using mutations similar to the operations of a graph grammar and a fitness function based on measurements of the mission's length, complexity, and balance. Generation then proceeds using a graph grammar and van der Linden's layout solver. The results unfortunately reveal the limitations of the layout solver. It performs poorly when presented with arbitrary graphs, especially highly connected graphs. Rather than finding an elegant layout, it resorts to placing long strings of empty rooms.

It may be observed that none of the surveyed methods provides a reliable general solution to the problem of converting a graph of a level to a valid and pleasing level design. These poor results may point to an underlying problem with the process of first generating a level as a graph and then attempting to convert it to a spatial layout. If the graph is generated without regard for

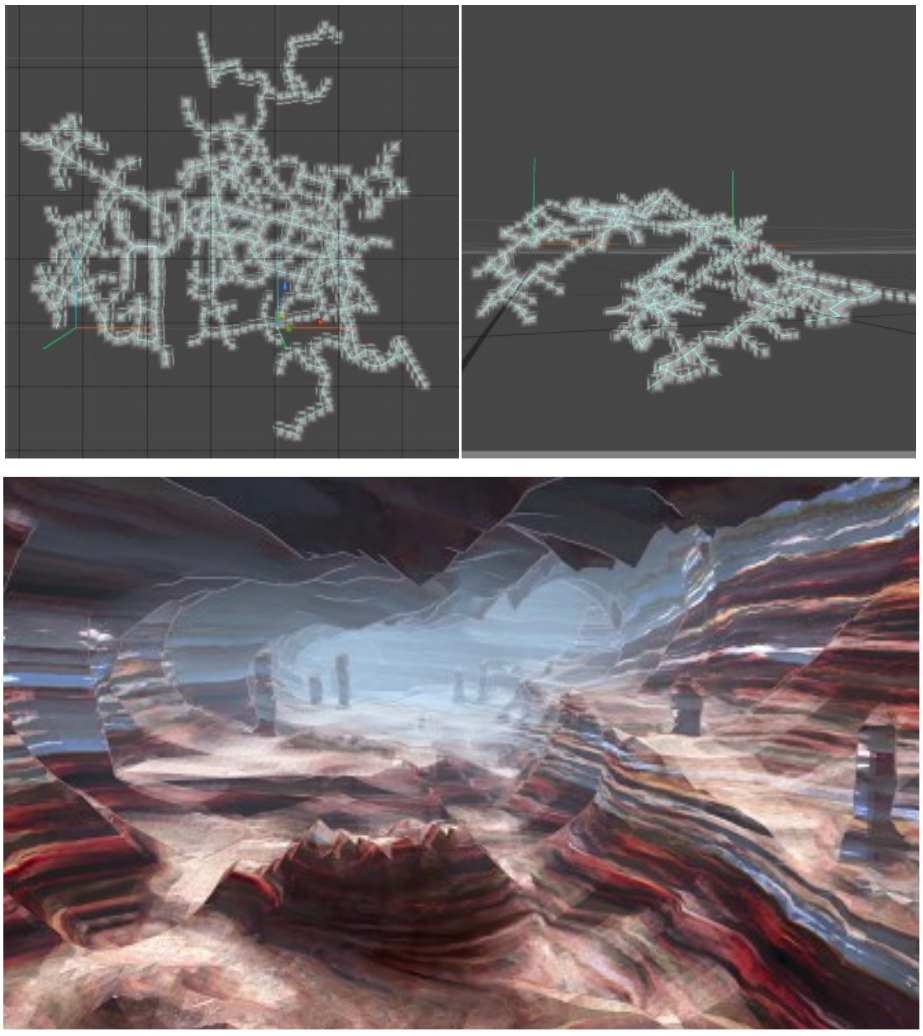


Figure 3-11. Cave structures as trees of lines in 3D space (top) and finished cave geometry (bottom) generated by Mark et al. [Mark 2015].

its eventual spatial layout, it is entirely possible to generate a graph for which no satisfying layout exists. A human level designer would not normally encounter this problem because it would not occur to most people to design a level as an abstract graph without some idea how the pieces would physically fit together. Instead, a human designer would attempt to work out the level's layout and its connectivity simultaneously. In light of these observations, we chose not to use a graph grammar approach in our own level generator. The benefits of a graph grammar over a simpler tree extension method are not apparent.

3.1.5 Caves with L-Systems

Mark et al. [Mark 2015] used L-systems to generates superficially realistic 3D caves for use as game levels. The authors' approach is inspired by the creation of caves in nature by water passing through cracks in limestone. Their system generates a tree of lines in 3D space using an L-system with randomly generated rules. These resulting lines are converted to 3D geometry using a voxel approach. Distorted spheres are run along the lines in the cave, carving out passages. The resulting voxel geometry is then converted to polygons using a marching cubes algorithm. A user study found that players generally enjoyed exploring the generated caves, although some users found the caves monotonous and suggested that there should be more variation in tunnel widths and textures.

Although the authors' approach is effective for creating organic, cave-like environments, it is not clear how it could be applied to manmade architecture. Shape grammars have been more effective for generating architectural forms.

3.1.6 Rhythm and Flow

Some approaches to level generation have been based on the idea that the arrangement of content in a level should follow a particular pattern or rhythm. These approaches are limited to levels with a linear design, where the player is sure to encounter the content in the intended order.

Smith et al. [Smith 2009; Smith 2011] generated levels for a *Mario*-like 2D platform game using a rhythm approach. The premise of this work is that levels will be more enjoyable if the button presses required to win form a repetitive but irregular rhythm. The authors generated such levels using two grammars. The first grammar generates a rhythm, i.e., a list of player actions and the number of seconds for which each action should be performed. The second grammar places obstacles that can be overcome by these rhythmic actions. The relationship between player actions and the placement of obstacles is based on the physical capabilities of the player character – movement speed, jump height, and so on. While this method is intriguing, its impact on player enjoyment has not yet been empirically studied.

Sorenson [Sorenson 2010] used a genetic algorithm to generate *Mario* levels with a fitness function designed to estimate challenge-based fun. This fitness function is based on Csikszentmihalyi's concept of flow, which suggests that people enjoy being challenged at the limit of their abilities, and on the Yerkes-Dodson law, which suggests that optimal human

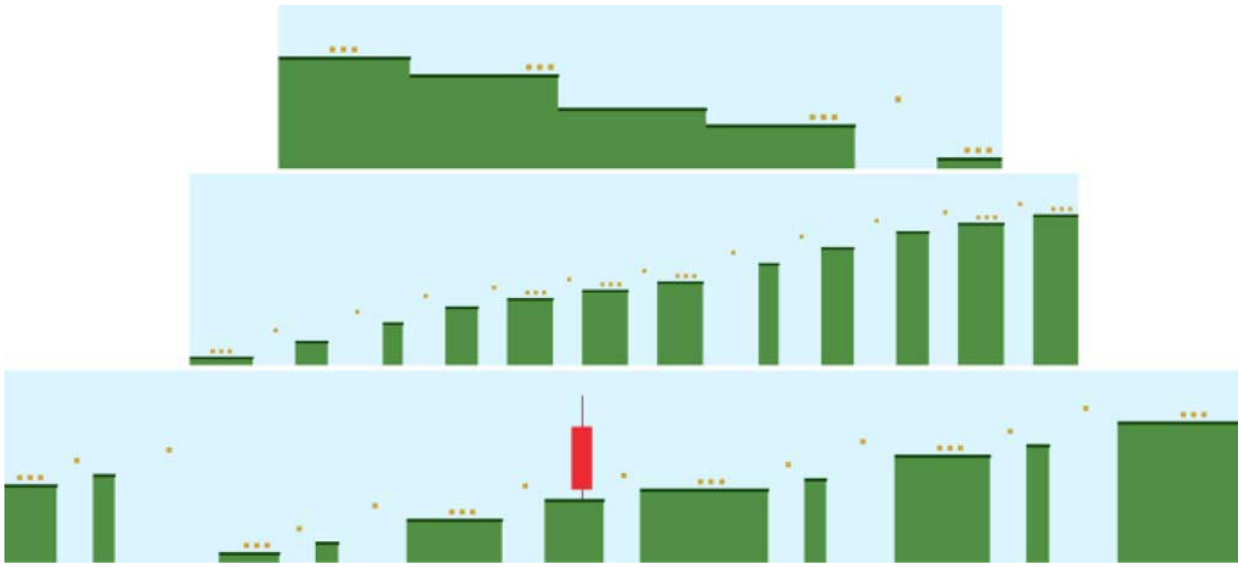


Figure 3-12. Level segments generated by Smith et al. [Smith 2011].

Figure 3-13. Segment of a Mario level generated by Sorenson [Sorenson 2010].

performance occurs at a certain level of arousal (mental and emotional involvement). Sorenson treats the difficulty of a game level as a continuous function that can be sampled at an instantaneous point in a level or integrated over a section of the level. As his main example, Sorenson uses *Super Mario Bros.*, where he defines difficulty based on the percentage of possible jump arcs that will allow Mario to clear a given hazard.

Based on an analysis of 28 *Mario* levels, which are assumed to represent good level design, Sorenson concludes that a good level should contain a series of "rhythm groups." Each rhythm group is a level segment that starts at a low difficulty, increases to a high difficulty, and then returns to a low difficulty. For example, a rhythm group in a *Mario* game could consist of some safe ground, followed by a series of deadly pits, and then some more safe ground. It is assumed that a player will experience increasing anxiety throughout a rhythm group and then relax again at the end. Sorenson's algorithm segments a level into a series of rhythm groups by analyzing the shape of its difficulty function. The difficulty of each rhythm group is

independently integrated and the result is compared to a constant representing the player's skill level. The closer the integrated difficulty of each rhythm group is to the skill constant, the better the level is scored by the fitness function.

Sorenson's fitness function is certainly clever and sophisticated, but it depends on many assumptions about skill, difficulty, anxiety, and fun. It requires a reliable way to measure the difficulty at each point in a level, which may be difficult in many games. It also requires that the player's skill level be accurately translated into a constant on an arbitrary scale. Furthermore, no empirical study has been performed to establish a correlation between the results of the fitness function and the amount of fun experienced by human players. An experiment was performed to confirm that the fitness function could distinguish with high accuracy between real *Mario* levels and test levels with deliberately poor designs.

Sorenson generates *Mario* levels using a feasible/infeasible two-population genetic algorithm and a constraint solver. A given level can be categorized as feasible or infeasible using constraints that measure whether it is solvable and free from errors. Feasible levels are evolved using the fun-based fitness function described above. If a level is found to be infeasible, the constraint solver is used to perform simple repairs. If these repairs fail, it is moved to the infeasible population. Levels in the infeasible population are evolved using an alternative fitness function that measures how egregiously they violate feasibility constraints. Those that evolve to a feasible state are returned to the feasible population.

For this genetic algorithm, a level is represented as an unordered set of design elements. Each design element consists of either a single game object, such as a coin, or an organized group of game objects, such as a staircase of blocks. Mutations consists of adding and removing design elements or altering their properties. A crossover operation consists of physically splitting two levels at a horizontal or vertical plane and recombining the halves.

The resulting *Mario* levels could pass for the work of a human designer, though perhaps not an especially inspired one. Sorenson also applies his algorithm to the generation of dungeons for a loosely *Zelda*-inspired game. This work is much more preliminary in nature and the results are not impressive. Sorenson's fitness function is designed for a linear series of challenges and does not translate easily or directly to a non-linear 2D world. His level representation is also a poor fit for a 2D dungeon because so much of the search space consists of infeasible dungeons with disconnected rooms.

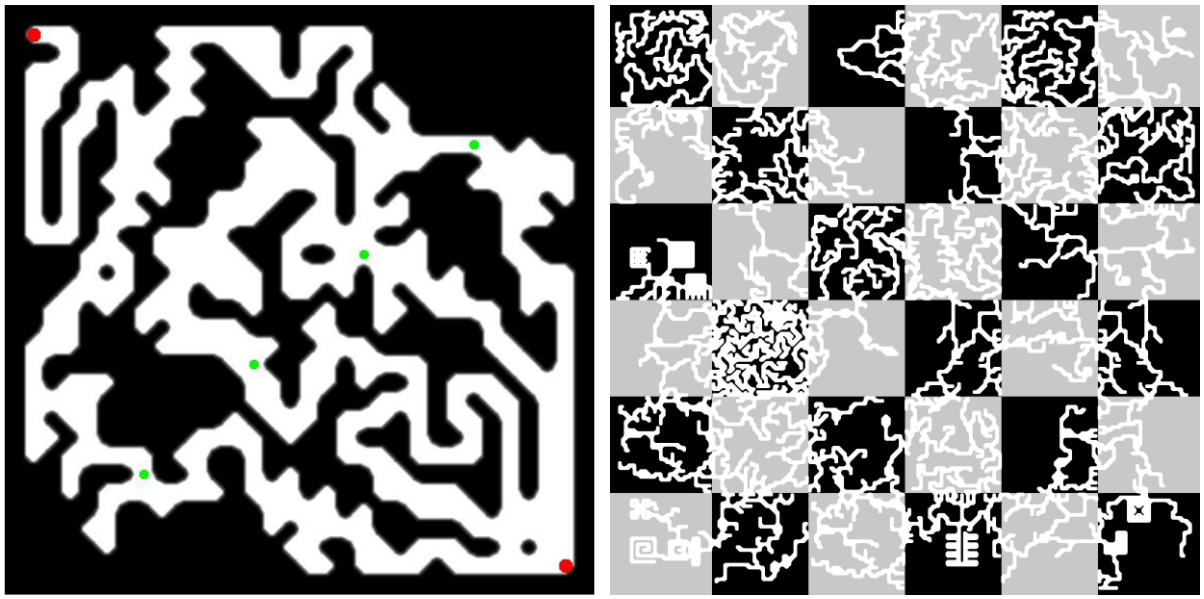


Figure 3-14. Maze-like levels generated by Ashlock et al. [Ashlock 2011a] (left) and McGuiness et al. [McGuiness 2011b] (right).

3.1.7 Evolution of Maze-Like Levels

Ashlock and his collaborators have applied genetic algorithms to the generation of mazes with various properties. In Ashlock's initial paper [Ashlock 2010], the fitness function is based on the length of the shortest path through a maze from the start point to the end point. When the path length is used directly as the fitness function, the algorithm produces trivial mazes with long winding paths and no branches. Such an arrangement maximizes the path length that can be achieved in a limited area. When the fitness function is changed to reward mazes where the path length is close to a manually selected constant, the algorithm produces more interesting mazes, with branches and dead ends.

Ashlock et al. continue this research by comparing several different representations and fitness functions for maze-like game levels [Ashlock 2011a]. The three types of representations described are a "direct" representation, an "indirect positive" representation, and an "indirect negative" representation. The direct representation is a grid of open and wall tiles. In the indirect positive representation, the lengths, positions, and orientations of solid walls are stored, and the remaining space is open. In the indirect negative representation, the positions and dimensions of open rooms are stored, and the remaining space is solid. Each representation

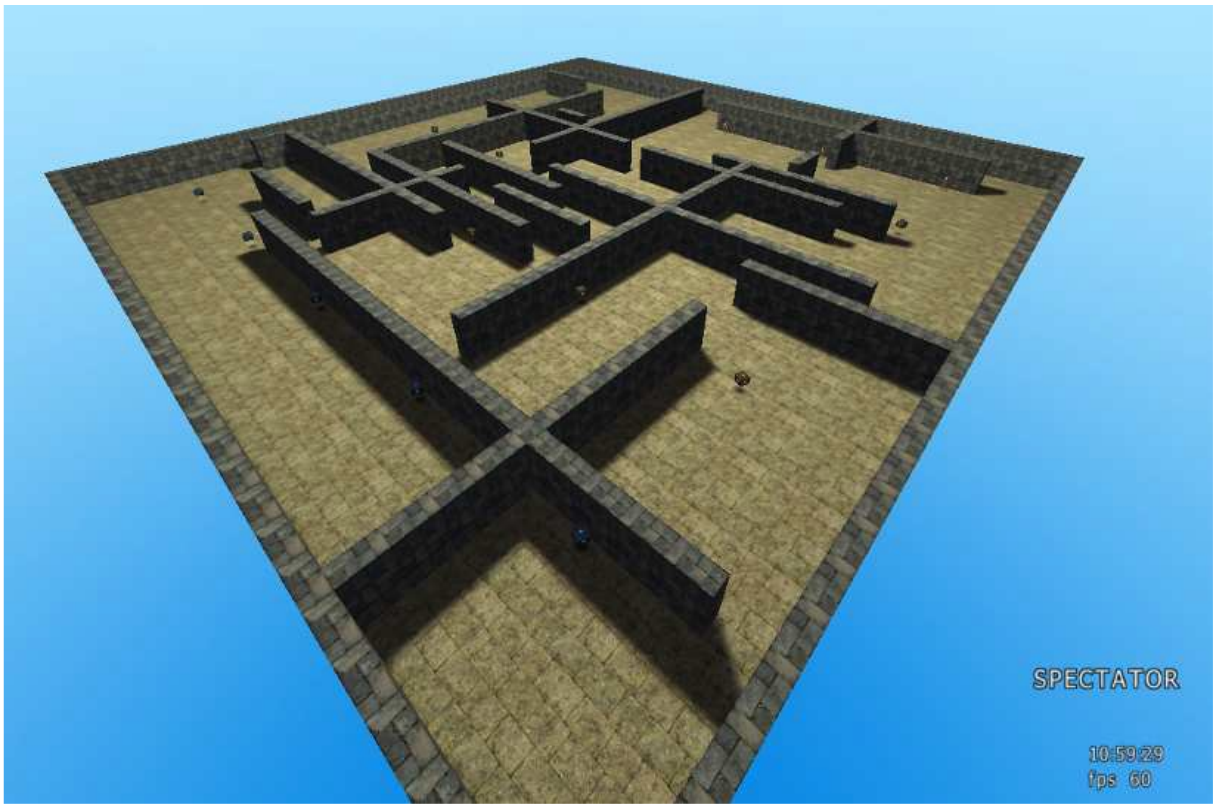


Figure 3-15. Shooter map for *Cube 2: Sauerbraten* generated by Cardamone et al. [Cardamone 2011].

creates mazes with a characteristic style, but the topological features of the mazes depend more upon the choice of fitness function than the choice of representation. Mazes with an indirect representation are converted to a direct representation before being assessed by the fitness function.

In this system, maze generation is guided by checkpoints, which are markers placed at fixed positions in the maze area before mazes are generated. Dynamic programming algorithms are used to determine which checkpoints appear on the shortest path to each point in a maze. Certain types of special points can then be detected, such as "reconvergence" points where two paths meet after passing different checkpoints. The authors use these special points to define various fitness functions. For example, one fitness function rewards mazes based on the length of the shortest path to the reconvergence point for each pair of checkpoints. This can produce mazes where each checkpoint appears on a separate path to the exit.

These techniques have been further explored in additional papers by the same authors. One paper considers mazes that can be traversed by agents with different capabilities [Ashlock

2011b]. For example, a maze might contain walls of fire and water that certain creatures can pass through, resulting in several embedded mazes within a single maze. Tactically interesting maps can be generated by using a different fitness function for each embedded maze. Another paper demonstrates how the generation of a large maze level can be decomposed into the generation of many smaller mazes [McGuiness 2011a]. Distinct areas in the large maze can be created by using different fitness functions for sub-mazes. The authors suggest that their checkpoint analysis algorithms could be adapted to place game content such as traps and monsters at key points in a level. Another paper discusses how the algorithm could be modified to generate symmetrical mazes or to incorporate hand-crafted content [McGuiness 2011b].

All these maze generation techniques assume a 2D, tile-based representation of a game level. They are subject to many of the same limitations as roguelike dungeon generators. Most of the fitness functions encourage the evolution of complex and twisting paths by rewarding long path lengths in a small area. This technique implicitly assumes that a maze will be generated within a flat grid of a tightly limited size. Such fitness functions are less appropriate for generating levels where this assumption does not hold, such as sparse and non-tile-based 3D levels. On the other hand, the dynamic programming algorithms for checkpoint analysis could be adapted to any type of graph, and could be useful even outside the context of an evolutionary algorithm.

3.1.8 Other Evolutionary Approaches

This subsection describes a number of interesting studies of level generation with evolutionary algorithms that do not fall into any of the earlier categories.

Cardamone et al. compared several different representations for a level map in the context of *Cube 2: Sauerbraten* (2004), an open-source first-person shooter game [Cardamone 2011]. This game supports 3D gameplay, but the maps generated all consist of 2D mazes. Four different representations were compared: all white, all black, grid, and random digger. The all-white and all-black representations are equivalent to the indirect positive and indirect negative representations of Ashlock et al. The grid representation divides the map area into a nine-by-nine grid and encodes whether there is a wall between each pair of adjacent cells. The random digger representation is stochastic in nature, as it encodes the parameters used to control the random behaviour of digger agent.

Figure 3-16. *Prince of Persia* level generated by Mourato et al. [Mourato 2011].

The authors use the same fitness function to compare the maps evolved using each representation. This fitness function is based on the total area of the map (larger is considered better) and the length of gunfights between AI bots (longer fights are considered better). The grid representation and the all-white representation were found to produce the highest-scoring maps. However, these results depend on the behaviour of the *Cube 2* AI bot, and there is little evidence provided that they reflect the overall quality of the generated levels or the overall potential of the four representations studied.

Mourato at al. [Mourato 2011] generated levels for the platformer game *Prince of Persia* (1989) using a genetic algorithm with a tile-based level representation. In their method, each generated level is analyzed to find all possible paths through the level and the length of the path to each point. The level is then scored according to several heuristics: the number of different paths (which should be neither too high nor too low), the number of tiles that are not visited on any path, the balance between the number of tiles of different types, the length of the shortest path to the exit, and the percentage of the level area used by the shortest path. The weighted average of these heuristics is used as a fitness function. Information from the heuristic functions is also used to guide the operations of the genetic algorithm. Tiles identified as problematic by

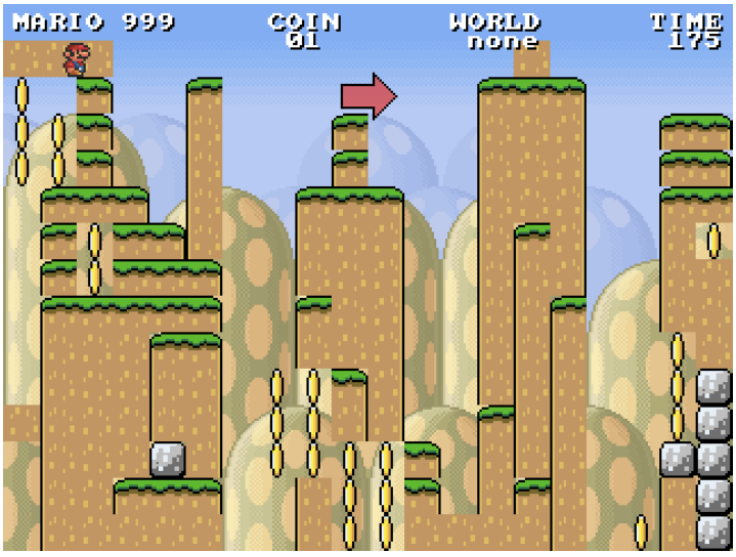


Figure 3-17. Mario level designed by a generator evolved by Kerssemakers et al. [Kerssemakers 2012].

the heuristics are more likely to be replaced by mutation, and tiles identified as being on the main path through a level are preferentially preserved during crossover.

Kerssemakers et al. [Kerssemakers 2012] present a "procedural procedural level generator generator" for *Super Mario Bros*. An interactive genetic algorithm is used to generate agent-based generators which are used to generate levels. Each generator consists of several agents that place level tiles in random, wave-like patterns. An initial population of generators is evolved using a fitness function that measures playability based on how far a simulated player is able to travel through the levels they produce. Once many generators that reliably produce playable levels have been evolved, they are turned over to a human user who guides their further evolution. The handiwork of each generator is shown to the user, who chooses which generators survive to the next generation.

This two-layered process can produce levels with a wide variety of features. Unfortunately, most of the levels look like the work of obsessive maniacs; they are full of oddly specific small patterns repeated over and over for no apparent purpose. The generator agents do not take context into account when placing tiles and are not sophisticated enough to produce complex structures individually. A larger problem with using a stochastic generator as a representation for an evolutionary algorithm is that this arrangement prevents the algorithm from converging on a single good level. Instead, it must evolve a generator that reliably produces playable levels. Inspection of the results suggests that the most reliable generators are those that

Figure 3-18. Multiplayer shooter levels generated by Bhojan and Wong [Bhojan 2014].

make dense and repetitive levels. Because of these limitations, this research is of greater theoretical interest than practical application.

Liapis et al. [Liapis 2013] present the results of experiments with a mixed-initiative approach to level design. This is an approach where a human designer actively collaborate with a PCG algorithm. The authors introduce a software tool named the Sentient Sketchbook, which can be used to edit maps for a generic real-time strategy game. The tool analyzes the user's map

according to several fitness functions, then uses an evolutionary algorithm to generate suggested improvements to the map. The user can accept or reject any of the tool's suggestions.

A user study was run in which 5 experienced designers used the tool over a total of 24 sessions. The designers only occasionally accepted the generator's suggestions, on average just under once per session. Designers indicated in comments the generator's suggestions, though interesting, were often "messy," lacked symmetry compared to human designs, and did not relate to what they were trying to do. These results suggest the authors' fitness functions did not capture all the qualities that the human designers considered important in a level design.

Bhojan and Wong [Bhojan 2014] designed a system that generates multiplayer maps for a first-person shooter game in a matter of seconds by combining a space partitioning algorithm with an evolutionary algorithm. The spatial portioning algorithm is based on domain knowledge about multiplayer map design, so the initial designs are already reasonably good. This algorithm is used to generate an initial population of 3 maps. An evolutionary algorithm is then run using only 9 new maps in each generation, a very small population. Mutations consists of deleting part of a map and regenerating it using the space partitioning algorithm. A simple fitness function assigns up to 1 point for each of 4 design objectives that a map fulfills. In experiments, a fitness score close to 4.0 was obtained after an average of 26 generations. A user study found that participants rated the generated levels as acceptable (though not outstanding) on several traits. Although used for a 3D shooter game, the map layouts generated by this system are purely two-dimensional.

Font et al. [Font 2016] generated graphs of *Zelda*-style dungeons with lock-and-key puzzles using evolutionary algorithms constrained by context-free grammars. In their method, a context-free grammar is generated which describes all valid structures for a level. An evolutionary algorithm is then used to select which grammar rules to apply. The grammars used are traditional string grammars, not graph grammars, but the output of the generator can be interpreted as a graph. A unique feature of this system is that grammars are not hand-authored, but generated automatically based on parameters set by a human designer. For instance, if the designer specifies that the dungeon should have three zones with three rooms in each zone, a grammar is generated that describes all such dungeons. The generated level graphs are not translated into a playable representation.

Figure 3-19. Interactive dungeon generator developed by Baldwin and Holmberg [Baldwin 2017].

Baldwin and Holmberg [Baldwin 2017] developed a system to generate tile-based dungeon levels using evolutionary algorithms with a fitness function based on a series of design patterns identified by Dahlskog et al. [Dahlskog 2015]. These design patterns include both micro-level patterns, which are specific arrangements of tiles such as chambers and corridors, and meso-level patterns, which are based on the relationships between micro-level patterns. In Baldwin and Holmberg's system, the desired frequency of each pattern can be set as a parameter by a human designer. These parameters control the evolution of the dungeon. The system supports a mixed-initiative workflow where the human designer can edit the level manually or accept suggestions from the generation algorithm. A user study was performed in which five game developers built levels with the system. The users found the tool lacking in several capabilities, and found it difficult to design meaningful levels using abstract tiles in a limited space, but generally agreed that a tool of this type could be useful to enhance creativity and provide new design ideas for a designer. In several cases, the generator failed to detect a design pattern the user was attempting to use, suggesting that a better library of patterns was required.

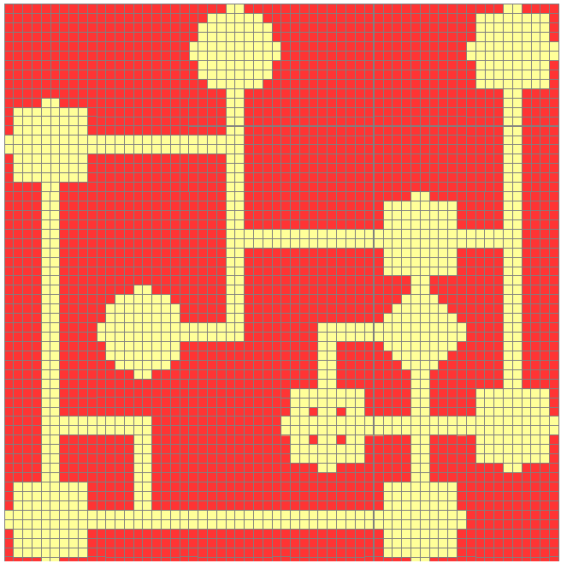


Figure 3-20. Dungeon region generated in a two-step process by Smith and Bryson [Smith 2014].

3.1.9 Answer Set Programming

Answer set programming is a form of logical programming that can be used to find a solution to a constrained problem [Lifschitz 2008]. The properties of the desired solution must be described in a series of bulletproof logical propositions. A general-purpose constraint solver can then be used to seek a solution. Constraint solvers use many heuristics to find solutions more quickly than a brute force algorithm, although they remain very slow compared to most of the other methods discussed in this chapter.

Horswill and Foged [Horswill 2012] used constraint solving to procedurally populate a game level with enemies and pickups. Their constraints are based on the expected amount of health and ammo a given enemy will cost the player to defeat. Using these constraints, a level can be populated such that at least one path through the level is survivable, or so that every path through the level is survivable. This method does not generate a level map, only adds enemies and pickups to an existing level represented as a graph.

Smith and Mateas [Smith 2011b] used answer set programming to generate small (6-by-6) mazes. Smith and Bryson [Smith 2014] applied the method to roguelike dungeon maps, but found that generation times increased prohibitively when the method was applied to larger maps. They addressed this limitation by first generating an entire level at a low resolution, then

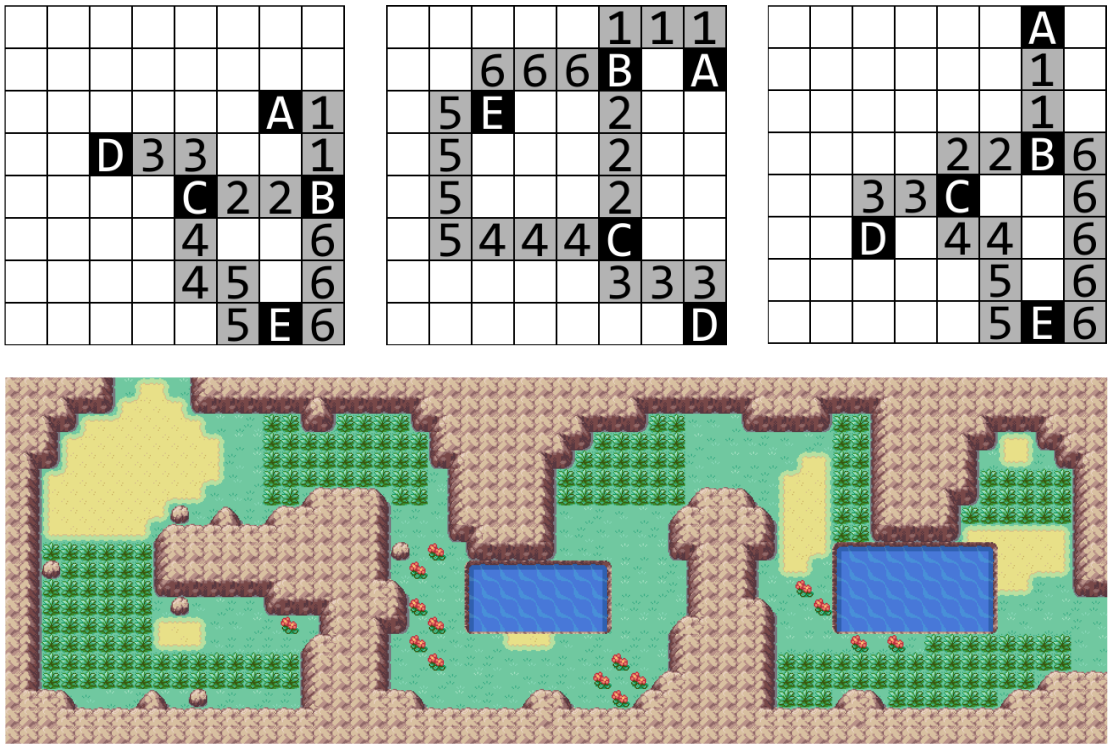


Figure 3-21. *Pokémon* region maps (top) and an individual level map (bottom) generated by Beyer [Beyer 2017].

separately generating each room of the dungeon at a higher resolution. The dungeons generated are simple tile-based maps with equally sized square rooms.

Beyer [Beyer 2017] used answer set programming as part of a system to generate stories and worlds for an RPG similar to *Pokémon FireRed and LeafGreen* (2004). His system first generates a story consisting of a list of events and the names of locations where they will take place. A world map is then generated by laying out the locations from the story on a square grid using answer set programming. The requirements for a valid map are that every location can be reached from the start city, locations can be visited in the order they appear in the story, each location is adjacent to at most three other locations, and the map contains at least one cycle. Once the overall layout has been determined, an individual level map for each location on the world map is generated using a number of techniques such as space partitioning, cellular automata, and noise.

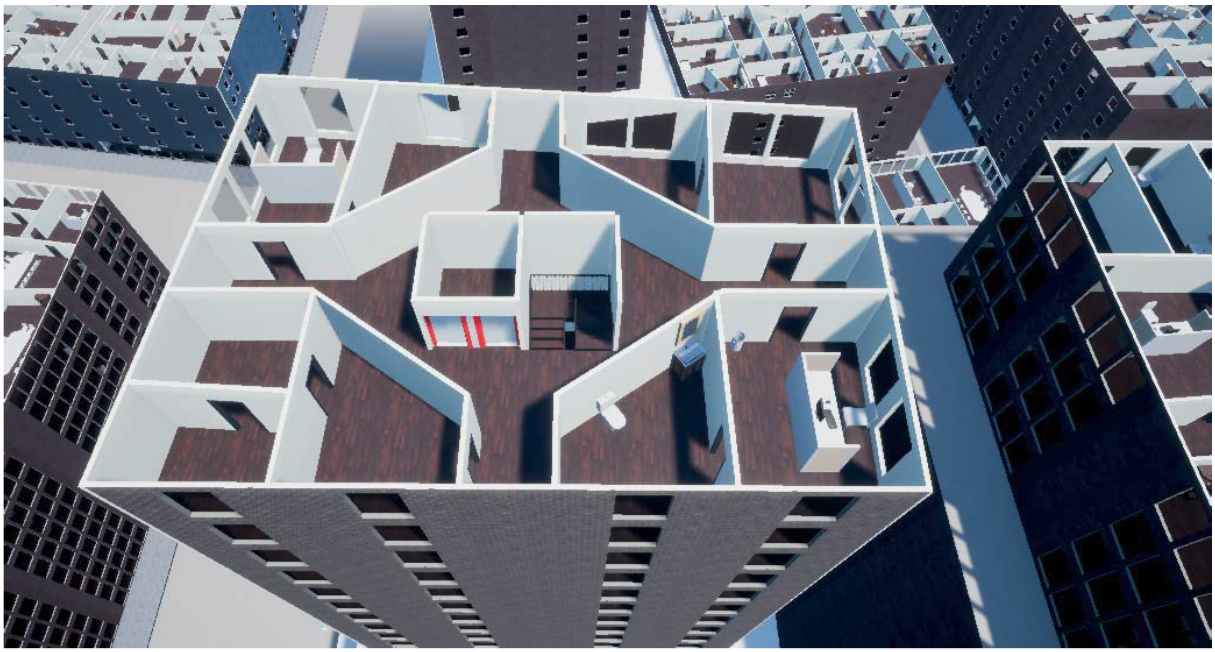


Figure 3-22. Top floor of an apartment building floor generated by Elinder [Elinder 2017]. Elinder's system generates a large city where all buildings use this basic floor layout.

3.2 Computer-Generated Architecture

This section describes research on the computer-generated architecture, *i.e.*, the procedural generation of building models. We first describe research using L-systems and shape grammars, which have proven very effective for generating exterior architecture. We then describe a variety of other methods that have been used to generate interior architecture. Unlike the level generators discussed in the previous section, which were mainly designed to produce levels with good gameplay, the systems in this section were mainly designed to produce realistic and convincing models of buildings.

3.2.1 L-Systems and Shape Grammars

Parish and Müller [Parish 2001] introduced a system for modelling large cities using L-systems. As part of city generation, three types of building models are generated: skyscrapers, commercial buildings, and residential houses. The building models are not highly detailed, but each building

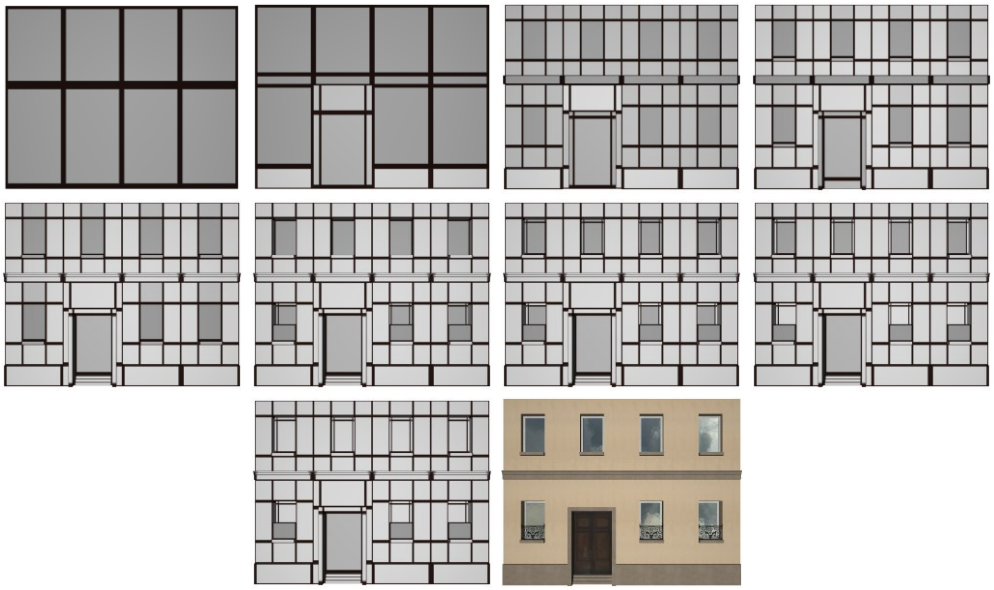


Figure 3-23. Subdivision of a building façade using a split grammar in the work of Wonka et al. [Wonka 2003].

Figure 3-24. Three building façades generated by Wonka et al. using the same grammar with different attributes [Wonka 2003].

is uniquely generated from the shape of its footprint. The authors' L-systems include symbols for geometric transformations such as scaling, translation, extrusion, branching, and termination, as well as to instantiate predefined objects such as roofs and antennae. Their method was later translated by Barrett [Barrett 2007] into a simpler algorithm using a priority queue instead of an L-system.

Elinder [Elinder 2017] built on Barret's algorithm to generate an entire city with both interior and exterior architecture. The generator uses a top-down approach, starting with a city layout and working down to individual buildings, floor layouts, and finally individual rooms.

The main accomplishment of this project was its massive scope. However, this is accomplished at the cost of realism and variety. Every building uses the same overall floor layout, with an elevator and stairs at the centre of every floor and four straight hallways around it. This divides each floor into four areas, each of which is then broken into apartments and rooms using a space partitioning algorithm.

Wonka et al. [Wonka 2003] developed a more sophisticated system for exterior architecture generation using a new type of shape grammar that they called a split grammar. Their split grammar operates on a set of shapes, where each shape is an object with both a symbol (a descriptive string such as "façade" or "roof") and geometric data (a bounding box and a local coordinate system). During the production process, multiple shapes may share the same symbol while storing different geometric data. A shape's symbol determines which rules may be applied to it. Each rule either splits a shape into multiple shapes that together fill its volume, or splits a shape into a smaller shape and some empty space. For example, a facade may be split into multiple storeys, which may then be split into windows, ledges, wall segments, and so on. The rule specifies the symbol for each new shape.

This split grammar is also an attributed grammar. Each shape has attributes which can be used as conditions for the application of a rule. Attribute values for a shape are determined by inheritance. Thus, an attribute set at a large shape early in the production process will be passed down as it is recursively split into smaller shapes. The attribute will ensure that all the descendant shapes develop in the same way, avoiding the undesirable case where different parts of the same building develop into different architectural styles. At the end of the production process, each shape is instantiated in the final building as a geometric primitive or a 3D mesh, based on its attributes.

This system was further developed by Müller et al. [Müller 2006]. They call their refined grammar a CGA shape grammar (presumably for Computer-Generated Architecture). This grammar supports the subdivision of shapes like the earlier split grammar, but also allows rules to specify more general geometric operations such as rotation, translation, and scaling, as well as extruding flat faces into 3D shapes and subdividing 3D shapes into their constituent faces and edges. The authors demonstrate how their system can generate buildings in multiple styles, including urban skyscrapers, Beverly Hills mansions, and Roman houses based on the ruins of Pompeii. Concave and non-rectangular buildings can be generated by overlaying multiple

Figure 3-25. Exterior building architecture generated by the CGA shape grammar of Müller et al. [Müller 2006]. Complex buildings are built by rotating and overlaying multiple shapes, while problematic architectural collisions (left) are resolved gracefully using snap lines and occlusion tests (right).

shapes and then applying similar rules to each one. This process can sometimes lead to undesirable collisions between architectural elements, as shown in Figure 3-25. The authors resolve these collisions using snap lines (which align storeys across multiple shapes) and occlusion tests (which prevent certain shapes, such as windows, from being placed if they are blocked by another shape).

The authors and their collaborators have since written multiple extension papers based on the CGA shape grammar. This subsequent research has generally not sought to change the underlying system, but rather to improve its user interface or apply it to new problems. Müller et al. [Müller 2007] used photographs of building façades to deduce shape grammar rules automatically. Lipp et al. [Lipp 2008] developed a visual interface to allow rules for the CGA shape grammar system to be specified and edited without manually writing rules as text. Dylla et al. [Dylla 2008] used the system to generate a virtual reconstruction of the city of Rome in A.D. 320, combining known ancient buildings such as the Circus Maximus with procedurally generated filler buildings. Meanwhile, several of the authors launched a private company to market their system commercially. It is now sold as the product *Esri CityEngine*.

Figure 3-26. *Rome Reborn 2.0*, featuring architecture generated by Dylla et al. [Dylla 2008].

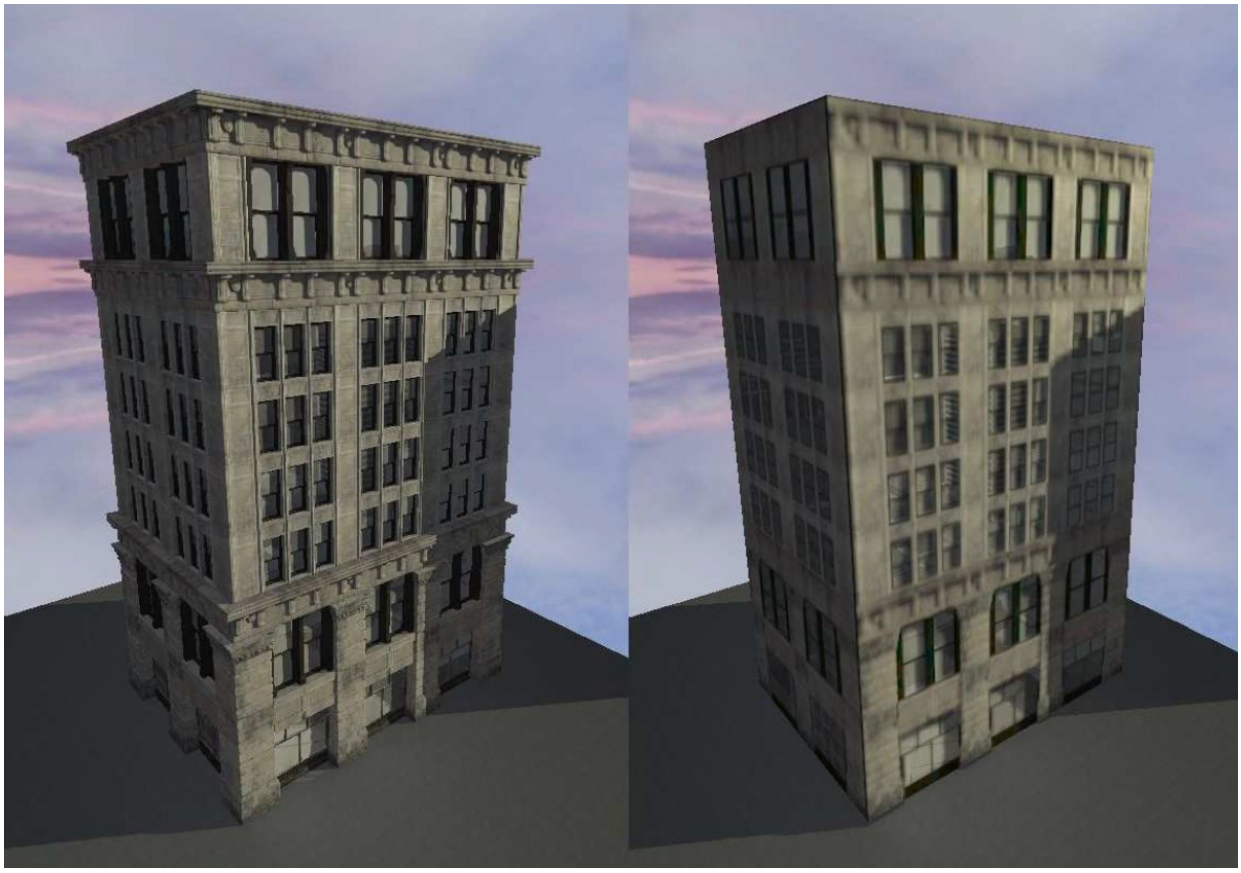


Figure 3-27. Detailed building model (left) and low-LOD mesh (right) generated using the Epic Games system described by Golding [Golding 2010].

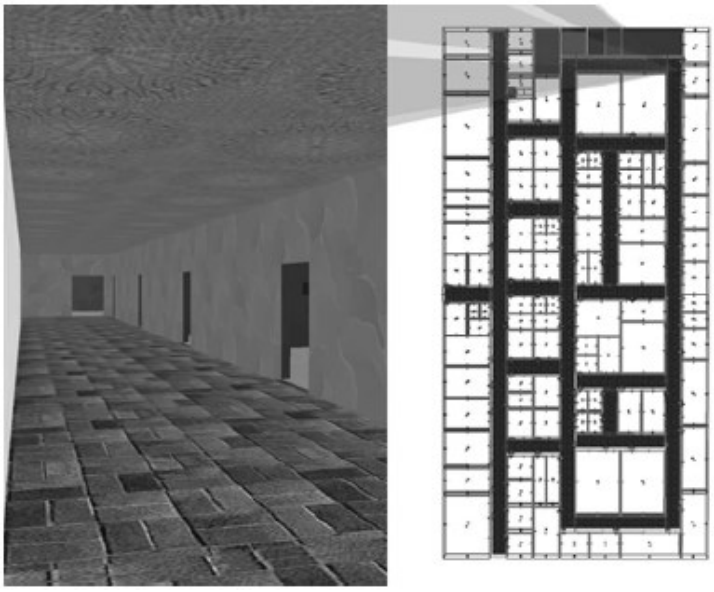


Figure 3-28. First-person and top-down views of floor layout generated by Hahn et al. [Hahn 2006].

Golding [Golding 2010] described a building generator that was developed at Epic Games, modelled on the work of Müller et al. In this system, a level designer specifies the overall shape of a building using simple solids. The details of the building are then generated using a shape grammar, which places modular meshes as its terminal shapes. An important concern for the tool designers at Epic Games was that their system should be usable by artists without a programming background. They therefore designed an interface for specifying a shape grammar as a tree using a visual interface. This allowed their artists to easily design a grammar while creating the meshes it would use. Another concern was that the system should be able to generate building models at multiple levels of detail (LOD) to improve performance when viewing buildings from far away. Their system automatically generates a low-LOD model of each building by baking the textures from the detailed building back onto the original low-poly solids used as the input to the grammar.

3.2.2 Interior Architecture Generation

Hahn et al. [Hahn 2006] developed a system to generate persistent interior architecture for games. Their system generates huge rectangular office buildings using a space partitioning method. A building is recursively divided into subregions, from the whole building down to floors, hallways, room clusters, and individual rooms. The subdivision rules are based on

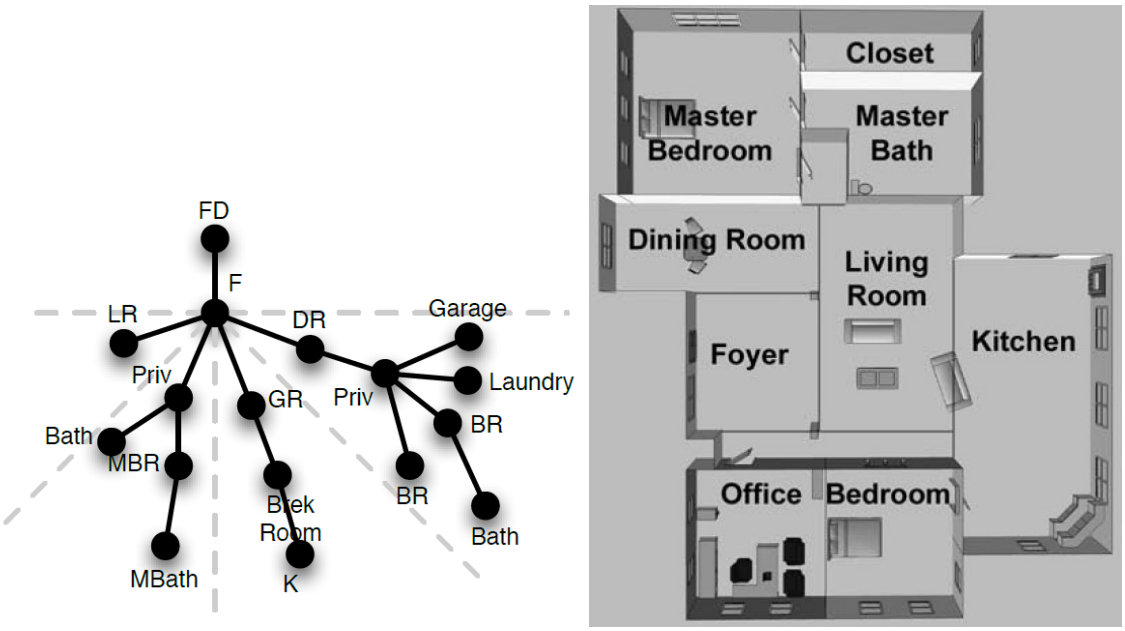


Figure 3-29. Left: A house floor plan laid out as a tree, generated by Martin's system [Martin 2006a]. Right: A spatial room layout generated from such a tree [Martin 2006b].

principles from the architecture treatise *A Pattern Language* by Alexander et al. [Alexander 1977]. The system is designed to allow generation in real time. To accomplish this, at each level of subdivision the only subregions generated are those that contain a room visible to the player. A persistent random seed is used so that sections of the building can be discarded when not needed and later re-generated. However, the generated rooms and hallways have very little architectural detail.

Martin [Martin 2006a; Martin 2006b] generated floor plans for residential houses using a graph grammar. His approach consists of several steps. First, general constraints are specified about the sizes, shapes, and relationships between rooms. These constraints are based on a study of many house floor plans and on the architectural treatise *A Pattern Language* by Alexander et al. [Alexander 1977]. A room graph is then generated using a graph grammar. The rules are designed around four steps: (1) adding public rooms, (2) specifying public rooms, (3) adding private rooms, and (4) adding stick-on rooms, such as closets and pantries. The graph is converted to a spatial layout using a push-out algorithm. Treating the front door as the root of the graph, each node is pushed away from its parent by a distance depending on the intended size of the room. Rooms are then converted to rectangles and expanded until they join together.

Figure 3-30. Floor plan for a two-storey house, generated by Merrell et al. [Merrell 2010].

Compared to the elegant graph generation system, this layout approach is unsatisfying. A major problem is that rooms do not conform to any planned footprint. An implication is that the generation of multi-storey buildings would be impossible using this system.

Merrell et al. [Merrell 2010] generated more realistic house layouts using a method inspired by close observation of the process used by human architects. The architect takes the high-level requirements provided by a client (such as the size of the house and the number of bedrooms) and develops them first into a bubble diagram (a graph of rooms), then an architectural program (a detailed list of rooms with their sizes and adjacencies), then a concept sketch, and at last into a detailed floor plan. At each stage of the process, the architect considers many options in a lengthy process of creative experimentation. Merrell et al. sought to replicate this process in two major stages. First, to generate an architectural program from a set of high-level requirements, they trained a network on a corpus of 120 real architectural programs. The

network learned the relationships between features such as a house's total square footage and the size of each room. Second, to generate a layout from an architectural program, they used a separate optimization system based on a function measuring room accessibility, area, aspect ratio, and shape. Their system can generate houses with multiple storeys and in multiple styles.

3.3 Summary of Novelty

Previous academic work on level generation has revealed many promising techniques for generating 2D game levels, especially side-view platformer levels and top-view dungeon layouts. However, there appears to have been very little work on generating true 3D game levels, and none on generating 3D levels with detailed interior architecture (as opposed to cave-like spaces).

The present work addresses this gap by applying three existing search methods to the new problem of generating 3D level layouts. Our randomized depth-first search algorithm is similar to the algorithm used by Koens [Koens 2015] to generate 2D platformer levels, but we introduce a partial reset counter which allows the algorithm to more effectively explore a large search space. Our evolutionary algorithm is similar to the algorithm used by Valtchanov and Brown [Valtchanov 2012], aside from some details of the level genotype which are adapted in order to support arbitrary 3D room designs. To our knowledge, we are the first to apply a Monte Carlo tree search to level generation.

We also address the problem of automatically placing 3D meshes from a modular mesh kit to create the details of a level's architecture. This topic is specific to the design of 3D levels with interior architecture, and as such has not been addressed in previous work.

In the field of computer-generated architecture, shape grammars have proven highly effective for generating 3D models of building exteriors, but they have not previously been applied with equal success to the generation of interior spaces. Other techniques have been developed to generate realistic interior layouts for modern office buildings and residential houses, but most game levels do not require realistic layouts of this type.

Unlike previous work, we show how a shape grammar can be used to generate interior building architecture. A novel feature of our shape grammar is that we use shapes to represent interior spaces (*i.e.*, negative space) rather than to directly represent physical objects such as walls and floors.

4 Methods

Our level generator consists of three major modules: room generation, level layout, and mesh placement. Our room generation module uses a shape grammar. Our mesh placement module uses tree extension guided by a search algorithm. Our mesh placement module uses simple case-based rules. In this chapter, we first describe the overall design of the generator and the roles of the three modules. We then describe each module in detail.

4.1 Overall Design

The overall design of our level generator is shown in Figure 4-1. Before generation begins, a room grammar and a set of mesh placement rules must be specified. The grammar and mesh placement rules together control the architectural style of rooms in the level. Additionally, the parameters of the level region and the fitness function must be set in configuration files. These parameters allow the designer to control the size, shape, and style of the level. Once these configuration steps are completed, the generator can be run.

Execution begins with the level layout module. The level layout module tests many possible arrangements of different rooms to find an arrangement that scores well on the fitness function. While generating level layouts, the level layout module repeatedly invokes the room generation module. Each time it is invoked, the room generation module generates a single room based on parameters provided by the level layout module. Each room is represented as a collection of abstract rectangular blocks called shapes. The level layout module uses these rooms to build its layouts. Once a level layout has been selected, execution advances to the mesh placement module. The mesh placement module converts rooms from their abstract shape representation to a list of meshes. Finally, the list of meshes is written to an output file, which can be loaded and parsed using a script in *Unreal Engine 4*.

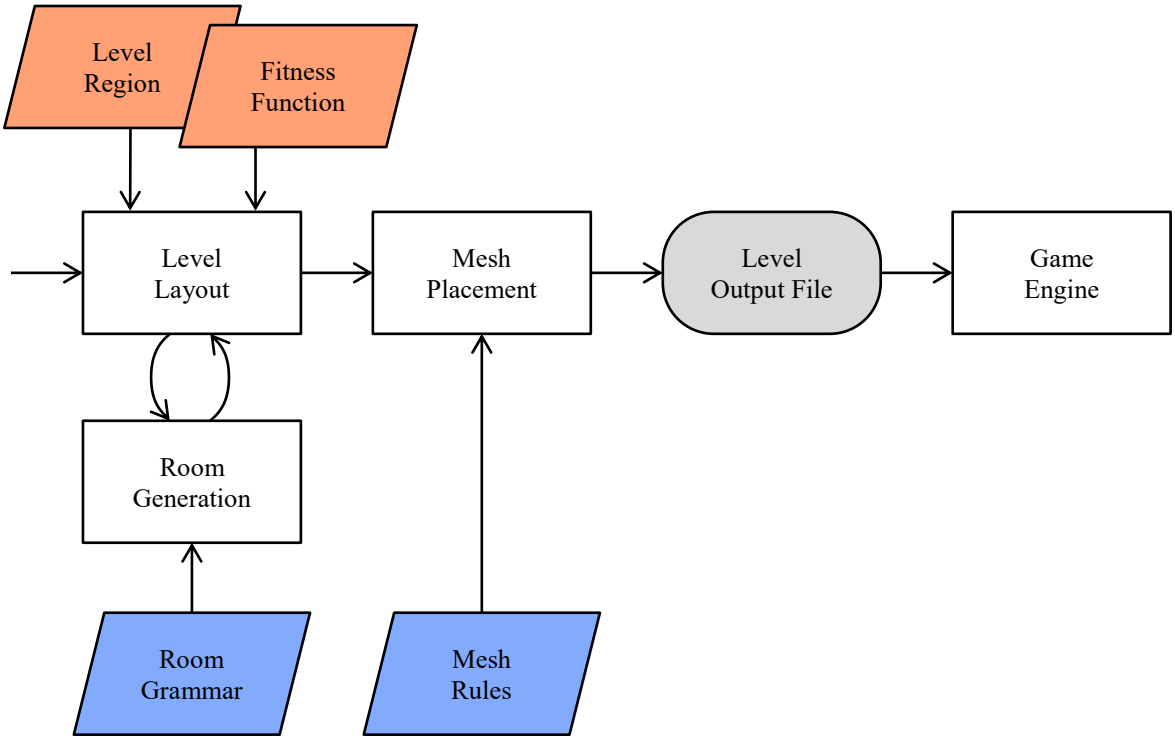


Figure 4-1. Overall design of our level generator.

The design of our level generator was inspired by Co's description of the human level design process, summarized in Section 2.1.3. The level layout module is designed to emulate the work of a human designer sketching many different designs on paper. The room generation module builds rooms with abstract blocks just as a level designer initially blocks out a level using simple geometric shapes. The mesh placement module serves the role of a quality pass, replacing the abstract geometry with more detailed architecture. Our current system does not perform the work of later quality passes such as adding decals, lights, and ambient sounds.

4.1.1 Building Architecture and Level Design

One question we were forced to consider was how closely our generated levels should resemble real buildings. The considerations that go into designing a real building are typically very different from those that go into designing a game level. Likewise, the previous methods that have been developed for architecture generation are very different from those that have been developed for level generation. Nonetheless, game levels are commonly designed to resemble

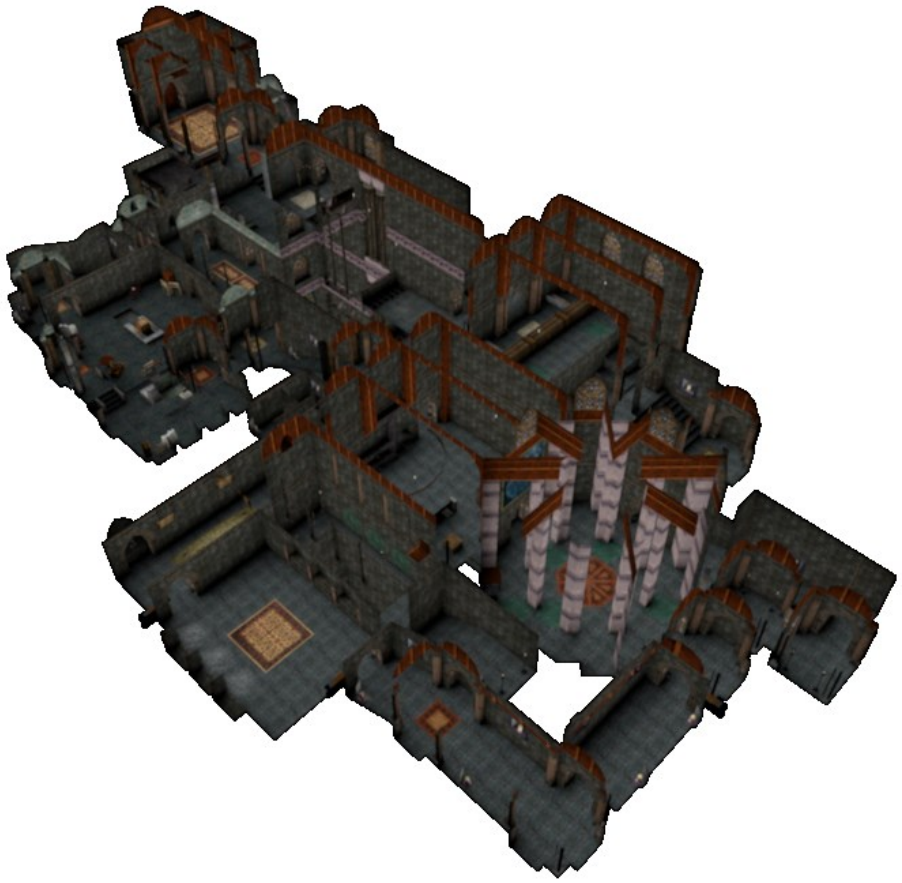


Figure 4-2. Cutaway view of the Skurge Challenge from *Harry Potter and the Chamber of Secrets* [Models].

real buildings, especially in high-budget games with detailed 3D environments. Realistic architecture can help make the game world feel more believable to the player.

To resolve this conundrum, we decided to generate levels that are architecturally realistic on a small scale but not on a large scale. The principle is that each individual room should resemble a real space, so that at any given moment the player can look around and see a superficially realistic world. However, the level as a whole need not be designed like a real building. We find that this principle often holds true for the levels in existing video games. For example, consider the design of the Skurge Challenge level from *Harry Potter and the Chamber of Secrets* (2002), one of the levels that Co [Co 2006] discusses in his book on level design. As can be seen in Figure 4-2, the individual rooms of the level are architecturally realistic, with pillars, archways, vaulted ceilings, and everything else one would expect to see inside a castle, but the overall layout of the level does not look like anything like a real castle building, or any

type of real building at all. This is not a problem during gameplay because the player never sees the entire level as a whole.

Another example can be drawn from the game *Dark Souls* (2011), a game often held up as an example of good level design [Boyd 2011; Caldwell 2017; Foerster 2017]. One level, the Undead Parish, has as its centrepiece a large church haunted by zombies and gargoyles. The church sanctuary itself contains very believable architecture in a medieval Romanesque style, but the connections between the church sanctuary and the surrounding areas of the level do not hold up to scrutiny. The front door of the church opens into a heavily fortified castle courtyard, while the side door leads over a bridge and down some stairs into a forest, and the other side of the church contains an elevator leading down to a completely separate temple. These connections make no practical sense. What use is it to fortify only one entrance to the church? Why put your elevator in a place that forces everyone to walk through the middle of the sanctuary to use it? But the layout supports good gameplay by requiring the player to find a way around the heavily fortified entrance to reach the side door and access the useful elevator.

The principle that a level only needs to be architecturally realistic on a small scale helped motivate our choice of methods. We used shape grammars in our room generation module because shape grammars are effective for generating realistic architecture. On the other hand, we used tree extension and search algorithms in our level layout module. These methods do not produce realistic architecture, but allow us to test many different possible layouts and find the ones with desirable gameplay properties.

4.1.2 Resources Used

We implemented our generator in C++. We used OpenGL for graphics, and the *OpenGL Mathematics* library [GLM] for linear algebra. We stored configuration data in JSON files, which we parsed using the *JSON for Modern C++* library by Niels Lohmann [Lohmann].

We used *Unreal Engine 4* to view finished levels. We wrote a simple script to parse the output file from our generator and place meshes using the engine API. Meshes were placed using the Instanced Static Mesh Component feature of the engine.

We used two different mesh kits to test our generator. The first was the *Multistory Dungeons* kit produced by Mana Station [Mana 2015]. This kit contains meshes for walls, columns, arches, and other architectural elements in a medieval gothic style. The meshes are

designed for modular use and fit together easily on a grid, so they were very convenient to use with our generator. The second kit we used was the *Soul: City* kit made by Epic Games [Epic 2014], the developers of Unreal Engine. This kit contains meshes for a cyberpunk city slum environment. The *Soul: City* meshes were originally designed for use in a particular level as part of a tech demonstration. We used the meshes from an interior hallway area of the level, which were designed in a modular manner.

4.1.3 Grid Measurements

Our generator builds rooms and levels on a three-dimensional grid. We use a grid because the meshes in modular mesh kits are normally designed to be laid out on a grid. Each grid cell represents the standard size of a mesh in the kit. During room generation and level layout, we treat grid cells as cubes with a size of one *grid unit* in each dimension. This allows the sizes of shapes (rectangular blocks) to be conveniently expressed as small integers. During mesh placement, it is necessary to convert grid coordinates to the distance units used by the game engine. *Unreal Engine 4* uses a distance unit that conventionally represents one centimetre, although its interpretation may vary depending on the game. We will refer to this unit as one *engine unit*.

The size of each grid cell in engine units can be configured as a parameter of the mesh placement module. The size should be set to correspond to the grid size of the meshes in the mesh kit. The grid size is not required to be the same in every dimension. For example, most of the meshes in the *Multistorey Dungeons* kit are multiples of 600 engine units long and multiples of 400 units tall, so when using this kit we set the size of each grid cell to (600, 600, 400). We use the convention that the z-axis points up.

The abstract shapes manipulated by the room generation and level layout modules are required to align to the grid. The sizes and positions of shapes in grid units must be integers, and shapes may only be rotated by increments of ninety degrees. The grid is also used during mesh placement. The positions of shapes in grid units are multiplied by the grid size to find their positions in engine units, and these positions are used to guide mesh placement. However, meshes in the final level are not required to align to the grid. The mesh placement module can place meshes with any transformation (translation, rotation, and scale) relative to a grid position.

We were initially cautious of the decision to use a grid, as we wished to permit the creation of unusual level features like diagonal hallways or hexagonal rooms. However, we found the benefits of using a grid to be significant. Rooms that do not align to a grid are difficult to fit into a level and difficult to connect to one another, making the problem of searching for a valid level layout vastly more difficult. Modular mesh kits are normally designed around a grid, so it is difficult to place meshes for architecture with non-standard sizes. Furthermore, using a grid makes it possible to iterate over all the grid cells in a room, which is a useful operation during mesh placement.

4.2 Room Generation

The room generation module generates a single room each time it is invoked. This is accomplished using a shape grammar. To distinguish it from other implementations of the shape grammar concept, we will refer to our shape grammar as a *room grammar*. Our room grammars are most strongly influenced by the CGA shape grammars of Müller et al. [Müller 2006], but are designed to describe the interior architecture of a room rather than the exterior architecture of a building.

Room grammars are set-based shape grammars. They may additionally be characterized as context-free grammars and attributed grammars. Each rule of a room grammar replaces a single shape, known as the *predecessor shape*, with one or more new shapes. The attributes and geometric properties of the new shapes are inherited from the predecessor shape, as modified by the rule. Room generation begins with a single shape known as the *axiom*. Rules are applied to the room until no more rules can be applied, at which point the room is finished.

The specification for a room grammar consists of a list of rules, a list of valid symbols, and a list of which symbols are valid axioms. The following subsections describe the features of room grammars in greater detail, and explain how room grammars are used to generate rooms in our room generation system.

4.2.1 Shapes

A room grammar operates on shapes, which are three-dimensional, rectangular blocks. Note that in a room grammar, a shape represents an interior space (such as a hallway), not a solid object

(such as a wall). Adjacent shapes represent adjoining interior spaces, whereas gaps between shapes represent non-traversable areas (such as walls). In technical terms, a shape is an object consisting of a symbol, geometric data, and optionally numeric attributes.

Symbols. A shape's *symbol* is a label indicating its type and purpose within the grammar. For example, a shape representing a hallway might have the symbol *Hallway*.

Geometric data. A shape's geometric data consists of a transformation matrix and a size vector. The transformation matrix is a 4-by-4 matrix which describes the local coordinate space of the shape. The size vector is a three-dimensional vector which specifies the extent of the shape along each axis in its local coordinate space. Together, the transformation and the size define an oriented bounding box. As noted in Section 4.1.3, we require shapes to be aligned to the grid. A shape's size must be a positive integer in each dimension, its translation must also be an integer in each dimension, and its rotation must be composed only of ninety-degree rotations.

Attributes. A shape may have zero or more attributes. Each attribute is a pair consisting of a name (a string) and a value (an integer). Attributes are determined by inheritance. That is, when a shape is created by a rule, it inherits all the attributes of the predecessor shape. Rules in a room grammar may also add new attributes to a shape or modify the values of inherited attributes.

Restrictions on symbols. A room grammar must specify a list of all the symbols that are valid within the grammar. For each symbol in the grammar, size restrictions may optionally be specified. These restrictions apply to all shapes of that symbol. A shape that violates the size restrictions for its symbol is invalid and cannot be added to a room. Size restrictions may include a minimum and maximum size in each dimension. Additionally, a required factor may be given in each dimension. Shapes of the symbol must have a size that is a multiple of the required factor. This can be useful if meshes are only available in specific sizes. For example, the x-size of a hallway could be restricted to multiples of two if the applicable wall mesh is two grid units long.

Axioms. The starting point for room generation is a single shape known as the axiom. The rules of a room grammar are deterministic, so a production process that begins with the same axiom will always produce the same room. Different rooms can be produced by starting with different axioms. For example, a larger axiom will normally produce a larger room.

A room grammar must specify a list of which symbols are valid axioms. For each of these *axiom symbols*, axiom size restrictions must also be specified in each dimension. These consist of a minimum axiom size, a size increment, and the maximum number of increments. A size is valid if it is equal to the minimum size plus a multiple of increments between zero and the maximum number, inclusive. The axiom size restrictions must be consistent with the normal size restrictions of the symbol in question, but may be more stringent if not all valid possible sizes are suitable for an axiom.

To generate a random room in a room grammar, an axiom symbol can be selected randomly, and then the size of the axiom in each dimension can be selected by starting with the minimum axiom size and adding a random number of increments.

4.2.2 Connectors

Connectors are a special type of shape that do not represent a part of room's structure. Rather, they represent possible connections to other rooms. Connectors from two different rooms can be joined to create a connection between the two rooms. Joining multiple rooms is part of the process of level layout, and as such it is discussed in more detail in Section 4.3. For now, we will concern ourselves with how connectors are defined and how they are placed during room generation.

A room may have one or more connectors in various places depending on its size, shape, and structure. For example, a simple rectangular room might have a row of connectors along each wall at ground level. These connectors would indicate that doors could be placed anywhere along the walls at ground level. Conversely, a room with a balcony might include connectors above ground level, but only such that they lead onto the balcony.

Properties of connectors. Connectors are considered a type of shape, and are generally treated similarly to other shapes, with several important differences. A connector cannot be used as the predecessor for a replacement rule. Connectors are considered to have no physical form and as such cannot collide with other shapes. It is possible for two connectors to overlap (indicating two possible, though mutually exclusive, places to put a door).

The rotation of a connector is meaningful. A connector's local x-axis indicates the direction it is facing. A connector must face outward, away from its own room towards empty space (where another room could be placed to connect to it). A connector's negative-x face must

be touching the face of a shape in its own room, while its positive-*x* face must not be touching any shape of its own room.

Connector symbols. Connectors use their own set of symbols, separate from those used for other shapes. Like the symbols for other shapes, the symbols for connectors must be listed in the grammar specification. For each connector symbol, an exact size must be specified. All connectors of the same symbol must be of the specified size. This allows connectors of the same type to be matched up and connected together.

Unlike normal shapes, a connector may have a size of zero in its local *x*-dimension (but not in any other dimension). A connector with an *x*-size of zero is called a *flat connector*, and indicates a place where two rooms could be placed directly touching. For example, a flat connector could be placed at the end of a hallway to indicate where the hallway should join directly with a room. A connector with an *x*-size greater than zero is a *deep connector*. A deep connector indicates that two rooms can be joined, but some other shape should be placed between them to join them. For example, an archway with a non-zero length could be used to join two rooms. The symbol for a deep connector must specify a *bridge symbol* as one of its properties. The bridge symbol is the symbol of the shape to be used to join two rooms.

Connector flags. It is important for the level layout module to know which connectors are compatible and may be joined together. The basic rule is that only connectors of the same size are compatible. Deep connectors must also share the same bridge symbol to be compatible. Additional constraints on joining connectors can be added using connector flags. A connector flag is simply a label that can be attached to a connector symbol. Each connector symbol may be marked with any number of connector flags. Additionally, for each connectors, any number of connector flags may be listed as required. A connector of symbol *A* is compatible with a connector of symbol *B* if and only if *A* has every flag required by *B* and *B* has every flag required by *A*.

Connector flags can be used to add semantic information to connector symbols in a room grammar. For example, flags could be used to indicate doors to different types of rooms (public rooms, private rooms, secret rooms, etc.), preventing them from being connected inappropriately. Note that it is possible to make a connector symbol that is not compatible with itself, if it does not possess every flag it requires. This feature can be used to prevent inappropriate connections

between rooms of the same type, such as the side door of one hallway opening into the side door of another hallway.

4.2.3 Rules

A grammar rule conventionally has a left-hand side, which describes when the rule may be applied, and a right-hand side, which describes the results of applying the rule. In a room grammar, the left-hand side of a rule contains a predecessor symbol, which must be some valid symbol of the grammar. The rule may only be applied to shapes of the predecessor symbol. The left-hand side of a rule may also contain one or more conditions. Each condition places a restriction on an attribute or on the size of the predecessor shape. The rule may not be applied to a shape that does not meet the conditions.

When a condition refers to an attribute, the condition must also specify whether it should be considered true or false in the event that the predecessor shape has no attribute of the specified name. For example, a condition might specify "*Aisles* < 2; true if undefined". This would mean that the rule could be applied either to a shape with an *Aisles* attribute less than two, or to a shape with no *Aisles* attribute. If a condition refers to the size of shape, the size is measured in the shape's local coordinate system.

The right-hand side of a rule in a room grammar consists of a series of operations. These operations may include setting attribute values, setting geometric data, and placing new shapes. A full list of rules operations is given in Appendix B. When a rule is applied, its operations are performed in order. Effectively, the operations form a short program, albeit one written using a very limited selection of instructions.

Because the rules of a room grammar may be somewhat complex, we present them in a list format rather than in traditional left-to-right format. Two examples of simple rules are given in Algorithm 4-1. The first rule, *ShrinkHallway*, removes a shape that has the symbol *TallHallway* and replaces it with a new shape that has the symbol *ShortHallway*. The new shape is one unit shorter than the old shape in the z-dimension (the vertical dimension). This reduction in size occurs on the *z*-positive side of the shape (the top is lowered) because the *Anchor* parameter of the *Reduce* rule indicates that the shape should be anchored in place on its negative side.

Algorithm 4-1. Two simple rules for a room grammar.

rule *ShrinkHallway*
predecessor symbol: *TallHallway*
conditions: none
operations:
Reduce (*Dimension*: *z*, *Anchor*: *Negative*, *Amount*: 1)
PlaceShape (*Symbol*: *ShortHallway*)

rule *DecomposePillarRoom*
predecessor symbol: *PillarRoom*
conditions: none
operations:
SingleSplit (*Axis*: *z*, *Anchor*: *Positive*, *Distance*: 1, *SymbolBefore*: *Pillars*, *SymbolAfter*: *Ceiling)*

The second rule, *DecomposePillarRoom*, splits a shape with the symbol *PillarRoom* into two separate shapes. The shape is split on the *z*-axis, i.e., into an upper section and a lower section. The split point is one unit away from the positive-*z* end of the shape (the top). The new shape before the split (the lower section) is assigned the symbol *Pillars*, while the new shape after the split (the upper section) is assigned the symbol *Ceiling*.

4.2.4 Room Generation Algorithm

The algorithm to generate a room using a room grammar is given in Algorithm 4-2. The outer loop of the algorithm iterates over every shape in the room, in the order they were added to the room. The inner loop iterates over every rule in the grammar until it finds one that can be successfully applied to the current shape. This rule is then applied, replacing the shape with one or more new shapes. If no rule can be applied to a shape, it is passed over. A shape that has been passed over will appear unchanged in the finished room.

Room generation begins with a single shape (the axiom), so the outer loop of the algorithm may give the false impression that it will only execute a single time. This is normally not the case, because the inner loop adds one or more new shapes to the back of the list each time a rule is successfully applied. These new shapes will subsequently be processed by the outer loop.

The actual process of applying a rule to a shape involves three main steps. First, a working state is established. Second, the operations of the rule are executed in order. These operations manipulate the working state and add shapes to the output. Third, the output is validated, and applied to the room if valid. Each of these steps requires some explanation.

Working state. The working state consists of a projected shape and a stack. The projected shape is a temporary shape used as a draft to build up the properties of shapes before adding them to the output. It is initialized as an exact copy of the predecessor shape. The stack is initially empty, but can be used to save and restore the state of the projected shape. For example, a series of transformations can be applied and used to place a single shape, then reversed by restoring the state from the stack. After that, different transformations can be applied and used to place another shape.

Executing rule operations. In the second step, all the operations of the rule are applied in order. The output of this step is a list of shapes to add to the room. Each operation executed either changes the working state or adds to the output. A detailed list of all the rule operations supported by our room grammar is given in Appendix A. Here we summarize the most significant operations.

One important type of operation is to resize the projected shape. The size in a given dimension can be set to a new absolute size, or changed relative to the current size. Typically the size is reduced, in order to subdivide space. When resizing the projected shape, an anchor point must be specified. Either one face may be anchored so that space is added or removed from the opposite face, or the shape may be anchored in the centre, so that an equal amount of space is added or removed on each side. The new size for the projected shape, or the amount by which to change the size, can be specified by a constant number or by an attribute.

Similar operations exist to change the attributes of the projected shape. An attribute can be set to a new values, changed by a relative amount, or set using a provided function. Other operations exist to push the state of the projected shape to the stack, to restore the state of the projected shape from the top of the stack, and to pop the top state off the stack.

The simplest output operation adds a copy of the projected shape to the output. Other output operations split the projected shape into two or more sections along a specified axis, and add each section to the output as a separate shape. The size of each section can be specified in absolute or proportionate terms, and can be specified either by a constant number or by an attribute. A different output operation can be used to add connectors to the output. Unlike other shapes, which are placed by copying or subdividing the projected shape, connectors are added such that they are touching an outside face of the projected shape.

Algorithm 4-2. Room Generation

```
function GenerateRoom (shape Axiom, grammar Grammar):
    Room ← empty list of shapes
    add Axiom to back of Room
    for each shape Shape in Room:
        for each rule Rule in Grammar:
            if Shape.Symbol ≠ Rule.PredecessorSymbol:
                skip to next loop iteration
            if Rule.ConditionsAreSatisfied(Shape) is false:
                skip to next loop iteration
            Results ← GenerateResults(Rule, Shape)
            if ResultsAreValid(Room, Shape, Results)
                delete Shape from Room
                add Results to back of Room
                break from loop
    return Room
function GenerateResults (rule Rule, shape Predecessor):
    Stack ← empty stack of shapes
    ProjectedShape ← Predecessor
    Results ← empty list of shapes
    for each operation Operation in Rule:
        Operation.Apply(Stack, Predecessor, Results)
    return Results
function ResultsAreValid (list of shapes Room, shape Predecessor, list of shapes Results):
    for each NewShape in Results:
        if NewShape is not a valid size for its symbol:
            return False
        for each shape OldShape in Room:
            if OldShape = Predecessor:
                skip to next loop iteration
            if OldShape collides with NewShape:
                return False
    return True
```

Output validation. Once all the operations of the rule have been applied, the output must be validated. The output is invalid if any shape violates the size restrictions for its symbol, or if the bounding box of a new shape collides with any existing shape in the room (other than the predecessor shape, which will be removed when the new shapes are added). If the output is invalid, the rule cannot be applied; the output is discarded and the room generation algorithm goes on to test other rules. If the output is valid, the predecessor shape is deleted and the shapes in the output are added to the room.

Figure 4-3. Cloister of the Cathedral of Monreale, Sicily, Italy [Britannica].

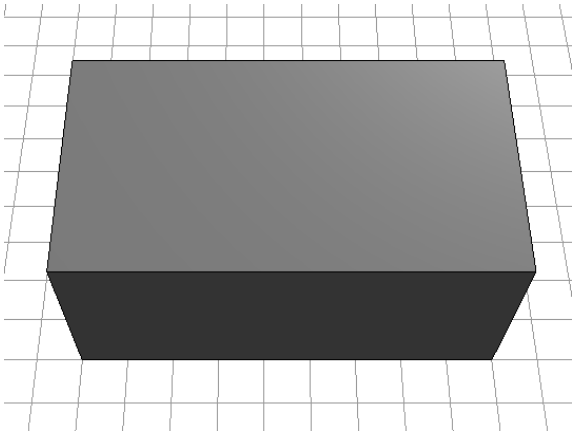
4.2.5 Examples of Room Generation

This section shows two examples of room generation using our method. Complete details of the grammar used in these examples are given in Appendix B.

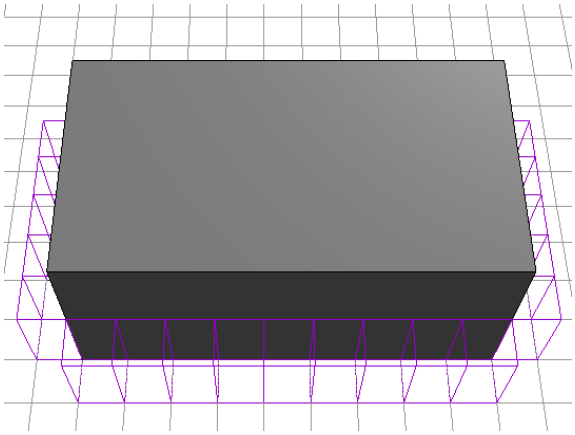
In our first example, we generate a cloister. A cloister is a rectangular courtyard enclosed by a covered walkway. Cloisters are commonly found in medieval monasteries and cathedrals. A real-world example is shown in Figure 4-3.

We used five rules to generate a cloister in our room grammar, as shown in Figure 4-4. The rules divide the axiom shape into the courtyard and the surrounding walkway, then add a plus-shaped path to the courtyard and a groin vault ceiling to each walkway. A groin vault is an x-shaped ceiling vault formed by the perpendicular intersection of two arches. In our room grammar, a long shape with the *GroinVault* symbol represents a row of such vaults.

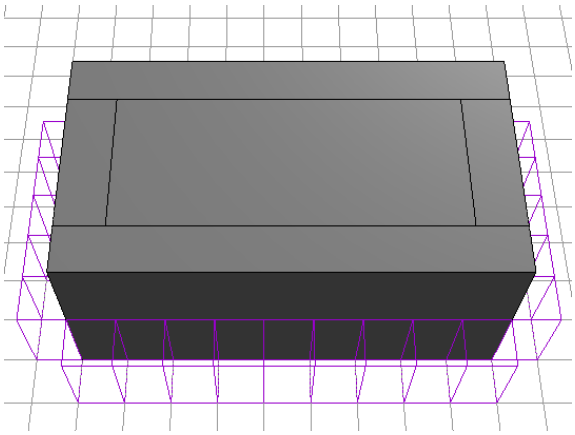
In our second example, we generate the main room of a cathedral. Cathedrals are traditionally built with a cross-shaped layout. The main east-west hall of the cathedral is intersected by a north-south hall known as the transept. The transept separates the short east end



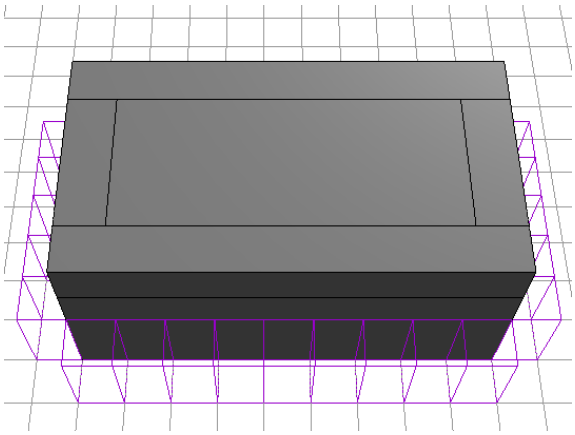
(0) Axiom shape



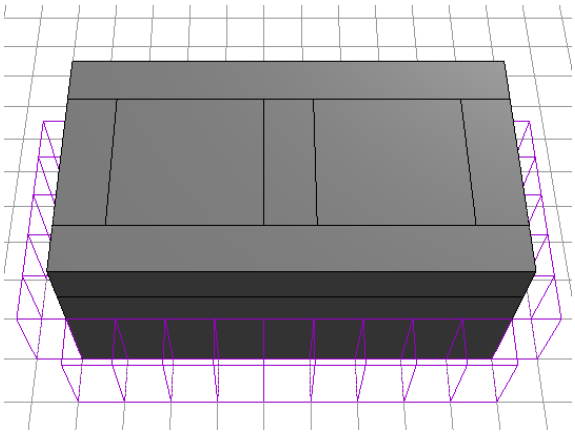
(1) Connectors (purple) are added at ground level



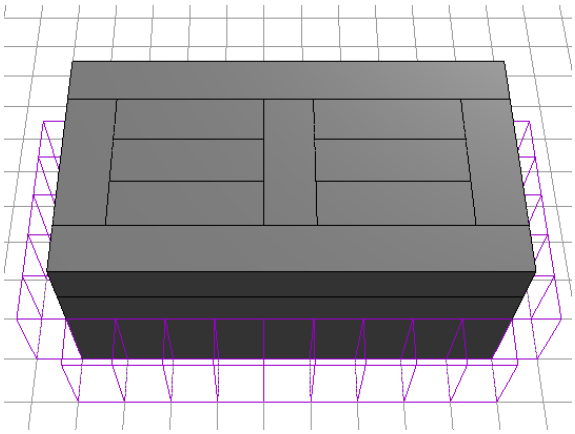
(2) The cloister is subdivided into a courtyard and four walkways.



(3) Each walkway is subdivided into a groin vault ceiling (top) and columns (bottom).

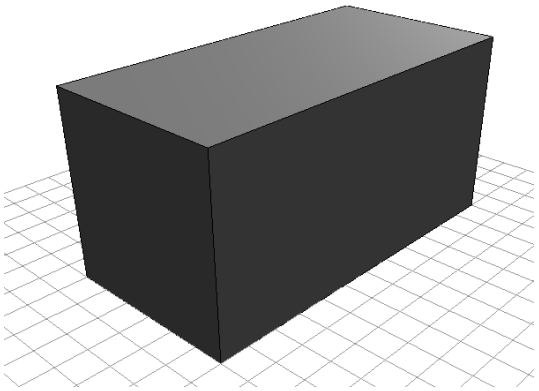


(4) The courtyard is subdivided into a path and two half-courtyards.

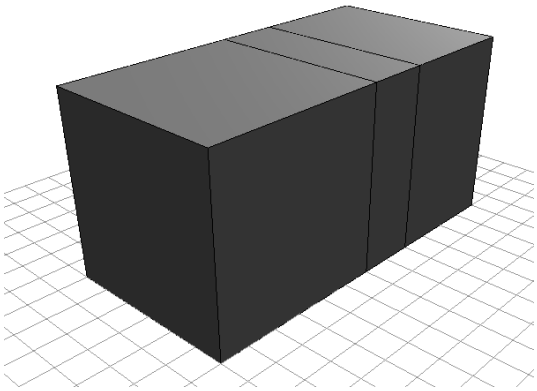


(5) Each half-courtyard is subdivided into a path and two quarter-courtyards.

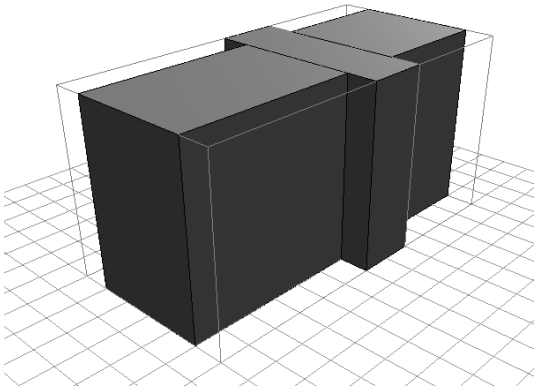
Figure 4-4. Generation of a cloister using five rules.



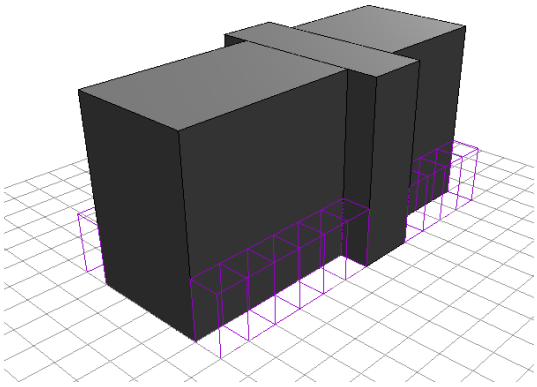
(0) Axiom shape



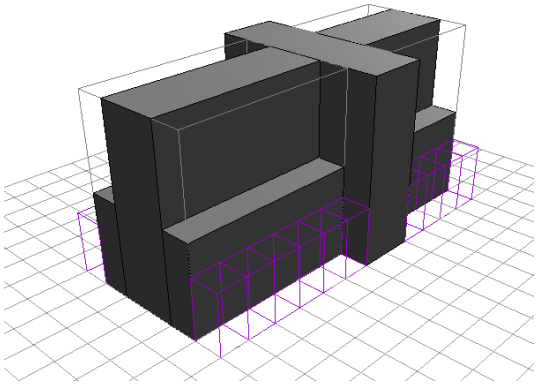
(1) The cathedral is subdivided into the nave (west part), transept (centre), and sanctuary (east part).



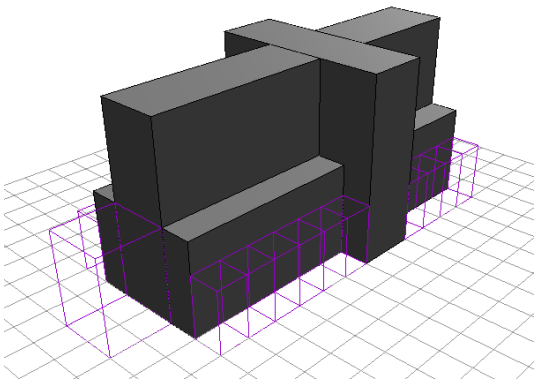
(2) The nave and sanctuary are each subdivided into a narrower shape and some empty space.



(3,4) Small connectors are added to the sides of the nave and sanctuary.

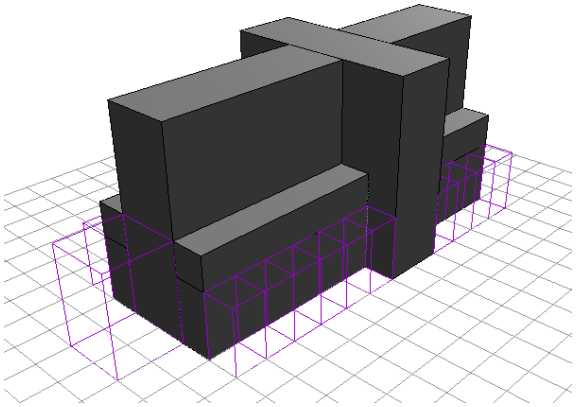


(5) The nave and sanctuary are each subdivided into a tall narrow centre and lower aisles on the sides.

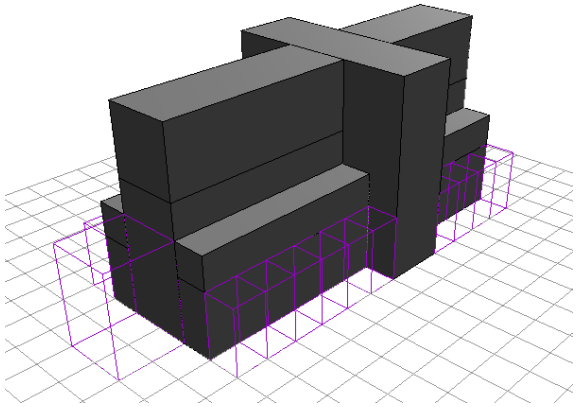


(6) A large connector for the main doorway is added at the west end of the nave.

Figure 4-5. Generation of a cathedral room using eight rules (continued on next page).



(7) Each aisle is split into a vaulted ceiling on top and columns below.



(8) The aisle and sanctuary are each split into a large vaulted ceiling on top and large columns below.

Figure 4-5 (cont'd). Generation of a cathedral using eight rules.

of the cathedral, known as the sanctuary, from the long west end, known as the nave. The east-west hall is often flanked by aisles with a lower ceiling, separated from the main hall by rows of arches or columns.

In our room grammar, we generate a cathedral using eight rules, as shown in Figure 4-5. The axiom is first subdivided into the nave, the transept, and the sanctuary. The width of the nave and the sanctuary are reduced to create the cross shape of the cathedral. The nave and sanctuary are further subdivided to create aisles. Depending on the amount of available space, grammar creates zero, one, or two aisles on each side of the main hall. In the example shown, one aisle is added on each side. The top of each shape is finally converted into a row of groin vaults.

4.3 Level Layout

The purpose of the level layout module is to arrange multiple rooms into a well-designed level. Our level layout module uses a tree extension method guided by a search algorithm. As described in Section 3.1.2, the principle of tree extension is to start with a single room and repeatedly add an additional room by connecting it to an existing room. During tree extension, there are normally many different mays that another room could be added to a given level. In our system, a search algorithm is used to choose the option that will maximize a level fitness

score. We tested three different search algorithms for this purpose: a randomized depth-first search, a Monte Carlo tree search, and an evolutionary algorithm. Our system uses a variable fitness function that can be configured to fit the goals of the designer. The designer can also configure the level region, which is the volume of space available for the level to be built in. This section describes each of these features of the level layout module in detail.

4.3.1 The Level Region

The level region is configured by providing one or more axis-aligned bounding boxes, a start position, and one or more checkpoints. These values are set in a JSON file loaded by the generator.

Region volume. The region volume is the union of the provided bounding boxes. The level layout system is only permitted to place rooms within this region volume.

Start position. The start position is used to place the first room of the level. In our system, the first room is always a dummy room consisting of a single connector and no other shapes. The only purpose served by the start room is to provide a place for the second room—the first real room—to be attached.

Checkpoints. Checkpoints indicate positions that the level should go through. These can be used to control the shape of the level within the region volume. The fitness function can be set to reward levels based on how closely they follow the provided checkpoints. The last checkpoint listed in the level region is considered the *endpoint*, which is used in some features of the fitness function. Aside from affecting a level's fitness score, checkpoints have no effect on level layout.

It is beneficial to use multiple checkpoints when generating level layouts in complex level volumes. If the region volume contains corners (such as if the volume is C-shaped or S-shaped), it may be difficult for the search algorithm to find a route around the corners, especially if it is necessary to build a path that first leads away from the endpoint before leading back towards it. Placing a checkpoint at each corner rewards the generator for building towards the corners.

4.3.2 Tree Extension

Tree extension is in an iterative process. Each iteration consists of two steps. First, a new room is generated by invoking the room generation module. Then, the new room is connected to an existing room using one of its connectors.

Connecting rooms. We can connect a new room to a level by placing it such that one of its connectors aligns with a compatible connector of an existing room, which we call the *parent room*. Two connectors are aligned if they occupy the same space, but face in opposite directions (each pointing into the other room). Note that the level layout module treats a room as a single unit, with all its shapes moving and rotating together in tandem. Therefore, to align the connector of a new room with the connector of its parent room, we rotate the entire new room until its connector is oriented correctly, and then move the entire new room until its connector is positioned correctly. Once two connectors are aligned, the connectors can then be marked as joined, forming a connection. If the connectors are deep connectors, a shape with the bridge symbol of the connectors is added in the same space as the connectors.

When a new room is generated, there are usually numerous possible ways it could be connected to the existing level. Theoretically, each connector of the new room could be connected with each unused compatible connector of every existing room in the level, so the maximum number of possible connections is given by the number of connectors on the new room times the number of rooms in the level times the number of unused compatible connectors per room. Each of these possible combinations implies a specific placement (position and rotation) for the new room. However, some of these placements may be invalid. A placement is invalid if it results in a collision between any shape of the new room and any shape of an existing room. A placement is also invalid if any shape of the new room is adjacent (touching faces) with a shape from another room, unless there is a flat connector between them. The purpose of flat connectors is to control where rooms are allowed to join, so rooms that join without connectors are not allowed. Finally, a placement is invalid if it would cause any shape of the new room to protrude outside the region volume.

Incidental connections. When a room is placed in a valid position, one of its connectors always connects to its parent room. However, it may be that one or more other connectors of the room are also incidentally aligned with compatible connectors of other rooms in the level. We

Table 4-1. Features supported by our fitness function.

Feature	Parameter	Description
VolumeFill	*MaxPercent*	The percent of the region volume that is filled with shapes.
RoomCount	*MaxCount*	The number of rooms in the level.
AverageConnections	*MaxCount*	The average number of connections per room.
DeadEndRooms	*MaxPercent*	The percent of rooms that have only a single connection.
LinearRooms	*MaxPercent*	The percent of rooms that have exactly two connections.
JunctionRooms	*MaxPercent*	The percent of rooms that have three or more connections.
CheckpointProximity	none	Reward based on the proximity of each checkpoint to the centre of the nearest room.
LevelLength	*LengthFactor*	Reward based on the length of the shortest path from the start room to the end room.
RoomsOnAnyPath	*MaxPercent*	The percent of rooms that lie on any path from the start room to the endpoint.

can call these unplanned connections *incidental connections*. In our level layout system, we rely upon incidental connections to create levels with non-tree layouts.

When an incidental connection arises, we can choose whether we wish to accept the incidental connection (joining the two connectors) or reject it (leaving the connectors disconnected despite their compatibility). We use the following rules to decide whether to accept incidental connections. If the connectors are flat connectors, we have no choice but to accept the connection—the two rooms have effectively already joined by virtue of their positions. If the connectors are not flat, we reject the incidental connection if there is already a connection between the two rooms. Otherwise, we accept it. If multiple incidental connections arise between the same pair of rooms simultaneously, we accept one and reject the others.

4.3.3 The Fitness Function

The design of the fitness function has a profound impact on a search-based algorithm. The better the search algorithm, the more closely its output will correspond to the fitness function. Conversely, we may also say that the better the fitness function, the more effectively it will guide the search algorithm towards good content.

To increase the flexibility of our generator, we designed it to support a variable fitness function. The fitness function is made up of a series of possible features, which are listed in Table 4-1. Each feature describes some characteristic that could be desirable in a level. Each feature can be used to evaluate a level, giving a score in the range from zero to one.

Like the level region, the fitness function is configured by setting values in a JSON file. In the configuration file, a weight can be set for each feature to determine its impact on the level's score. Setting a feature's weight to zero disables it, so it has no impact. The overall fitness function is the weighted average of the scores for all features. It is not intended that all features of the fitness function be used simultaneously. Different features are intended to reward different level design goals, some of which are contradictory.

In addition to their weights, most features of the fitness function have a parameter that can be set in the configuration file. These parameters affect how the features are calculated, generally by specifying a goal that must be met to achieve the maximum reward. For example, when the *VolumeFill* feature is evaluated, the maximum reward (1.0) is given if at least *MaxPercent* of the region volume is filled. When the *RoomCount* feature is evaluated, the maximum reward is given if the level has at least *MaxCount* rooms. Rewards fall of linearly below the maximum.

Most of the features listed in Table 4-1 are self-explanatory and simple to calculate. However, the last three features—*CheckpointProximity*, *LevelLength*, *RoomsOnAnyPath*—require some explanation.

CheckpointProximity. For each checkpoint (see section 4.3.1), a score from zero to one is calculated based on the distance to the centre of the nearest room. The maximum score is achieved when the checkpoint is exactly in the centre of a room. The minimum score is achieved when no room is closer to the checkpoint than the start room is. The score falls off linearly for distances between these extremes. The reward for the *CheckpointProximity* feature is the average of the scores for all checkpoints. This implies that the impact of each checkpoint on the overall fitness function is reduced when there are many checkpoints, so it may be desirable to increase the weight of *CheckpointProximity* when increasing the number of checkpoints in the region.

LevelLength. This feature defines a target length for the shortest path through the level. The target length is *LengthFactor* multiplied by the Euclidean distance from the start room to the end room, which the room closest to the endpoint (see section 4.3.1). Thus, a *LengthFactor* of 1.0 indicates that the level should be a perfectly straight line from start to finish, while a higher number indicates that the level should twist and turn on the way to its destination.

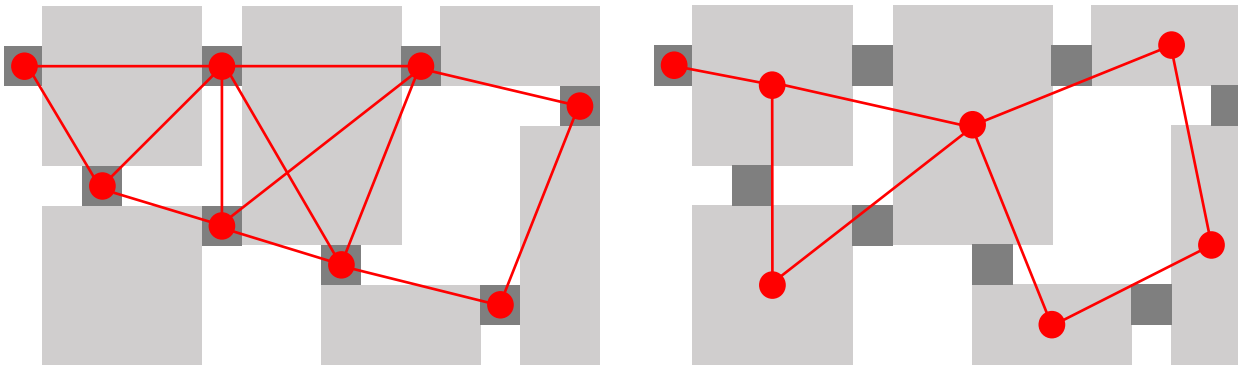


Figure 4-6. Two methods of converting a level to a graph. In the method on the left, each connection is converted to a node. In the method on the right, each room is converted to a node.

To calculate the length of the shortest path through the level, we convert the level to a weighted graph. Each connection between two rooms becomes a node in the graph, as does the start connector. An edge exists between each pair of connectors that connect to the same room, as shown in Figure 4-6 (left). The weight of each edge is given by the Euclidean distance between the two connectors it joins. Dijkstra's algorithm is then used to find the shortest path from the sole connector of the start room to any connector on the end room.

The maximum reward is given when the length of the shortest path is exactly equal to the target length. If the projected length is less than the target length, the reward falls off linearly to zero at an actual length of zero. If the projected length is greater than the target length, the reward falls off linearly to zero at an actual length of twice the target.

Note that the *LevelLegnth* feature is calculated solely based on the length of the path from the start room to the end room, regardless of how close the end room is to the endpoint specified in the level region. To ensure that the end room is actually close to the endpoint, the *CheckpointProximity* feature should be used.

***RoomsOnAnyPath*.** This feature measures the percent of rooms that lie on any path from the start room to the end room. A path may not visit the same room more than once. In other words, this feature tests what percent of rooms are not on a dead-end branch of the level.

To determine which rooms in the level lie on a path from the start room to the end room, we convert the level to a graph where the rooms are the nodes and the connections between rooms are the edges, as shown in Figure 4-6 (right). We then use a graph search algorithm to

Algorithm 4-3. Number of Rooms on Any Path

```
function RoomsOnAnyPath (level Level, room Start, room End):
    for each room Room in Level:
        State[Room] ← UNVISITED
    Stack ← empty stack
    StartSearch(Level, Start, End, none, Stack)
    while Stack is not empty:
        BlockRoom ← pop from Stack
        if State[BlockRoom] = GOOD:
            for each room Room connected to BlockRoom:
                if State[Room] = UNVISITED:
                    StartSearch(Level, Room, End, BlockRoom, Stack)
```

```
function StartSearch (level Level, room Start, room End, room Blocked, stack Stack)
    State[Start] ← VISITED
    for each room Room connected to StartRoom:
        if State[Room] = UNVISITED:
            PathSearch(Room, End, Blocked, SearchStack)
            if State[Room] = GOOD:
                State[Start] ← GOOD
                push Start to Stack
                break from loop
    for each room Room in Level:
        if State[Room] = VISITED:
            if State[Start] = GOOD:
                State[Room] = UNVISITED
            else
                State[Room] = BAD
```

```
function PathSearch (room Here, room End, room Blocked, stack Stack)
    if Here = EndRoom:
        State[Here] ← GOOD
        return from function
    else
        State[Here] ← VISITED
    for each room Room connected to Here:
        if Room = Blocked:
            skip to next loop iteration
        if State[Room] = UNVISITED:
            PathSearch(Room, End, Blocked)
        if State[Room] = GOOD:
            State[Here] ← GOOD
            push StartRoom to Stack
            return from function
```

examine the nodes of the graph. The full details of this graph search algorithm are given as Algorithm 4-3; a simple explanation follows.

First we mark every node as "unvisited." Then we perform a depth-first search of the graph from the start room. While searching, we keep track of which nodes we have visited and

do not allow the search to explore the same node more than once. When the end room is found, we backtrack along the path to the start, marking each node as good (where a "good" node means a node on a path from the start room to the end room).

Having found one path, we need to check each side path to see if it is a dead end or another good path. We try "blocking" each good node we have found, one at a time, starting with those closest to the start room and working outwards. When we block a node, we do not allow the search to visit it, but we perform a new depth-first search from each of its neighbours (except those already marked as good or bad) in order to find other possible paths. Each search is prevented from visiting the current blocked node, and prevented from visiting any node more than once. If the search finds the end room or any other good node, we know that we have found a new path: that is, the path that leads from the start room to the blocked node, along the newly explored path to another good node, and thence along the known path from the good node to the end room. Thus we mark each node on the newly explored path as good. We then try recursively blocking each of the nodes on this new good path to search for further side passages.

If a search terminates without finding any good node (aside from the blocked node), we know that no path to the end room exists in the explored area of the graph, so we can mark all nodes visited during the search as bad. Once we have tried blocking all good nodes, we can count the number of good nodes to see what percent of the rooms in the level lie on any path from the start room to the end room.

4.3.4 The Room Placement Machine

To apply a search algorithm to level layout, we need to identify all the decision points in the tree extension process, and all the possible options at each decision point. Then we can use a search algorithm to test different options and find the best ones.

There are seven decision points in each iteration of the tree extension algorithm. The first four decisions relate to the axiom shape which we provide to the room generation module. The other three decisions involve how to connect the new room to the existing level. These seven decisions are:

1. What is the symbol of the axiom?
2. How large is the axiom in the *x* dimension?
3. How large is the axiom in the *y* dimension?

4. How large is the axiom in the *z* dimension?
5. Which existing room is the parent room?
6. Which connector of the parent room will the new room be connected to?
7. Which connector of the new room will be connected to the parent room?

These are the seven questions we must answer for each room we add to the level.

To provide a structured way of dealing with these decisions, we introduce an interface called the *room placement machine*. The room placement machine can be pictured as a box that asks the seven questions in order and takes in the answers as input. When the fourth question is answered, the machine invokes the room generation module to generate a room. When the seventh questions is answered, the machine connects the room to the level and returns to the first question.

In our level layout module, the room placement machine is an object that controls the addition of rooms to a level. The interface of the room placement machine provides five functions, listed in Table 4-2. Using these functions, it is possible to write many possible search algorithms. For example, Algorithm 4-4 shows how a simple random search can be implemented using the interface of the room placement machine. This algorithm adds rooms by selecting random answers at each decision point until it reaches a point where no valid answer is possible. After each decision point, the algorithm checks the fitness score of the current level, and saves it if it is the best level found so far.

Our three main search algorithms interact with the room placement machine in a similar manner. At all times, we keep a record of the best level found so far; this is the level we will output when the level layout stage is complete. Normally, we call the chosen search algorithm repeatedly until a satisfactory level is obtained.

Table 4-2. Interface of the room placement machine.

Function (Parameters)	Description
InitLevel (*InitialConfiguration)*	Resets the level to its initial configuration, and resets the state of the machine to the first decision point.
GetLevel ( )	Returns the current level.
GetValidInputSet ( )	Returns the set of valid options at the current decision point.
ProcessInput (*Input*)	Reads in the decision at the current decision point, and advances to the next decision point.
Backtrack (*StepCount*)	Deletes the last *StepCount* input values read, removing any rooms placed by those input values from the level and returning to the appropriate decision point.

Algorithm 4-4. Random Search

```
function RandomSearch (room placement machine RPM, function FitnessFunction,
        configuration InitialConfiguration):
    RPM.InitializeNewLevel(InitialConfiguration)
    BestLevel ← RPM.GetLevel( )
    ValidInputSet ← RPM.GetValidInputSet( )
    while ValidInputSet is not empty:
        NextInput ← random member of ValidInputSet
        RPM.ProcessInput(NextInput)
        ValidInputSet ← RPM.GetValidInputSet( )
        if FitnessFunction(RPM.GetLevel( )) > FitnessFunction(BestLevel)
            BestLevel ← RPM.GetLevel( )
    return BestLevel
```

Valid input values. The most complex function of the room placement machine is *GetValidInputSet*, which must generate all valid answers to the question at the current decision point. Our method for finding the set of valid input values for each question is as follows. For question 1, the valid options are given by the grammar's list of valid axiom symbols. For question 2–4, all the sizes permitted by the size restrictions of the chosen axiom symbol are valid. For question 5, all rooms with at least one unconnected connector are valid. For question 6, all unconnected connectors of the chosen parent room are valid. For question 7, all connectors of the new room that result in a valid room placement are valid. To determine this, it is necessary to try aligning each compatible connector of the new room to the chosen connector of the parent room and test whether the resulting placement is valid.

For simplicity, the valid answers for every question are expressed as integers. To allow this, symbols, rooms, and connectors are numbered based on the order they are created.

Note that for the last three questions, the set of valid answers depends on the initial level configuration and the sequence of all previous inputs to the room placement machine. In many cases, no valid answers exist. For example, if the chosen connector of the parent room points directly at the boundary of the level region, none of the connectors of the new room will allow for valid room placement. In such a case, no further input will be accepted by the room placement machine. The *Backtrack* or *ResetLevel* function must be used before any further rooms can be added to the level.

Valid layout tree. We can describe the space of all valid input sequences to the room placement machine as a tree. The root of the tree represents the initial level configuration. Every other node of the tree stores an integer. Each node *N* corresponds to the state produced by

traversing the tree from the root down to *N* and feeding the room placement machine with the integer at each visited node. The children of *N* contain the valid answers at the next decision point given the current state. Travelling deeper and deeper into the tree represents adding more and more rooms to the level, with every seventh node visited adding one room. Eventually, a leaf of the tree will be reached, *i.e.*, a state where no valid answers exist at the next decision point.

The tree of integers we have described contains the entire search space of valid level layouts given some initial configuration. We will call this tree the *valid layout tree*. We cannot actually generate the valid layout tree in its entirety, as to do so would be computationally intractable for nontrivial level configurations. Nonetheless, the tree is a useful description of the space we wish to search for good level layouts. The goal of level layout is to search the tree selectively, looking for branches that lead to good levels.

Not all areas of the valid layout tree are rewarding to search. A *dead end subtree* of the valid layout tree is a subtree containing no nodes that place rooms. A dead end subtree may be at most six layers deep, because every seventh layer of the tree contains nodes that place rooms. Nonetheless, depending on the number of possible axioms in the grammar and the number of existing rooms and connectors in the current level, a dead end subtree may contain thousands of nodes. It is therefore a desirable quality for a search algorithm to be able to avoid or escape dead end subtrees.

A note on metaphors. As we describe our search algorithms, it is sometimes more convenient to speak in terms of the valid layout tree, and other times to speak in terms of the room placement machine. The room placement machine is a more concrete description of the system we implemented, while the valid layout tree is an abstract description of the space that the machine is used to explore. Any time we refer to an algorithm visiting nodes of the valid layout tree, it should be understood that, in the actual implementation, this is accomplished by issuing commands to the room placement machine.

4.3.5 Randomized Depth First Search

This section describes our randomized depth-first search (RDFS) algorithm. Our interest in using depth-first search for level layout was inspired by the work of Koens [Koens 2015], described in Section 3.1.3. Our level layout problem is significantly different from the room

layout problem addressed by Koens. For one thing, Koens was generating arrangements of polyominos (clumps of 2D tiles), while we are generating levels out of rooms (clumps of 3D shapes). For another, the goal of Koens' system was to find any layout that connected multiple endpoints, whereas our goal is to maximize a fitness function. Nonetheless, the two problems share some basic similarities. Both involve joining together pieces of content in a tree structure while attempting to find a good layout. It is interesting that a depth-first search was effective in Koens' problem, because it is quite a simple method. Therefore, it appeared worthwhile to investigate whether a similar method would prove effective for our level layout problem.

Our RDFS algorithm is designed to incorporate features of both a random search and a depth-first search while mitigating the major problems of each. A random search explores the valid layout tree by starting at the root and repeatedly moving to a random child. Once a leaf node is reached, the search returns to the root and starts over. The chief virtue of a random search is that it samples many different areas of the search space. However, it lacks any way to home in on a promising area of the search space. Thus, it is unlikely to find increasingly good levels over time except by sheer luck. We may say that the algorithm focuses entirely on exploration while eschewing any form of exploitation.

A particular problem faced by random search in our level generation framework is that it is unlikely to generate levels with many rooms. This is because our valid layout tree is littered with dead-end subtrees. To generate a level with many rooms, the search must "thread the eye of the needle" by repeatedly selecting deep branches purely by luck.

By contrast, a depth-first search algorithm can easily recover from selecting dead ends. Whenever a depth-first search reaches a leaf node, the search backtracks to the parent of the node and explores a different child. This is repeated until the search has exhaustively explored all descendants of each node. The eye of the needle is eventually threaded through trial and error. The problem with a pure depth-first search is that if it begins to search an unfruitful region of the tree, it will waste an enormous amount of time exploring every node in that area before moving on to a different region. It is an algorithm that focuses on exploitation to the exclusion of exploration. It is designed to exhaustively search an entire graph, something we do not wish to do.

Koens' solution to this problem is to reset the depth-first search periodically (every 5,000 iterations), forcing it to begin exploring a different region of the search space. For our RDFS

Algorithm 4-5. Randomized Depth-First Search (RDFS)

```
function RDFS (room placement machine RPM, function FitnessFunction,
        level configuration InitialConfiguration, integer CompleteResetPeriod,
        integer PartialResetPeriodPerRoom):
    RPM.InitializeLevel(InitialConfiguration)
    BestLevel ← RPM.GetLevel( )
    Iterations ← 0
    while Iterations < CompleteResetPeriod:
        Success ← TryToAddOneRoom(RPM)
        Iterations ← Iterations + 1
        if Success:
            Failures ← 0
            if FitnessFunction(RPM.GetLevel( )) > FitnessFunction(BestLevel)
                BestLevel ← RPM.GetLevel( )
                Iterations ← 0
        else
            Failures ← Failures + 1
            if Failures > (RPM.GetLevel( ).RoomCount( ) × PartialResetPeriodPerRoom):
                RPM.Backtrack(7)
                Failures ← 0
    return BestLevel
function TryToAddOneRoom (room placement machine RPM):
    StepsComplete ← 0
    ValidInputSet ← RPM.GetValidInputSet( )
    while ValidInputSet is not empty:
        StepsComplete ← StepsComplete + 1
        NextInput ← random member of ValidInputSet
        RPM.ProcessInput(NextInput)
        if a room was placed by RPM:
            # All seven steps are complete
            return true
        ValidInputSet ← RPM.GetValidInputSet( )
    RPM.Backtrack(StepsComplete)
    return false
```

algorithm, we adopt a similar approach, but we include the option of a partial reset as an alternative to a complete reset. A partial reset is a less extreme operation, which only forces the search to abandon the current branch and backtrack several levels up the tree. A complete reset forces the search to return to the root of the tree. In our algorithm, we do not reset the search after a fixed period. Instead, we delay resetting the tree as long as the area being explored appears promising. This allows our algorithm to strike a more intelligent balance between exploration and exploitation.

Our randomized depth-first search is given as Algorithm 4-5. The idea of the algorithm is as follows. The search explores the tree by choosing a random child of each node to explore.

When a leaf node is reached, the search immediately backtracks up the tree to the last node that successfully placed a room (these nodes occur at every seventh level of depth in the tree, because there are seven decisions to be made about the placement of each room). This is considered a failure. To prevent the search from becoming stuck in an area of the tree with many dead ends, we limit the number of times the search may fail back without successfully placing a room. If the number of failures exceeds the limit, we perform a partial reset by forcing the search to backtrack a further seven steps up the tree (removing the last room added to the level). The failure count is reset to zero each time a partial reset is performed or a room is successfully placed.

We also count the total number of iterations. The iteration counter is increased by one each time the search fails or succeeds to add a room to the level. If the iteration counter reaches its limit, we perform a complete reset, forcing the search to return to the root of the tree. However, we reset the iteration counter to zero every time a level is found with a better fitness score than the best level found since the last complete reset. This means we do not trigger a complete reset as long as the search is continuing to find better and better levels. We allocate more time to exploit promising areas of the tree.

The two reset counters serve different purposes. The partial reset counter restricts the amount of time the search may spend exploring an area of the tree where room placement is inhibited by many leaf nodes. A partial reset helps the search get out of difficult areas of the search space, and into areas where room placement is easy. Meanwhile, the complete reset counter restricts the amount of time the search may spend in areas that are both deep and unprofitable—that is, areas of the search space where it is easy to add rooms, but where the resulting levels do not score well on the fitness function.

The failure limit for the partial reset counter should be set to a low number so that the search will not waste too much time searching dead-end branches of the tree. A partial reset is not very costly as it only removes on room from the level. However, the counter must not be so low that the search struggles to ever find valid room placements. We observed that, in general, the proportion of room placements that are invalid increases with the as the number of rooms in a level increases. This happens for two reasons: firstly because the existing rooms occupy more of the available space in the level region, and secondly because the existing rooms may be closely packed together, making many of them unsuitable to serve as a parent room. Therefore, we

decided to set the failure limit for the partial reset counter based on the current number of rooms. The limit is the number of rooms times the constant *PartialResetPeriodPerRoom*. We use 2 for the value of this constant.

The iteration limit for the complete reset counter, *CompleteResetPeriod*, should be set to a higher value because a complete reset is more costly—it represents throwing away all the rooms the search has placed so far. Placing more rooms will not always lead to a good level, but for most fitness functions, a good level requires many rooms. We set *CompleteResetPeriod* to 5,000. This gives the algorithm time to test many variations on a potentially promising level layout.

Analysis. The RDFS algorithm is able to easily correct small, recent mistakes by backtracking. However, it cannot easily correct mistakes made early in the level generation process. For example, suppose the algorithm places one of the first rooms in the level in a poor position, then goes on to place many other rooms in good positions. The algorithm cannot remove the poorly placed room without first removing all the subsequent rooms. This is the major downside of using a depth-first approach within our level layout framework.

4.3.6 Monte Carlo Tree Search

Suppose you were playing a game of Go against the devil and found yourself unable to decide between two moves. How might you choose between them, assuming the devil kindly gave you a week to think about it? You might try playing each game out on your own, to see which move would lead to a win, but your own strategy is unlikely to be the same as the devil's, so the outcomes might not be the same. To gather more data, you enlist the help of a local children's Go club. You assign half of the club's members to play games starting from one move, and the other half to play games from the other move. Then you count which move won more games, and play that move against the devil. The children are not skilled Go players, and their strategy may be little better than random. Nonetheless, it is likely that the board position that led to more wins among unskilled players is the better board position overall.

This is the essential insight behind Monte Carlo tree search (MCTS), a type of algorithm that has proven effective in many domains [Browne 2012], though most famously in the game of Go [Coulom 2006; Silver 2016]. In a situation where it is computationally impossible to evaluate all possible outcomes, the results of each possible action can be repeatedly simulated

using a cheap method whose outcome may be expected to loosely correlate with the real results. The approximate value of each action can then be estimated and the best action chosen. MCTS is an any-time algorithm. The more time is available to make a decision, the more simulations can be performed. Based on the results of previous simulations, future simulations can be strategically targeted to further investigate the most promising options.

MCTS is effective in domains where early choices have long-term consequences. In Go, an early move may set the direction of the entire game, determining which moves are possible or desirable much later on. In this regard, level layout is a similar problem. The decision of where to place rooms early in the layout process has a huge impact on where it is possible or desirable to place rooms later on. MCTS can address such a problem by sampling the results of early choices and determining statistically which ones tend to eventually lead to problems or opportunities. Although MCTS is most commonly applied to adversarial games, it has also found application to various single-player puzzles and to optimization problems such as the Travelling Salesman Problem [Browne 2012].

We will now describe the MCTS as we applied it to level layout. Our algorithm is given in Algorithm 4-6. The algorithm gradually builds a search tree of possible moves and outcomes. This tree shares the same root and structure as the valid layout tree. However, the search tree represents only the explored portion of the valid layout tree, which is too large to generate in its entirety. A leaf node of the search tree may represent a dead end (if it corresponds to a leaf node of the valid layout tree), or a possible route for future expansion (if it corresponds to a non-leaf node of the valid layout tree).

Just as in the valid layout tree, each node of the search tree stores an integer representing an input to the room placement machine, and each node represents the level built by the sequence of integers leading down from the root to that node. Unlike in the valid layout tree, each node of the search tree also stores a value indicating its score. For a leaf node that cannot be expanded further, the score represents the actual fitness score of the level represented by that node. At any other node, the score is an estimate based on a sample of the node's descendants.

Each iteration of the MCTS algorithm can be divided into four stages: tree traversal, leaf expansion, playout, and backpropagation. In the tree traversal stage, the search starts at the root of the search tree. The search then proceeds down from each node to one of its children until a leaf of the search tree is reached. At each node, the child to explore is chosen using the *child*

Algorithm 4-6. Monte Carlo Tree Search (MCTS)

```
function MCTS (room placement machine RPM, function FitnessFunction,
        level configuration InitialConfiguration, tree node Root):
    RPM.InitializeLevel(InitialConfiguration)
    Cursor ← Root
    TreeTraversal(RPM, Cursor)
    Leaf Expansion(RPM, Cursor)
    if a node was expanded:
        BestLevel ← RDFS (RPM, FitnessFunction, RPM.GetLevel( ), 20, 2)
    else:
        BestLevel ← RPM.GetLevel( )
        mark node at Cursor for deletion
    Score ← calculate reward
    Backpropagation(Cursor, Score)
    return BestLevel
function TreeTraversal (room placement machine RPM, tree node Cursor):
    while Cursor is not a leaf:
        ChosenChild ← ChildSelectionPolicy(Cursor)
        RPM.ProcessInput(ChosenChild.Input)
        Cursor ← ChosenChild
function LeafExpansion (room placement machine RPM, tree node Cursor):
    ValidInputSet ← RPM.GetValidInputSet( )
    if ValidInputSet is not empty:
        for integer i in ValidInputSet:
            Cursor.AddChild(Input: i, TotalScore: 0, Visits: 0, Score: 0)
        ChosenChild ← ChildSelectionPolicy(Cursor)
        RPM.ProcessInput(ChosenChild.Input)
        Cursor ← ChosenChild
function Backpropagation (tree node Cursor, numeric score NewScore):
    Cursor.Visits ← 1
    Cursor.TotalScore ← NewScore
    Cursor.Score ← NewScore
    while Cursor is not Root:
        ChildNode ← Cursor
        Cursor ← Cursor.GetParent( )
        Cursor.Visits ← Cursor.Visits +1
        Cursor.TotalScore ← Cursor.TotalScore + NewScore
        Cursor.Score ← Cursor.TotalScore/Visits
        if ChildNode is marked for deletion:
            delete ChildNode
            if Cursor has no children:
                mark node at Cursor for deletion
```

selection policy. Once a leaf node of the search tree is reached, the leaf expansion stage occurs. The leaf node is expanded, adding its children to the search tree. Next is the playout stage. A simpler search algorithm, known as the *playout algorithm*, is used to quickly complete the level

layout. No nodes are added to the search tree during the playout stage. Finally, in the backpropagation stage, the completed level is evaluated using the fitness function. The level's score is stored at the newly expanded leaf node as an estimate of its quality. Additionally, the estimate at each of the expanded leaf node's ancestors is updated to incorporate the new score, using some policy called the *backpropagation policy*.

The leaf expansion stage may fail if the leaf node cannot be expanded. In this case, the playout stage is skipped. The level at the leaf node is evaluated and used for backpropagation. The leaf node is then culled from the tree because there is no need to search it again. A previously expanded node is also culled once all its children have been culled.

Child selection policy. Child selection policy is one of the most interesting parts of the MCTS algorithm. The goal of the tree traversal stage is to reach the leaf of the search tree that is most worthy of further investigation. The decision of which child to explore at each level of the tree can be seen as equivalent to a multi-armed bandit problem, *i.e.*, as the choice between playing several slot machines with different but unknown odds. Each time we play the same slot machine, we can improve the quality of our estimate of its average payout. The challenge is to balance playing the machine that appears to be best (exploitation) with testing other machines (exploration).

In the level layout problem, the reward we are hoping for is to find a level with a high score on the fitness function. Unlike in a traditional multi-armed bandit problem, we are not interested in gaining the same payout over and over—we are only interested in finding a level with a score higher than any other we have found so far. Nonetheless, at a single node of the tree, we may find that choosing to explore a particular child leads to higher rewards on average, because that child represents a decision that leads to many good level designs. For any given good level, it is possible that a similar variant with a slightly higher score exists, and such a variant is most likely to be found in a nearby branch of the tree.

It is not possible to determine a mathematically optimal strategy for dealing with slot machines with unknown random payout distributions. Nonetheless certain strategies have been found to be effective for many different types of distributions. The Upper Confidence Bound for Trees (UCT) policy and its variants are commonly used in MCTS. We used the version of this policy given by Browne et al. [Browne 2012 Survey].

In the UCT policy, the expected value of visiting each child i of a node n is given by the following equation:

$$UCT(i) = S_i + k\sqrt{\frac{2\ln(V_n)}{V_i}} \quad (1)$$

where S_i is the score stored at node i, k is a constant, V_n is the number of times node n has been visited, and V_i is the number of times node i has been visited. The child which scores the highest on this policy is selected to visit. An exception arises if any child has never been visited. In this case, the first unvisited child is selected to visit. Equation (1) cannot be used without a defined score S_i and a non-zero number of visits V_i for every child of n.

It may be observed that the square root term gradually decreases the more times node i is visited. This is the exploration term, used to prioritize visiting a child that has not been adequately explored. Meanwhile, the term S_i depends entirely on the historic payout of node i. This is the exploitation term. The constant factor k is used to control the balance between exploration and exploitation. We used $k = \frac{1}{\sqrt{2}}$, a standard value taken from Browne et al. [Browne 2012 Survey].

Playout algorithm. For our playout algorithm, we used our randomized depth-first search algorithm with a complete reset period of 20. The algorithm terminates quickly with this low parameter, while still finding better results than a pure random search.

Backpropagation policy. For the score at each node, we use the mean score of all samples from below that node. This represents the expected value of searching below the node. To allow us to calculate the mean score, we track the total score obtained below each node. Each time we backpropagate a new score to a node, we increase the total score by the new score, then divide the total score by the number of visits to find the mean score.

Analysis. The great advantage of the MCTS algorithm is that it is able to sample many different areas of the tree, returning to whichever appears most promising as it gathers more information. This is unlike the randomized depth first search, which effectively "forgets" each area of the tree when it leaves it.

The downside is that MCTS continuously consumes memory as it runs and expands its tree. Eventually it will consume all the available memory and be forced to stop. On our system, even when all available memory is used, the depth and breadth of the valid layout tree are such

that the deeper areas are only sparsely sampled. By contrast, the randomized depth-first search explores very thoroughly in specific areas deep in the tree.

4.3.7 Evolutionary Algorithm

Our evolutionary algorithm is a genetic algorithm. The algorithm consists of several components. Levels are evolved in a simple format, the *genotype*, which can be converted into a finished form, the *phenotype*. At any given time, a *population* of multiple candidate genotypes exists; this is simply a set of genotypes. Genotypes in the population are evaluated using the fitness function. In each iteration of the algorithm, a subset of the population is selected to survive, based on a *tournament*. Those not selected are replaced with new genotypes created by applying *mutation* and *crossover* operations to the survivors. We will now describe how each of these components works in our evolutionary algorithm.

Level genotype. The obvious choice for a genotype representation in our level layout framework would be an integer sequence representing an input sequence to the room placement machine. However, this obvious choice turns out to be a poor one. The numbers in the input sequence refer to existing rooms and connectors by index numbers. A subsequence taken from one sequence and inserted into another would no longer refer to the same rooms that it did before. Some of the numbers in the sequence might become invalid inputs in their new context. The connections between rooms would not remain stable through crossover.

We chose to instead use a tree-based genotype representation. Our representation is inspired by the work of Valtchanov and Brown [Valtchanov 2012], who obtained good results evolving 2D level layouts using a tree as a genotype. Our genotype differs from theirs because we use rooms generated by our room generator, whereas their system builds levels from a collection of prefabricated rooms.

In our system, a level genotype is a tree called a *genotype tree*. To avoid confusion, please note that a genotype tree does not correspond in structure or content to the valid layout tree.

Each node of the genotype tree stores a complete room as a collection of shapes. Additionally, the tree stores information about how rooms should be connected. Recall that in our tree extension algorithm for level layout, each room is placed by connecting it to a parent room. The genotype tree for a level replicates these connections. That is, if room p is the parent

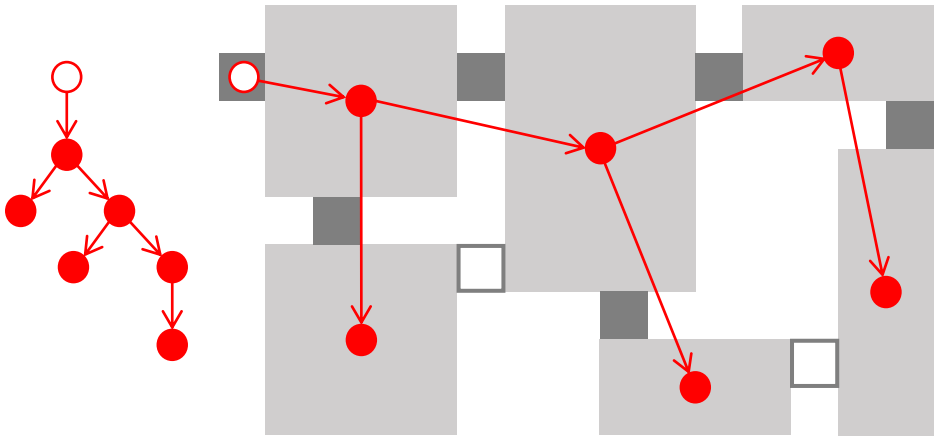


Figure 4-7. Correspondence between a genotype tree (left) and a level layout (right). Connections used to place rooms (small dark squares) are represented by the links of the tree. Incidental connections (small white squares) are not represented in the genotype, but are added automatically between adjacent rooms.

of room *r*, then the node for *p* in the genotype tree is the parent of the node for *r*. The root of the tree stores the start room. The correspondence between a genotype tree and a level layout is shown in Figure 4-7.

Recall that there are many possible ways a new room may be connected to its parent room (see section 4.3.2). In particular, any connector of the new room may be connected to any open connector of the parent room. To resolve this ambiguity, each node in the genotype tree stores two connector indices. The first index indicates a connector on the parent room, while the second index indicates a connector on the new room. These two connectors are aligned when placing the new room.

Given all these requirements, the information stored at each node of the genotype tree includes a link to a parent node, a room (a collection of shapes), two connector indices, and a list of links to child nodes. The root node is an exception, as it represents the start room. The start room consists of only a single connector, and its placement is unambiguously specified as part of the level region configuration (see section 4.3.1). Therefore, the only information stored at the root node of the genotype tree is a single link to a child node.

Level phenotype. A genotype tree can easily be instantiated as a finished level layout. First, the start room is placed. Then the tree is traversed in a depth-first order. Each node contains the information to place a room relative to its parent. Incidental connections are not

represented in the genotype tree, but are added automatically between adjacent rooms as described in section 4.3.2.

Population. We used a population with a size of 100 genotypes. Each genotype tree is initialized as a single node representing the start room, according to the initial level configuration.

Tournament. In each iteration of our evolutionary algorithm, we divide the population into 25 randomly selected groups of 4 genotypes. Each genotype is instantiated as a level layout, then evaluated with the fitness function. The two best genotypes are retained, while the other two are marked for replacement.

When a genotype is replaced, there is a 25% chance it is replaced using crossover between the two surviving genotypes in its group. Otherwise, it is replaced using mutation on a random one of the two survivors. There is a chance that crossover may fail, in which case mutation is used as a fallback.

Crossover. Crossover is an operation which creates a new genotype from two parents, which we designate randomly as the mother and father. During crossover, we first create the new genotype by copying the mother. We then replace a random subtree of the new genotype with a random subtree of the father genotype. The node at the root of the copied subtree is changed to make it connect to the appropriate parent connector.

It is possible that some rooms of the copied subtree cannot be placed in their new context. These nodes should be removed from the genotype. We instantiate the new genotype as a level layout and delete the nodes for any rooms that cannot be successfully placed. If at least one room from the new subtree is successfully placed, we consider crossover a success. If none of the rooms can be placed, we restart the crossover process, hoping for a better selection of random subtrees. If no rooms are placed after ten attempts, we consider crossover a failure.

Mutation. Mutation is an operation that creates a new genotype from a single parent genotype. We use two different mutation operations: subtree removal and room addition. When we mutate a genotype, there is a 50% chance that we apply subtree removal. In subtree removal, we select a random node of the genotype tree other than the root, and delete the subtree rooted at that node. Subtree removal fails if the genotype tree has only one node.

If we apply subtree removal, there is a 50% chance we also apply room addition. If we do not apply removal, or removal fails, we always apply room addition. In room addition, we

attempt to add one or more new node to the genotype. To do this, we instantiate the level and then select random valid answers to the seven questions of the room placement machine (see Section 4.3.2). If placement succeeds, we add the room to the genotype. We repeatedly attempt to generate and place rooms until three rooms have been successfully placed or until a total of ten attempts have been made. If both subtree removal and room addition fail, mutation produces an exact copy of the parent genotype.

Analysis. An evolutionary algorithm has the power to explore the search space in multiple ways. Mutations normally make only small changes to an existing solution (exploitation). Because many different individuals are generated and only the best kept, an evolutionary algorithm can act as a hill-climbing algorithm, gradually moving towards a local maximum. On the other hand, crossover operations can make large and drastic changes to an existing solution. This allows the algorithm to sample distant parts of the search space (exploration), while ensuring that the new level sampled nonetheless has some features in common with some of the best known levels.

In concrete terms, a distinct advantage of our evolutionary algorithm over our other two search algorithms is that it can remove a room from a level without backtracking to remove all the other rooms added since. This allows the algorithm to more easily recover from poor decisions early in the level layout process.

4.4 Mesh Placement

The third major module of our level generation system is the mesh placement module. This module takes the bare bones of the level and girds them in flesh. The input to the mesh placement module is a level layout composed of rooms composed of shapes. The output is a list of meshes to place, with a transformation to be applied to each mesh. To control mesh placement, the designer must define a series of mesh placement rules that are specific to the mesh kit being used. These rules act as a counterpart to the room grammar.

Our approach to mesh placement is to break down the space of a level into a series of slots in which meshes can be placed. We process all the shapes in the level and flag the slots where meshes need to be placed. We then process all the flagged slots and place the required

meshes. The processing of shapes and slots takes place over several successive passes, so that the flags added during each pass can be checked during later passes.

4.4.1 Challenges and Requirements

Mesh placement is a problem that appears simple but conceals some tricky complications. We originally conceived of mesh placement as being a part of the room generation process, and intended that meshes would be the terminal shapes placed by our shape grammar. Such an approach was used in the systems for generating exterior architecture discussed in section 3.2.1 of our literature review. Perhaps inspired by these systems, multiple authors including Ferrari [Ferrari 2013] and Dormans [Dormans 2011] have speculated that using a grammar to place meshes and add detail to interior level architecture would be a promising approach. However, when we attempted to implement such a system, we found shape grammars ill-suited to address certain cases that naturally arise in interior architecture.

Here, we will outline two key cases: border meshes and meshes with context-sensitive variants. We will show the problems that arise in these cases when using a shape grammar for mesh placement, in order to demonstrate why the cases are challenging and to suggest what is required for a better approach.

Border meshes. One challenging case is the placement of meshes that lie on the border between two different spaces. Consider one of the most fundamental elements of interior architecture: the wall between two rooms. The wall cannot be said to belong solely to either room, since it lies on the border between them. This means that the wall cannot easily be placed by further subdividing either room. The most likely outcome would be that two overlapping copies of the wall were placed, one from each side. Overlapping meshes like this can produce visual artifacts and are not an acceptable solution.

Note that these border cases do not normally emerge when generating exterior building architecture, since by definition the exterior wall of a building borders only on empty space. However, border pieces are an essential feature of interior spaces.

To handle such a border case in a shape grammar approach, it would be necessary to place the wall at the same time as placing the two rooms, perhaps by splitting a larger space into a large room section, a tiny wall section, and then another large room section. It would be very cumbersome to design rules in such a manner. Furthermore, such an approach is incompatible

Figure 4-8. Architecture built using the *Multistory Dungeons* mesh kit [Multistory]. A mesh can be placed within a grid cell (groin vault), on the face between two cells (wall), or on the edge between four cells (columns).

with any level layout method not based on subdivision, such as our search-based methods for level layout. A better mesh placement system would analyze the border between the two rooms and place a single wall between them.

Context-sensitive variants. Another challenge relates to meshes with context-sensitive variants. Consider a wall made by placing copies of two meshes: a middle piece and an end piece. A well-formed wall consists of an end piece, a series of middle pieces, and then another end piece. We could build such a wall in a shape grammar by using a split rule to decompose a single shape into the required components. However, the grammar fails in the event that there are two wall shapes placed end to end. The result is two adjacent end pieces where the shapes meet. We would prefer that the two shapes merge seamlessly into a single wall. A good mesh placement system would examine the context of each piece of wall individually to determine whether it should be a middle piece or an end piece.

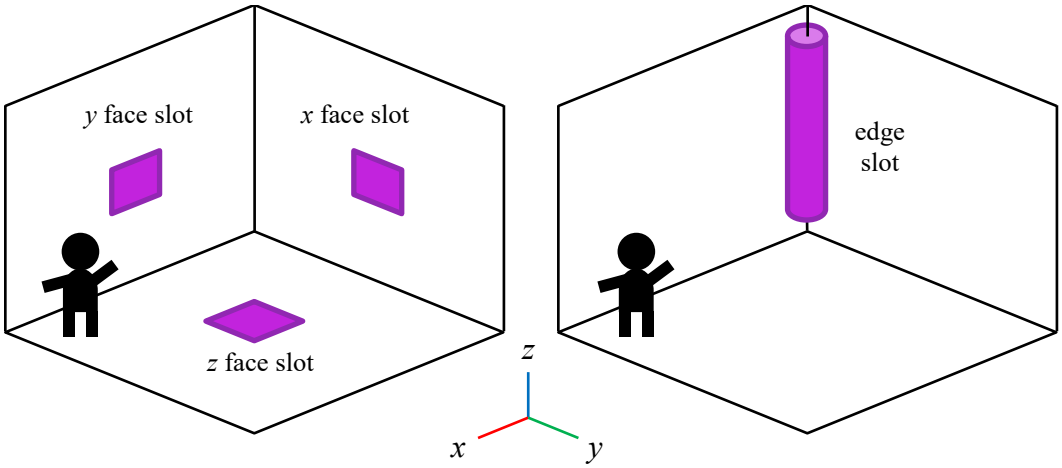


Figure 4-9. The face slots (left) and edge slot (right) associated with a grid cell.

Analysis of requirements. A common theme in both challenging cases is that choosing the right mesh to place is a context-sensitive decision. Perhaps, then, it is not surprising that context-free grammars are a poor solution.

A better solution would meet certain requirements. First of all, a mesh should not always correspond to a single shape in the level. A mesh might be placed as the result of multiple shapes, as in the case of a wall between two rooms. Secondly, when placing a mesh in a given location, it should be possible to query other nearby locations for relevant information. This would allow context-sensitive mesh variations. Our mesh placement module was designed to meet these requirements.

4.4.2 Mesh Slots

A mesh slot is a position in space where a mesh can be placed. Mesh slots are based on the rectangular grid described in Section 4.1.1. We allow a mesh to be placed at any transformation relative to a mesh slot. This allows meshes to be placed in any position, while taking advantage of the fact that mesh kits are normally designed around a grid.

By analyzing our two chosen mesh kits, we identified three main ways that meshes are commonly designed to be placed in a level. The simplest type of mesh fits within a grid cell. Other meshes, such as walls and railings, are designed to lie on the face between two cells. Finally, some meshes, such as columns, are designed to lie on the edge between four cells (at the intersection of four faces). Examples are shown in Figure 4-8. This observation leads us to

Table 4-3. The engine position of each slot at grid position (G_x, G_y, G_z), where the grid size is (S_x, S_y, S_z).

Mesh Slot	Engine x position	Engine y position	Engine z position
Cell slot	$S_x * (G_x + 0.5)$	$S_y * (G_y + 0.5)$	$S_z * (G_z + 0.5)$
x face slot	$S_x * G_x$	$S_y * (G_y + 0.5)$	$S_z * (G_z + 0.5)$
y face slot	$S_x * (G_x + 0.5)$	$S_y * G_y$	$S_z * (G_z + 0.5)$
z face slot	$S_x * (G_x + 0.5)$	$S_y * (G_y + 0.5)$	$S_z * G_z$
Edge slot	$S_x * G_x$	$S_y * G_y$	$S_z * (G_z + 0.5)$

define three types of mesh slot: cell slots, face slots, and edge slots. In our system, we only use vertical edge slots because we did not find any meshes that required horizontal edge slots to place.

There are five mesh slots for each position in the grid: one cell slot, three face slots (x, y, and z), and one edge slot. These five mesh slots all correspond the same position in grid coordinates, but each corresponds to a different position in engine coordinates. The cell slot corresponds to the position at the centre of the grid cell, while the other slots each correspond to a position at the centre of the appropriate face or edge, as shown in Figure 4-9. An equation for the position of each slot in engine coordinates is given in Table 4-3. The engine position of each slot at grid position (G_x, G_y, G_z), where the grid size is (S_x, S_y, S_z).

Some meshes are large enough to occupy multiple slots of the same type, *e.g.*, a long wall could fill several adjacent face slots. However, in our system each mesh is placed relative to a single slot.

Flags. A mesh slot can be marked with one or more flags. For example, a vertical edge slot could be marked with the flag *Column* to indicate that a column should be placed at that edge. Attempting to flag a slot with a flag that is already present has no additional effect. Flags can be defined as necessary to meet the needs of a particular mesh kit.

The flags for each slot are stored in an associative container indexed by the slot's grid position (G_x, G_y, G_z). The container used must support both fast lookup of individual members and iteration over all members in the order of their keys. We used the *map* data structure from the C++ Standard Template Library, which is backed by a red-black tree. This provided adequate performance for our purposes, although for significantly larger or more complex levels it could be advantageous to use a data structure specifically designed for storing sparse volumetric data.

4.4.3 Mesh Placement Algorithm

Our mesh placement algorithm consists of six passes over the entire level. Each pass flags mesh slots, adds meshes to the output, or both. Passes in which no meshes are placed are called pre-processing passes. The six passes we perform are as follows:

1. Shape pre-processing.
2. Shape processing.
3. Side face processing.
4. Bottom face processing.
5. Edge pre-processing.
6. Edge processing.

We will now describe what happens during each pass, using examples from the mesh placement rules used to complete the cloister and cathedral rooms shown in Section 4.2.5. The full mesh placement rules from these examples are given in Appendix C.

Shape pre-processing. During this pass, we flag cell slots. To do this, we iterate over every shape in the level. For each shape, we flag the cell slot at every grid position occupied by the shape. The flag used for each shape is the same as the shape's symbol in the room grammar. For example, each cell slot in a *GroinVault* shape is marked with a *GroinVault* flag. These flags can be queried during later passes to make context-sensitive decisions about which meshes to place.

Shape processing. During this pass, we place meshes that occupy cell slots, while flagging face and edge slots. We iterate over every shape and execute a function based on each shape's symbol. A function may be specified for each possible symbol, or multiple symbols may correspond to the same function. The details of these functions depend on the needs of the mesh kit and architectural style, but generally they are written using a small set of useful operations.

One operation is to place a mesh at every cell slot in a shape's volume. This is useful for meshes that tile to form a larger structure. For example, in a shape with the *GroinVault* symbol, we place a copy of the groin vault mesh at every cell slot. A relative transformation may be given, which will be applied to each shape after it is placed at its cell slot.

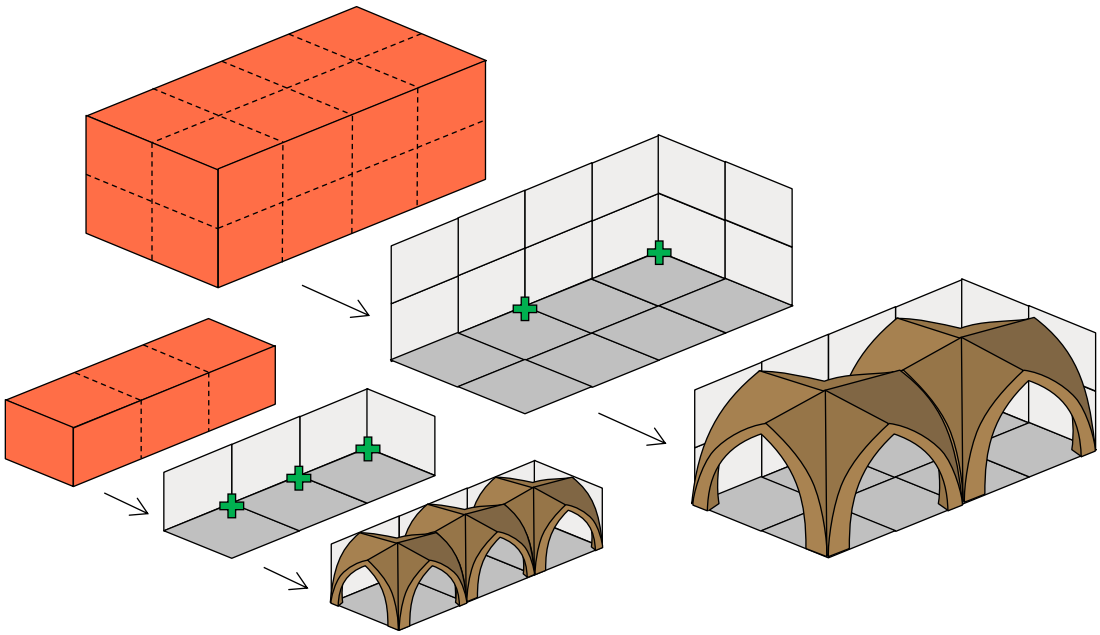


Figure 4-10. Placement of standard and large groin vault meshes. This diagram shows the original shape (orange block), the cell slots used for mesh placement (green plus marks), and the placed meshes.

Another operation is to place meshes at evenly spaced cell slots in a shape's volume. This operation is useful for large meshes. For example, the large groin vault mesh in the *Multistory Dungeons* kit is large enough to take up two grid cells in each direction. Therefore, in the shape processing function for shapes with the *LargeGroinVault* symbol, we place a copy of the mesh at every second cell slot in each dimension. The relative transformation for each mesh is used to centre it across its own cell slot and the adjacent cells. Figure 4-10 illustrates the placement of standard and large groin vault meshes. It may be observed that interval-based placement will only work for shapes of an appropriate size in grid units. The large groin vault meshes only work for shapes that are two units tall and an even size in each horizontal dimension. We ensure shapes will always be of an appropriate size by setting size restrictions in the relevant symbol of the grammar (see Section 4.2.1).

Other operations during this stage are used to flag face slots and edge slots. For most shapes, a useful operation is to flag all exterior side faces with a *CheckWall* flag, which is tells the system to check during a later pass whether a wall should be placed. Likewise, it useful to flag the bottom faces of many shapes to indicate a floor mesh should be placed. For a colonnade shape, we apply the *Column* flag to all vertical edge slots inside and surrounding the shape. This

indicates that a grid of columns is to be placed. These operations can also be invoked to add flags at intervals. For a large colonnade, we add the *Column* flag only at every second vertical edge slot in the *x* and *y* dimensions. These evenly spaced columns will correspond to the corners of large groin vault meshes above.

During shape processing functions, we can query the flags at nearby cell slots to determine whether certain operations are required. For example, we can query the cell slot above the current shape to see if it has the flag for a balcony. If it does, we add column flags so that columns will be placed to hold up the balcony.

Side face processing. During this pass, we add meshes at side face slots where required. We iterate over all flagged *x* and *y* face slots. For each slot, we select one or more functions to call based on the flags at the slot. The mesh placement cases for *x* face slots and *y* face slots are conceptually identical, but rotated ninety degrees. To avoid writing duplicate cases, we implemented functions only for the *y* face slots. The *x* face slots call the same functions but apply a 90° rotation matrix to all meshes placed.

The functions for mesh placement at side faces commonly query the flags of the cell slots at the two adjacent cells. For example, when we process a slot with the *CheckWall* flag, we check whether there is a shape on each side of the face. If there is a shape on only one side, we place a wall mesh facing that side. If there are shapes on both sides, no wall is placed, so adjacent shapes join into non-rectangular open spaces.

Bottom face processing. This pass adds meshes at bottom face slots. This includes floor and ceiling meshes. The process is analogous to side face processing.

Edge pre-processing. The purpose of this pass is to set edge slot flags that can only be set based on other flags at surrounding edge slots. We found it necessary to add this pass because of the many complex cases involved in placing variant columns. The *Multistory Dungeons* kit contains columns of two heights. The short columns are one cell tall, while the tall columns are two cells tall. To make a column more than two cells tall, multiple meshes must be placed in a stack. The standard column meshes each have a large plinth at the bottom, but there is a variant with a small plinth that looks better when stacked on top of another column. For an example of the correct placement of these column variants, see Figure 4-11.

In the edge pre-processing step, we go through all the edge slots marked with a *Column* flag and determine which edges should contain the base of a column. We mark these columns

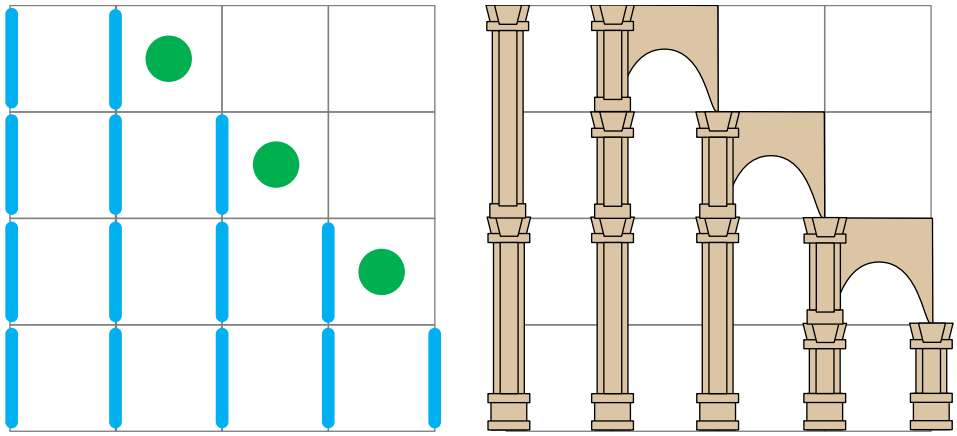


Figure 4-11. Correct placement of variant column meshes based on *Column* flags (blue bars) and *GroinVault* flags (green circles).

with the *ColumnBase* flag. The other edges will contain only the upper half of a tall column, so they will not require a mesh to be placed. The rules for this process are as follows. An edge slot should be marked *ColumnBase* if it has the *Column* flag but the edge slot directly below does not; or if the edge slot directly below it has the *Column* flag but not the *ColumnBase* flag. Note that for this latter check to work correctly, edges must be processed in increasing order of their *z*-coordinate.

In some cases, an edge may already have been marked with the *ColumnBase* flag during an earlier pass. For example, our shape processing function for the *GroinVault* space type marks all surrounding edges with the *ColumnBase* flag because the groin vault mesh does not look good beside the top half of a column. These cases do not interfere with the edge pre-processing step.

Edge processing. In this step, we place meshes at edge slots. In the mesh kits we used, this involves only column meshes. We place a column mesh at each edge slot with the *Column* and *ColumnBase* flags. If the edge slot above has the *Column* flag but not the *ColumnBase* flag, we place a tall column; otherwise, we place a short column. If the edge slot below has the *Column* flag, we use the variant mesh with a small plinth.

5 Evaluation

This chapter presents some of the results produced by our level generator and evaluates the extent to which these results achieve our goals. We also compare the effectiveness of the three search algorithms we tested for level layout generation.

5.1 Room Generation and Mesh Placement

In Section 4.2.5, we gave two examples of rooms generated by our room generation module, a cloister courtyard and the main room of a cathedral. Figure 5-1 and Figure 5-2 show the same two rooms in *Unreal Engine 4*, after mesh placement. Here we also show several other examples of rooms generated with our system. Figure 5-3 shows a stairway generated using three rules in our room grammar. Figure 5-4 shows a multi-level room with balconies, generated using five rules (including one of the same rules used to generate a cloister). Figure 5-5 shows a large room with a raised ceiling, generated using three rules in our room grammar (including one of the same rules used to generate a cathedral). Meshes for all these rooms were placed using the same mesh placement rules.

These results show that our room generation and mesh placement methods satisfy the major requirements set forth in Chapter 1. Our system can generate rooms that are detailed and architecturally realistic, with three-dimensional elements such as stairs and balconies. The rooms are assembled from modular meshes according to structural principles specified in simple rules. These rules can encode designs from actual architectural traditions, in this case medieval church architecture. A small set of rules can be used to generate many rooms of different sizes but similar structures. Figure 5-6 shows examples of multiple rooms generated by our room grammar from axioms with the same symbol but different sizes. A useful feature of our method

Figure 5-1. Exterior and interior views of the finished cloister room (after mesh placement).

is that rules in a room grammar can be reused to generate similar structures across multiple different types of rooms, as shown in our examples which share several rules.

Our room generation module produces deterministic results given the same axiom. This is a valuable feature when experimenting with different designs with search algorithms. The downside of this feature is that each axiom symbol tends to produce a single characteristic room layout, limiting the total variety of different results that can be produced. This limitation could be addressed within our system by making heavier use of attributes to control rule selection. For

Figure 5-2. Exterior and interior views of the finished cathedral room (after mesh placement).

Figure 5-3. A stairway room.

Figure 5-4. A multi-level room with balconies.

Figure 5-5. A large room with a raised ceiling in the centre.

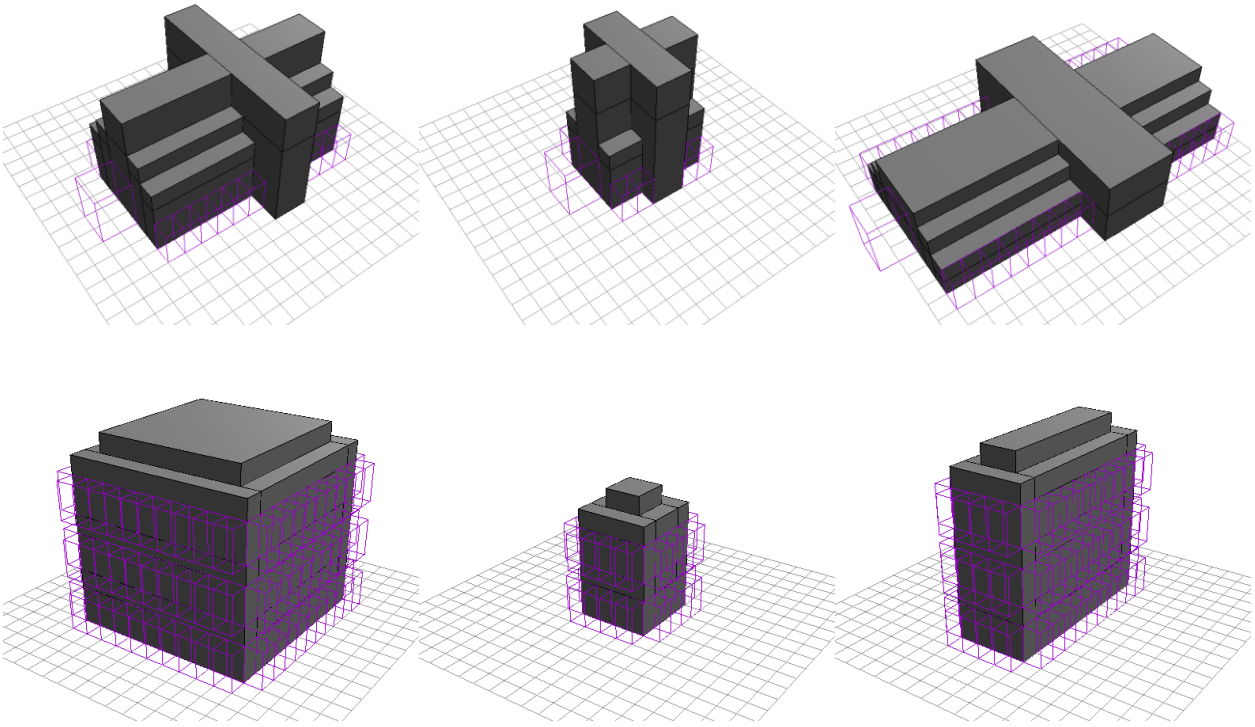


Figure 5-6. Rooms generated by applying the same rules to differently sized axioms. Top row: rooms generated from the cathedral axiom. Bottom row: rooms generated from the multi-level room axiom.

example, different rules could be used to divide an axiom into either a square, an L-shape, or an X-shape, depending on an attribute.

In our room grammars, we specify rules in a procedural format, where the right-hand side of a rule is given as a series of operations. This format is very powerful, allowing many different designs to be expressed. On the other hand, it is needlessly verbose in certain cases. Some of the rule operators are concise and easy to use, especially the split operations which can divide one shape into two or into a row of shapes of variable sizes. However, other cases (such as splitting a room into a central block and an outside ring) are cumbersome to specify in our grammar, requiring multiple operations for a conceptually simple task. The design of the grammar could be improved by identifying other common structures and implementing them as single operations.

The most awkward aspect of designing rules for the room grammar is working around the requirements of mesh placement. We found this especially problematic when designing the rules for the cathedral room example. Because we wanted to use large groin vaults in the nave, transept, and sanctuary of the cathedral, it was necessary to ensure that these areas would always have sizes divisible by two. This meant the cathedral shape could not simply be divided into the desired proportions at each level of subdivision. Instead, in our grammar (see Appendix B) we used special-case functions to calculate in advance the amount of space required for each section of the cathedral, shift the proportions so that the size of each element is a multiple of two, and store these results in attributes so they can guide later subdivision. This solution is not very elegant. The underlying cause of the problem is that the sizes and shapes of meshes available in the mesh kit place implicit constraints on what structures can be built with them. We rely on special-case functions in our grammar because we lack a better way to represent these constraints and take them into account.

A deeper limitation of a grammar approach to room generation is that grammars are best suited to generating content with a regular and predictable structure. None of the rooms we have generated with our system are as complex and interesting as the room shown in the Piranesi etching on page x. Piranesi's etching contains something like a three-dimensional maze of platforms, archways, and staircases, all built around a regular grid of arched columns. A more extreme example of a similar design is found in the plate XIV of Piranesi's *Carceri d'Invenzionne* ("Prisons of the Imagination"), shown in Figure 5-7. Although there are clearly

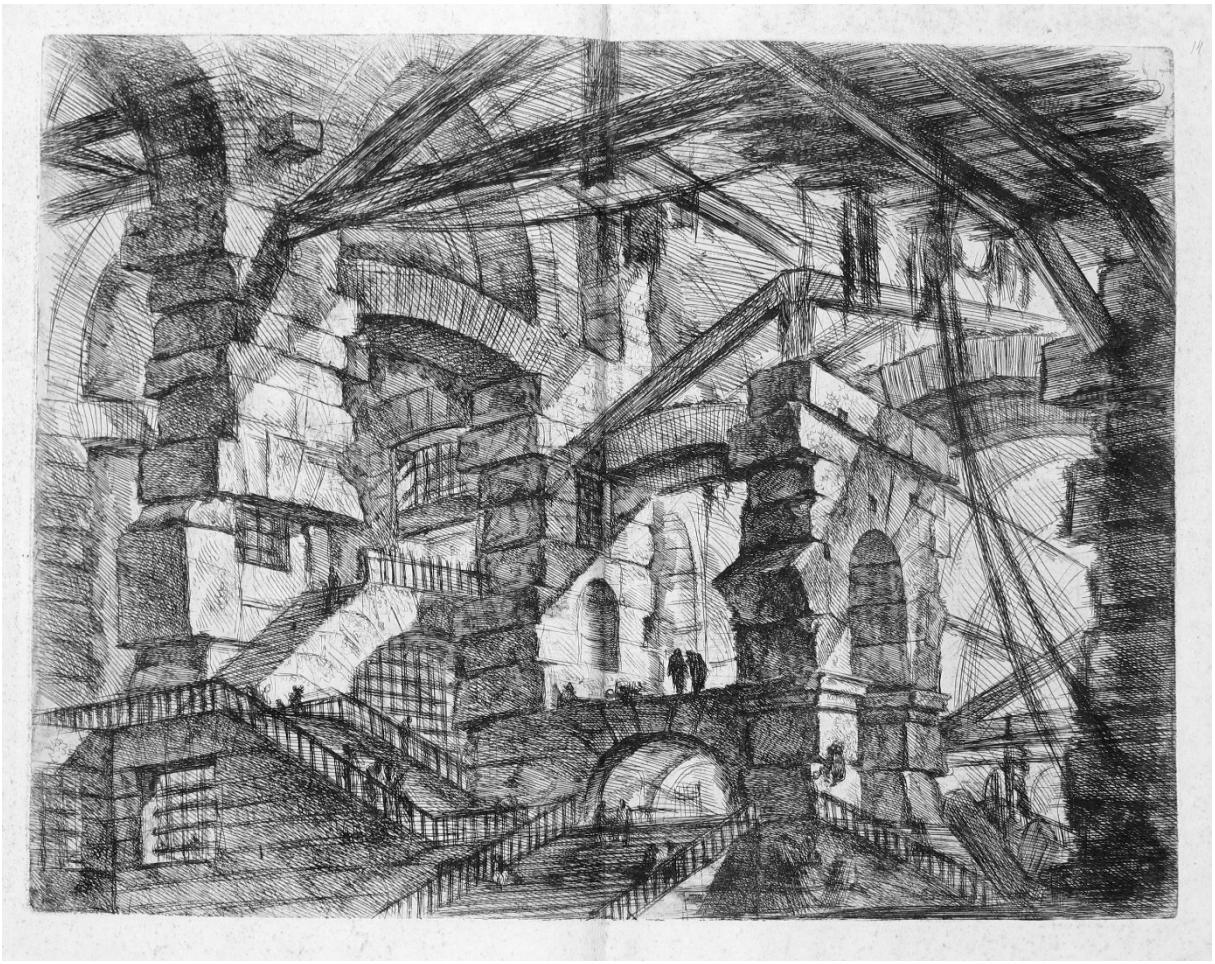


Figure 5-7. Plate XIV, "The Gothic Arch," from Piranesi's *Carceri d'Invenzione* [Piranesi 1745].

patterns and rules to the design, the overall layout is chaotic rather than structured. We were unable to come up with any way to create such layouts in our room grammar. It seems likely that other methods would be better suited to generating this type of chaotic room design—perhaps search methods similar to those we used to generate level layouts at a larger scale.

So far, all of the examples we have shown were generated using the *Multistory Dungeons* mesh kit. One of our goals for our generator was that it would be possible to apply to multiple different mesh kits and architectural styles. To test the versatility of our system, we also designed grammar rules and mesh placement rules to generate levels using the modular hallway meshes from the *Soul: City* mesh kit by Epic Games [Epic 2014]. Figure 5-8 shows part of a level we generated using this kit.

Figure 5-8. Hallway generated by our system using the *Soul: City* [Epic 2014] mesh kit.

We found that the most difficult part of using our system with a different mesh kit was simply understanding how the meshes in the new kit were intended to fit together. The meshes in the *Soul: City* kit are built around the grid somewhat differently than in the *Multistory Dungeons* kit. In particular, in *Soul: City*, the floor pieces are only two-thirds as long as the wall pieces, so three pieces of floor must be used for every two pieces of wall; and the walls are designed to fit within grid cells, rather than lying between them. Once we understood these relationships, we were able to express them in our system without difficulty using shape size constraints and mesh placement rules.

5.2 Search Algorithms

A good search algorithm for level layout is one that quickly finds levels with a high fitness score. Of course, the performance of a search algorithm may depend on the problem configuration—that is, on the fitness function and the level region. A search algorithm that performs well across

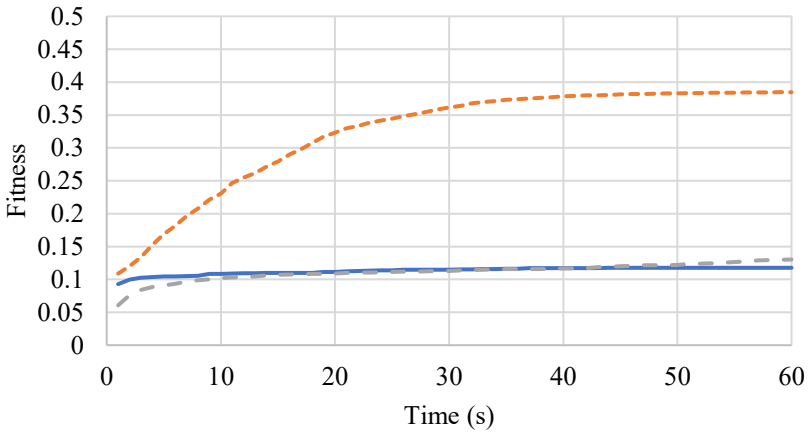
Table 5-1. Test cases for level layout search functions.

Case	Region	Fitness Function
Fill Cube	Volume: (-25, -25, 0) to (25, 25, 25) Start: (-25, 0, 0)	*VolumeFill*(*MaxPercent*: 100%)
Tower	Volume: (-10, -10, 0) to (10, 10, 50) Start: (-10, 0, 0) Endpoint: (0, 0, 45)	*CheckpointProximity*
S-Shape	Volume: (-20, -25, 0) to (20, -15, 10) Volume: (10, -15, 0) to (20, -5, 10) Volume: (-20, -5, 0) to (20, 5, 10) Volume: (-20, 5, 0) to (-10, 15, 10) Volume: (-20, 15, 0) to (20, 25, 10) Start: (-20, -20, 0) Checkpoint: (15, -20, 0) Checkpoint: (15, 0, 0) Checkpoint: (-15, 0, 0) Checkpoint: (-15, 20, 0) Endpoint: (15, 20, 0)	*CheckpointProximity*
Long Linear	Volume: (-25, -25, 0) to (25, 25, 10) Start: (-25, 0, 0) Endpoint: (20, 0, 0)	*LinearRooms*(*MaxPercent*: 90%) *CheckpointProximity* *LevelLength*(*LengthFactor*: 3) *RoomsOnAnyPath*(*MaxPercent*: 100%)
Branching Paths	Volume: (-25, -25, 0) to (25, 25, 10) Start: (-25, 0, 0) Endpoint: (20, 0, 0)	*AverageConnections*(*MaxCount*: 4) *CheckpointProximity* *LevelLength*(*LengthFactor*: 2) *RoomsOnAnyPath*(*MaxPercent*: 100%)
Complex Maze	Volume: (-25, -25, 0) to (25, 25, 10) Start: (-25, 0, 0) Endpoint: (20, 0, 0)	*RoomCount*(*MaxCount*: 50) *DeadEndRooms*(*MaxPercent*: 40%) *JunctionRooms*(*MaxPercent*: 40%)

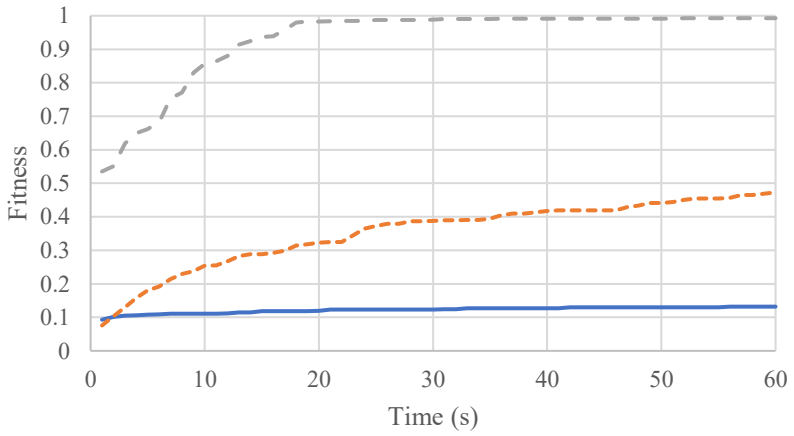
multiple fitness functions is especially good, because this means it its output can effectively be controlled.

To compare our three different search algorithms, we tested each of them on six different configurations, which are listed in Table 5-1. Each test case was designed to represent a different set of qualities that a might be desirable in a level design. The test cases were also chosen to include all supported features of our fitness function between them. The simplest case, *Fill Cube*, requires the algorithm to fill as much of the level volume as possible with rooms, creating a dense level in a limited area. The *Tower* case tests the search algorithm's ability to build in three dimensions by requiring a level to be built in a tall and thin region volume with the start point at the bottom and the end point at the top. The *S-Shape* case tests the algorithm's ability to build around corners in an S-shaped level region. Checkpoints are placed at each

(1) *Fill Cube*



(2) *Tower*



(3) *S-Shape*

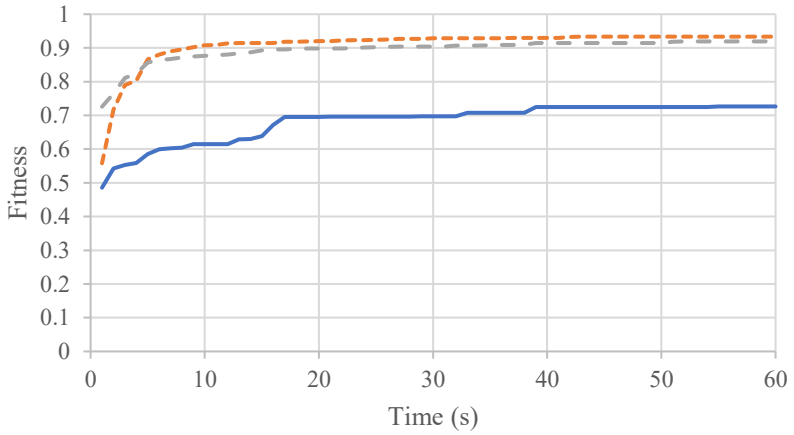
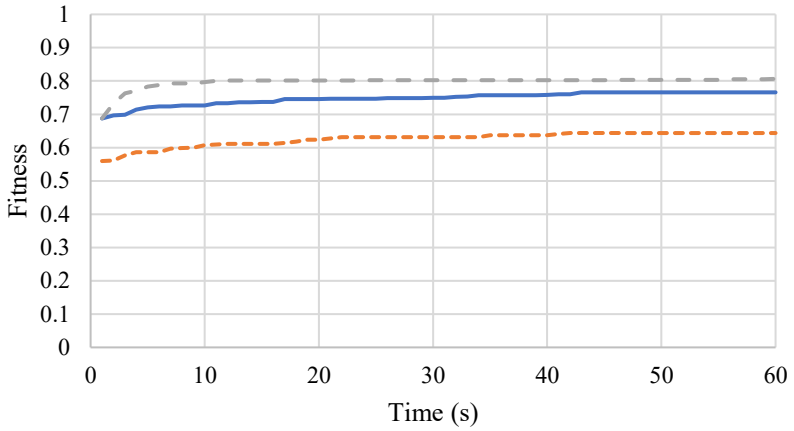
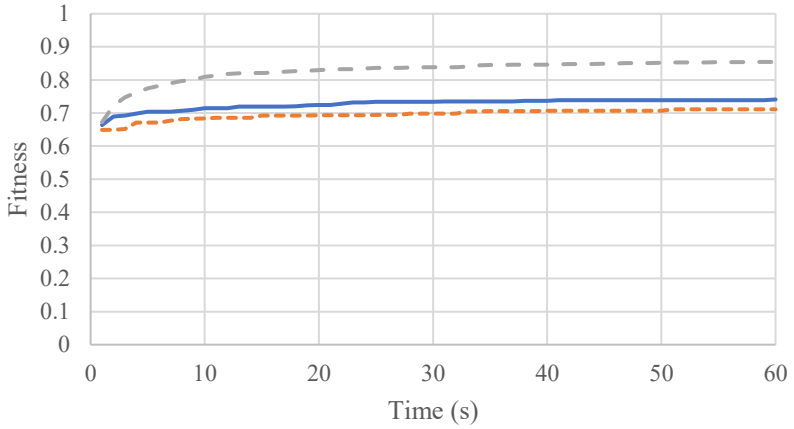


Figure 5-9. Comparison of search algorithms (continued on next page).

(4) *Long Linear*



(5) *Branching Paths*



(6) *Complex Maze*

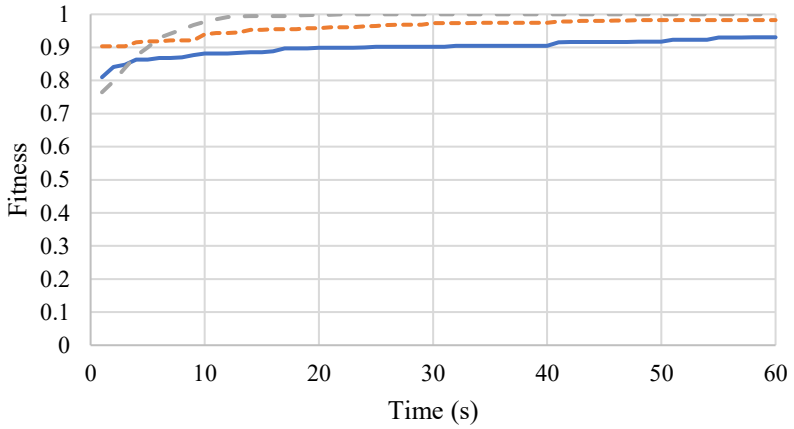


Figure 5-9 (cont'd). Comparison of search algorithms.

corner to help guide the algorithm. The *Long Linear* case requires the algorithm to make a long, convoluted path between the start point and the end point with few branches or dead ends. The *Branching Paths* case requires the algorithm to make many rooms with many connectors, while still ensuring that every room is on some path to the end point. The *Complex Maze* case requires the level to contain both many junctions and many dead ends, and to include as many distinct rooms as possible. In each case, the included features of the fitness function are equally weighted.

Note that fitness scores between different cases are not directly comparable because some fitness functions are easier to fulfill than others. For example, to achieve a perfect fitness of 1.0 on the *Fill Cube* test, it would be necessary for the rooms of a level to entirely fill the region volume. This is not possible in our system, since to do so would leave no space between rooms for solid walls. On the other hand, on the *Tower* and *S-Shape* tests, any level that successfully connects the start point to the end point will receive a perfect score.

Each search algorithm was tested ten times on each test case. In each trial, the algorithm was allowed to run for one minute and the fitness of the best level found so far was recorded once per second. The results of the ten trials were then averaged to give an overall measure of the algorithm's performance on that test case. All tests were performed on a single computer running Windows 10 with a 4.00 GHz Intel Core i7-6700K processor. The average results are shown in Figure 5-9.

The RDFS algorithm is very good at filling space with rooms. The algorithm's partial reset feature allows it to quickly escape from dead-end branches of the valid layout tree and keep adding rooms to a level. Thus it performs very well on the *Fill Cube* test case (see Figure 5-10) and the *S-Shape* test case. It performs only moderately well on the *Tower* case, which requires it to add staircase rooms in particular. It does gradually find its way up the tower, but wastes a lot of time and blocks itself by adding non-staircase rooms that do not help it climb. The RDFS algorithm does poorly on the *Long Linear* and *Branching Paths* cases. Both these cases penalize the algorithm for creating dead ends, which it tends to do while aggressively adding rooms. The RDFS algorithm cannot easily remove a badly placed room from the level if it has subsequently added other rooms. Its only options are a partial reset (removing the last room placed) or a complete reset (throwing away the entire level). It does perform well on the *Complex Maze* case, another case that rewards aggressively placing rooms wherever they fit. Overall, the crudeness

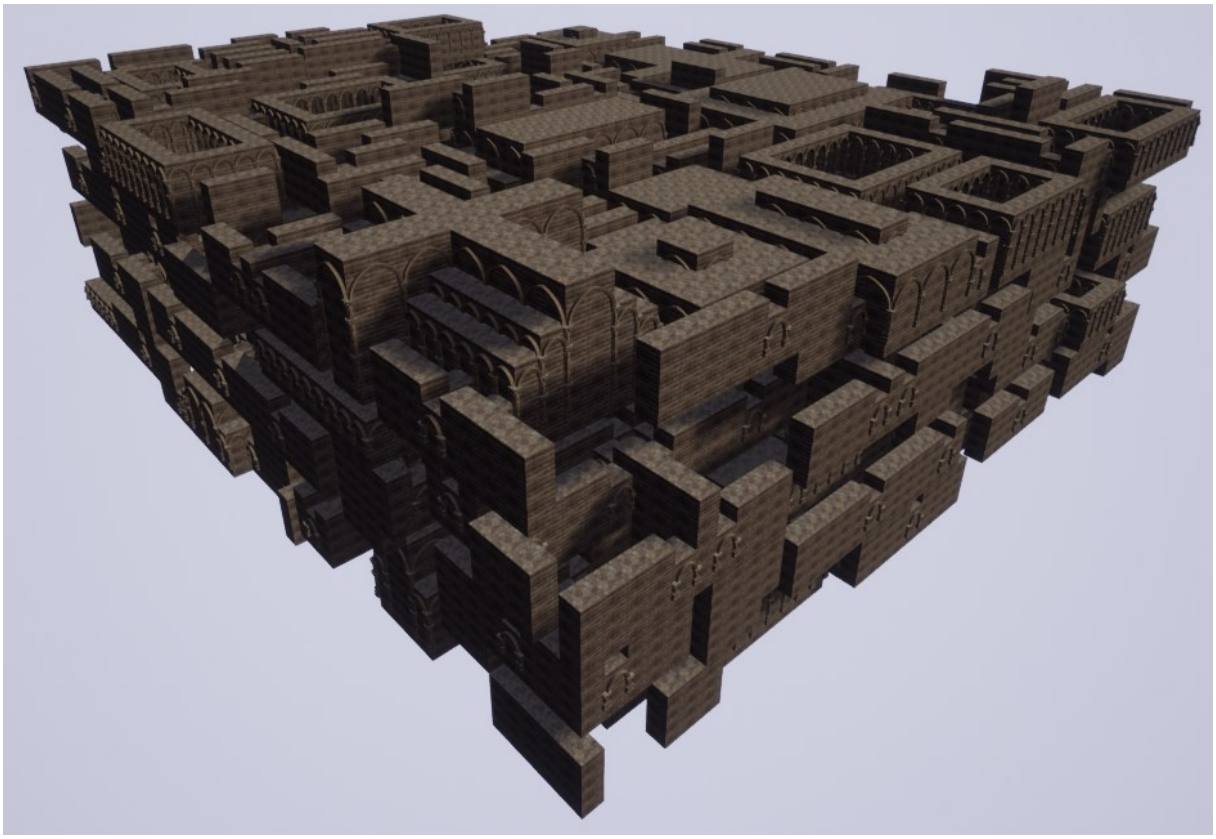


Figure 5-10. Level with a high score (0.37) on the *Fill Cube* test case (generated by our RDFS).

of the RDFS algorithm is apparent. It is an effective algorithm when the goal is simply to find any valid way to add rooms to a level, but struggles to fulfil more specific requirements.

The Monte Carlo tree search (MCTS) algorithm does not perform notably well on any case. We are forced to conclude that MCTS is not a good choice of algorithm for level layout, at least when using a tree extension approach. Our motivation for testing MCTS was that rooms placed early in the level layout process can have a long-term effect on the final outcome. MCTS spends its searching effort testing all possible branches high in the valid layout tree, while only sparsely sampling deeper nodes. We hoped that MCTS would help overcome the limitations of RDFS (which extensively searches deep branches of the tree) by avoiding bad room placements early in the room layout process. However, the results suggest that particular room placements early in the process are not good or bad in the same way that a Go move can be good or bad. It seems likely that a particular room placement is only good or bad in the context of other rooms placed much later, so the MCTS is not able to gain useful information by sampling random levels using particular rooms. The case on which MCTS performed the best, the *Long Linear*

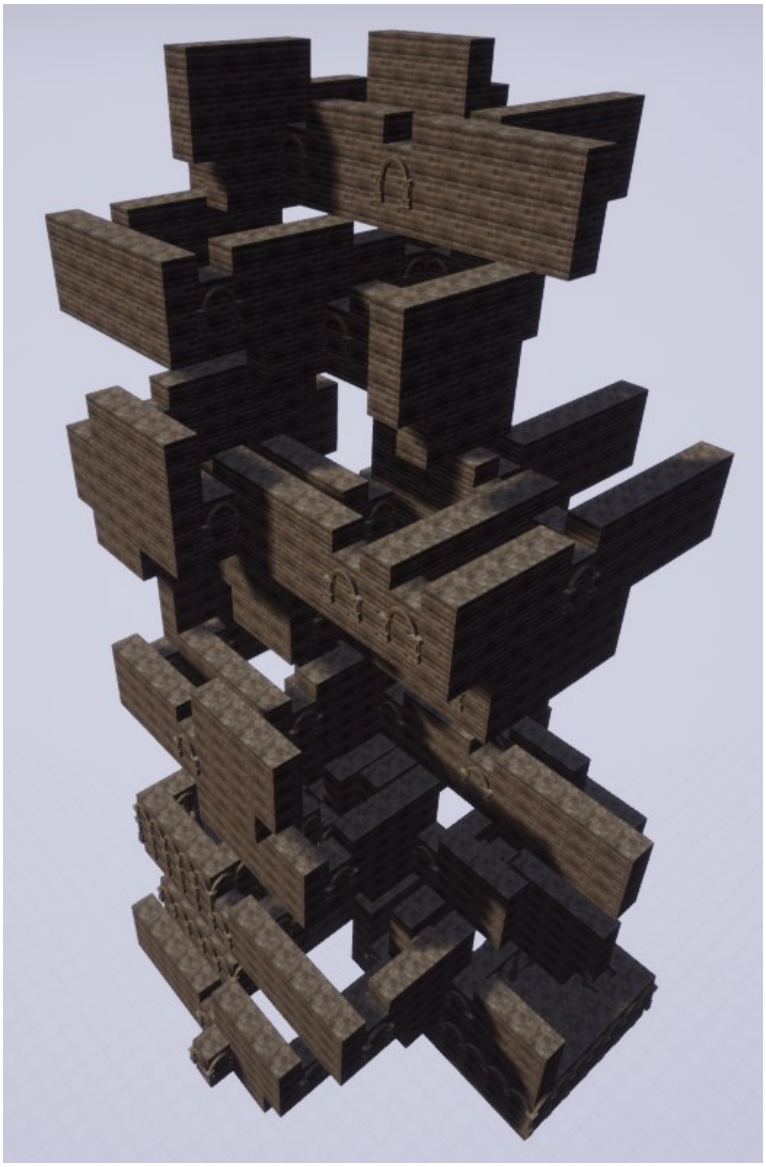


Figure 5-11. Level with a perfect score (1.0) on the *Tower* test case (generated by our EA).

case, may be an exception to this rule. In this case, a room placement that creates a junction in the level rather than extending the longest linear path is obviously bad.

Our evolutionary algorithm (EA) performed the best or close to the best on every case except *Fill Cube*. It is evidently both versatile and effective, since it is able to find high-scoring levels on many different fitness functions. There are several likely reasons for its strong performance. One is its use of crossover operations. Once the EA happens to assemble a useful pattern of rooms, it can duplicate the same pattern many times throughout a level (by making multiple similar copies of the level in the population, and then combining them through

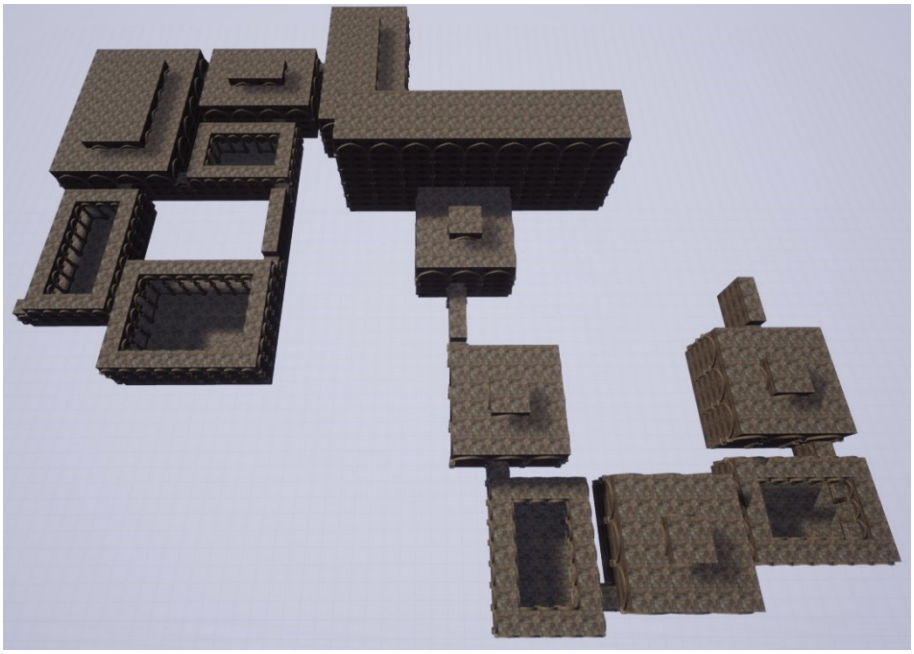


Figure 5-12. Level with a high score (0.84) on the *Long Linear* test case (generated by our EA).

crossover). This behaviour can be observed especially clearly on the *Tower* test case (see Figure 5-11). Typically, the EA will first evolve an upward corkscrew of rooms, then duplicate the corkscrew repeatedly until it reaches the top of the region volume. It may be noted that, although this behaviour allows the EA to score well on certain fitness functions, it is in some ways undesirable because it results in many identical rooms throughout a level.

The other main advantage of our EA is its use of a tree genotype. The tree genotype means that mutation operations can add or remove rooms anywhere in a level, without disrupting the remainder of the level structure. Thus, when working on a particular design, the search does not become trapped by early decisions about room placement, but can amend them if they later become problematic. This allows the EA to home in on good designs much more efficiently than either of the other two algorithms. It is able to remove rooms when necessary to create levels with specific topological features like those required in the *LongLinear*, *BranchingPaths*, and *ComplexMaze* test cases (see Figure 5-13, Figure 5-12, and Figure 5-14).

Observations of our EA suggest that it normally converges quickly on a single solution, which is then improved iteratively over many generations. That is, its population of levels consists of fifty variants on the current best solution, not a diverse variety of different level

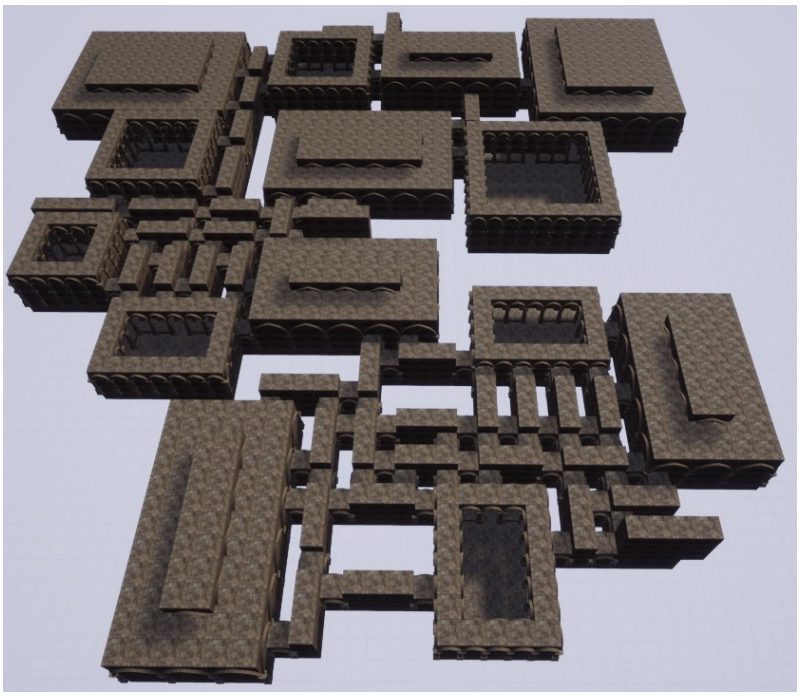


Figure 5-13. Level with a high score (0.88) on the *Branching Paths* test case (generated by our EA).

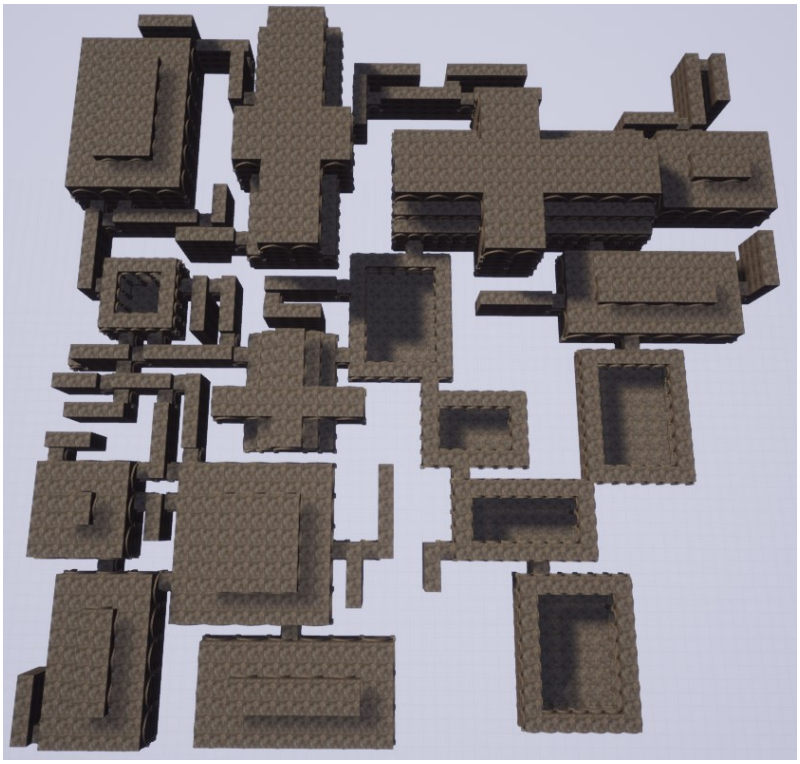


Figure 5-14. Level with a high score (0.98) on the *Complex Maze* test case (generated by our EA).

designs. Crossover operations appear to virtually always result in introducing a new copy of a structure existing elsewhere in the same level, not introducing a drastically new idea. Given this observation, it is possible that many of the benefits of the EA could be obtained using a simpler hill-climbing algorithm with a similar tree genotype and mutation operations. It is also possible that the performance of the EA could be further improved by modifying the algorithm to promote greater diversity in the population. For example, an island model [Whitley 1994] could be used to let different solutions develop separately.

The results suggest that, of the three algorithms, an evolutionary algorithm with a tree genotype should be the preferred method for search-based level layout in almost all situations. The other two algorithms we tested both rely on a linear representation of a level as a sequence of instructions to the room placement machine. It seems that such a linear representation is too restrictive to effectively search the space of tree-like level designs. It is possible that a linear representation might be more useful for building linear levels in particular, if the room placement machine were modified such that new rooms were always connected to the most recently added room. This would allow the level topology to correspond more closely with the level representation.

5.3 Fitness Functions

In the previous section, we evaluated our search algorithms using fitness functions. In this section, we will evaluate the quality of the best-scoring level layouts our system produces. It is one thing to say that a level scores highly on a mathematical fitness function, but is the level actually well designed? The evaluation of high-scoring levels can equivalently be seen as an evaluation of the fitness functions themselves. A fitness function for levels is good if the levels that score highly on the function are in fact good levels.

Naturally, what makes a level "good" is something of a subjective question, and depends on the requirements of a specific game or situation. The features supported by our fitness function were chosen to reward various properties that may be desirable in a game level. For the most part, these features accomplish what they were designed to do. Unfortunately, the desired properties sometimes come with unintended side effects. In some cases, features interfere with one another, making it difficult to produce levels with particular combinations of traits. In other

cases, producing a desired result requires combining multiple different features in an unintuitive way, which makes the system difficult to control.

The *LinearRooms* feature, which rewards levels based on the percentage of rooms with exactly two connections, was intended to produce levels with an essentially linear layout. However, it can sometimes also inadvertently reward the creation of long dead-end paths. Only the last room in such a path is a dead end, so every room added increases the fraction of rooms in the level that are linear. By the same logic, the *DeadEndRooms* feature, which rewards rooms with a single connection and was intended to produce levels with dead-end paths, usually results in dead ends consisting of a single room. The *ComplexMaze* level shown in Figure 5-14 provides an example of this behaviour. If the goal is to reward levels with many long dead-end paths, it is necessary to include both *LinearRooms* and *DeadEndRooms* in the fitness function.

The unintuitive behaviour of the *LinearRooms* and *DeadEndRooms* features was what prompted us to add the *RoomsOnAnyPath* feature to our fitness function. This feature is more effective than *LinearRooms* at removing dead-end paths from a level because it penalizes every room in such a path, not only the last room. However, it is more expensive to calculate because it requires multiple depth-first searches to test which of the rooms in a level lie on a path to the end room.

The *AverageConnectors* function was intended to reward highly connected levels, and does so. However, it often produces tight knots of many interconnected hallways in a small area, since this is an easy way to raise the average number of connectors per room. Examples of this undesirable phenomenon can be seen in the *BranchingPaths* level in Figure 5-13. The *JunctionRooms* feature is less likely to produce such tight knots because it does not reward individual rooms for having more than three connectors.

The *LevelLength* feature correctly rewards long, twisty levels. It also implicitly rewards levels with a linear structure, because loops can short-circuit a level's length. Disappointingly, none of our search algorithms are able to reliably score well on this feature. The problem seems to be that although *LevelLength* rewards long levels when they happen to arise, it does not effectively guide generation to produce them. To make matters worse, the *CheckpointProximity* feature interferes with the *LevelLength* feature by encouraging the search algorithm to build a straight path towards the end point. This interaction is problematic because creating a long, twisty path to a specific destination it is a natural level design goal.

VolumeFill and *RoomCount* are the simplest features our fitness function supports. They serve a similar goal of trying to pack a lot of playable level into a small space. In theory, we would expect *VolumeFill* to implicitly reward large rooms and *RoomCount* to reward small rooms, but these effects are not very noticeable. Both these features interfere with other features such as *LinearRooms*, *RoomsOnAnyPath*, and *LevelLength*, because adding rooms wherever they fit tends to create both dead ends and incidental connections.

Fitness functions based on a single feature tend to produce extreme and unappealing designs, like the *FillCube* level in Figure 5-10 and the *Tower* level in Figure 5-11. However, this is not a major concern because it is intended that multiple features of the fitness function should be used simultaneously.

Certain design flaws appear in many of our generated levels that no current feature of our fitness function helps to address. One problem is that many levels feature hallways or stairways as dead ends. Many examples of this problem can be seen in Figure 5-14. Although game levels need not be laid out the same as real buildings, hallways or stairways to nowhere are so unrealistic that they are likely to draw attention to themselves and threaten the player's suspension of disbelief. The cause of this problem is that our fitness function does not distinguish between hallways or stairways, which should not be used as dead ends, and other types of room, which should be.

Another problem is that a room which is not considered a dead end by the fitness function may still appear to be a dead end to the player. This can happen when all the connections to a room are placed close together at the same end of the room. The result may again be a stairway to nowhere, if the system places a stairway room with multiple connections at the bottom and none at the top. This problem stems from the way we reduce rooms of a level to nodes in a graph when analyzing levels. Reducing a level to a graph expedites analysis, but conceals the difference between different parts of a room.

A related problem involves rooms with multiple disconnected areas, such as our multi-level rooms with balconies. The balconies in such a room are not connected to the main floor or to one another, requiring a stairway in another room to join them. Such a design can be desirable for gameplay in order to teases the player with currently inaccessible locations, or to provide sniper shots for ranged enemies. Unfortunately, our system does not distinguish between multiple areas of a single room. As a result, some layers of a multi-level room might not be

connected to the rest of the level. Worse yet, the system may place connectors on different layers and treat them as connected. This can result in impossible levels, where the player cannot reach the next room without magically flying up to a balcony. We disabled multi-level rooms in our grammar to prevent this problem from occurring. To fully address this problem and allow multi-level rooms would require some method of distinguishing between disconnected areas of a room.

The levels generated by our system are not appealing or realistic when seen from the outside. This is not a problem for levels such as underground dungeons, or in any scenario where the level will only be seen from the inside. However, can be problematic for levels that contain a mix of interior and exterior spaces. Our cloister rooms exemplify this problem. When standing in an open cloister, it is sometimes possible to see other rooms of the level floating in space overhead. The only way to avoid this problem in our current system is to disable rooms with open ceilings when generating levels with multiple storeys.

Generated levels sometimes contain multiple identical rooms or even identical sequences of rooms. This usually happens as a result of the crossover operator used by our evolutionary algorithm, but can also happen by coincidence if the same axiom is selected to generate multiple rooms. Multiple identical rooms are undesirable because they may be boring or confusing for the player. Our current fitness function does not penalize identical rooms because it treats rooms as interchangeable nodes in a graph. It would be beneficial if the fitness function distinguished between rooms of different sizes and types.

6 Conclusions

We have presented a system for generating game levels with detailed interior architecture. We generate rooms based on structural rules defined by a shape grammar, and arrange them into levels based on a fitness function that can be customized to meet the needs of a specific scenario. Our evolutionary algorithm is able to successfully generate levels with a variety of different properties, depending on the fitness function and region configuration. This makes our system highly controllable. Unlike previous level generation systems, we generate levels in three dimensions, using modular meshes like those used in high-budget games. Our finished levels can be loaded in *Unreal Engine 4*, a widely used game engine.

6.1 Future Work

The long-term goal of our work is to create a level generation tool that will be useful for game developers. In this thesis, we addressed many of the technical problems of 3D level generation with interior architecture. We developed and compared several algorithms. However, another extremely important consideration for a good PCG tool is its user interface. Our current level generator is a technical prototype and is not designed for usability by non-experts. Before the generator can be run, the designer must specify grammar rules and mesh placement rules in C++ code, and other parameters in JSON configuration files. Removing these barriers to usability would be the most important step required to turn our work into a valuable development tool.

A promising approach to improve the generator's usability would be to re-implement the generator as a plug-in for a game engine such as *Unreal Engine 4*, rather than as a separate program. This would allow the generator to be used more easily alongside other level design tools. A graphical user interface with sliders and buttons could be used to configure the fitness function and other features of the generator such as the placement of incidental connections.

More importantly, a graphical method of specifying grammar rules and mesh placement rules would make it significantly easier to apply our system to multiple mesh kits and architectural styles. The interface could be based on existing systems for exterior architecture generation [Lipp 2008; Golding 2010]. Once a user interface had been developed, user testing and collaboration with expert designers would be extremely valuable to ensure that the system met the requirements of professional game development.

We found our evolutionary algorithm to be effective for level layout, but it is likely that future work could improve upon it further. Additional mutation types could be added, such as inserting a room between an existing room and its parent, or replacing a room with a new room while preserving its children. These mutations would allow the algorithm to sample potentially promising areas of the search space that the current mutations do not. As we suggested in Section 5.2, an island model could be tested to increase the diversity of the level population during evolution, or a simpler hill-climbing algorithm could be tested to see if it would produce similar results with faster performance.

In Section 5.3, we described several weaknesses of our current method of evaluating level fitness. The most important problems with our current system are that it does not distinguish between different types of rooms and that it does not distinguish between different areas in the same room. Addressing these problems would allow us to avoid unrealistic designs such as stairways to nowhere, and to ensure level connectivity while incorporating 3D features such as balconies. Aside from these major concerns, there are many other features that could be added to the fitness function to improve the quality of generated levels and to provide additional control to the human designer. Numerous features employed in previous work on the evolution of 2D levels [Togelius 2010b; Ashlock 2011a; Valtchanov 2012] could be applied to 3D levels.

Our system does not address all stages of level design. For example, we do not address the placement of game objects such as enemies and pickups, or the placement of lights and sounds. We assume these tasks will be handled by the human designer. However, it is also possible that these tasks could be automated by adding new modules to the level generator.

The levels generated by our system sometimes have a monotonous appearance due to the repeated use of the same meshes. In real game levels, this concern is often addressed by using variant meshes, such as walls made of different materials, or floors with different grates in them. Our mesh placement module could be improved by making it randomly or systematically

Figure 6-1. Lights and variant meshes significantly improve the appearance of a level.

alternate between variant meshes for the same purpose. Shape attributes could be used to provide a consistent style within a single room, as they are for buildings in the work of Wonka et al. [Wonka 2003]. Figure 6-1 shows the effect of manually adding lights and variant meshes to a level built with the *Soul: City* mesh kit [Epic 2014]. Aside from these additions, the hallway shown is the same as those generated by our system.

One of the limitations of our system is its strict adherence to a rectangular grid. Most manmade architectural spaces do in fact have a basically rectangular design. Nonetheless, generating levels without a grid, or with a non-rectangular grid (such as a hexagonal grid), would be a worthwhile expansion of the present work. A possible method to add versatility while maintaining the benefits of using a grid is suggested by the "snap to reference" system used by Bethesda to develop games like *Skyrim* [Burgess 2013]. In this system, each section of the level is snapped to a rectangular grid, but the grids for different sections of the level can be rotated differently. The challenge of such an approach is that non-standard meshes are required to bridge the wedge-shaped gaps between differently rotated sections.

Bibliography

[Adams 2002] David Adams. 2002. *Automatic Generation of Dungeons for Computer Games.* Bachelor thesis. University of Sheffield, UK.

[Alexander 1977] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York.

[Amato 2017] Alba Amato. 2017. Procedural Content Generation in the Game Industry. *Game Dynamics*, Springer, Cham, 2017. 15-25.

[Ashlock 2010] Daniel Ashlock. 2010. Automatic Generation of Game Elements via Evolution. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, Dublin, 2010, 289-296.

[Ashlock 2011a] Daniel Ashlock, Colin Lee, and Cameron McGuinness. 2011. Search-Based Procedural Generation of Maze-Like Levels. In *IEEE Transactions on Computational Intelligence and AI in Games*, *3*, 3, 260-273.

[Ashlock 2011b] Daniel Ashlock, Colin Lee, and Cameron McGuinness. 2011. Simultaneous Dual Level Creation for Games. In *IEEE Computational Intelligence Magazine*, *6*, 2, 26-37.

[Baldwin 2017] Alexander Baldwin and Johan Holmberg. 2017. *Mixed-Initiative Procedural Generation of Dungeons Using Game Design Patterns*. Master thesis project. Malmö University, Sweden.

[Baron 2017] Jessica R. Baron. 2017. Procedural Dungeon Generation Analysis and Adaptation. In *Proceedings of the South-East Conference* (*ACM SE '17*). ACM, 168-171.

[Barrett 2007] Sean Barrett. 2009. "L-Systems Considered Harmful." Forum post preserved on author's website. *nothings.org/gamedev/l_systems.html*. Accessed: December 13, 2018.

[Benros 2012] Deborah Benrós, José P. Duarte, and Sean Hanna. 2012. A New Palladian Shape Grammar: A Subdivision Grammar as Alternative to the Palladian Grammar. In *International Journal of Architectural Computing*, *10*, 4, 521-540.

[Beyer 2017] Theobald Beyer. 2017. *Story Guided Procedural Generation of Complex Connected Worlds and Levels for Role Play Games.* Master's thesis. Technical University of Munich, Germany.

[Bhojan 2014] Anand Bhojan and Hong Wei Wong. 2014. ARENA: Dynamic Run-Time Map Generation for Multiplayer Shooters. In *International Conference on Entertainment Computing*, Springer, Berlin, Heidelberg, 149-158.

[Boyd 2011] Robert Boyd. Oct. 10, 2011. "9 Things We can Learn about Game Design from Dark Souls." *Gamasutra*. *www.gamasutra.com/blogs/RobertBoyd/20111010/90386/9_Things_We_can_Learn_about_Game_Design_from_Dark_Souls.php.* Accessed: December 13, 2018.

[Browne 2012] Cameron B. Browne, Edward Powley, Daniel Whitehouse, et al. (2012). A Survey of Monte Carlo Tree Search Methods. In *IEEE Transactions on Computational Intelligence and AI in games*, *4*, 1, 1-43.

[Burgess 2013] Joel Burgess. 2013. Skyrim's Modular Level Design. Transcript of presentation at *Game Developers Conference 2013*, San Francisco, CA. *blog.joelburgess.com/2013/04/skyrims-modular-level-design-gdc-2013.html*. Accessed: December 13, 2018.

[Caldwell 2017] Brendan Caldwell. Apr. 24, 2017. "In their haste to make 'soulslikes', devs have forgotten what makes Dark Souls unique—it's level design." *Rock, Paper, Shotgun*. *www.rockpapershotgun.com/2017/04/24/in-their-haste-to-make-soulslikes-devs-have-forgotten-what-makes-dark-souls-unique-its-level-design.* Accessed: December 13, 2018.

[Cardamone 2011] Luigi Cardamone, Georgios N. Yannakakis, Julian Togelius, and Pier Luca Lanzi. 2011. Evolving Interesting Maps for a First Person Shooter. In *European Conference on the Applications of Evolutionary Computation*, Springer, Berlin, Heidelberg, 63-72.

[Co 2006] Phil Co. 2006. *Level Design for* Games*: Creating Compelling Game Experiences*. New Riders, Berkeley, CA.

[Compton 2006] Kate Compton and Michael Mateas. 2006. Procedural Level Design for Platform Games. In *AIIDE*, 109-111.

[Compton 2017] Kate Compton and Michael Mateas. 2017. A Generative Framework of Generativity. In *Experimental AI in Games Workshop*, *AIIDE*.

[Coulom 2006] Rémi Coulom. 2006. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *International Conference on Computers and Games*, Springer, Berlin, Heidelberg, 72-83.

[Dahlskog 2015] Steve Dahlskog, Staffen Björk, Julian Togelius. 2015. Patterns, Dungeons, and Generators. In *Foundations of Digital Games (FDG)*.

[Dormans 2010] Joris Dormans. 2010. Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, ACM.

[Dormans 2011] Joris Dormans and Sander Bakkes. 2011. Generating Missions and Spaces for Adaptable Play Experiences. *IEEE Transactions on Computational Intelligence and AI in Games*, *3*, 3, 216-228.

[Elinder 2017] Tobias Elinder. 2017. General Methods for the Generation of Seamless Procedural Cities. Master's thesis. Lund University, Lund, Sweden.

[Epic 2014] Epic Games. 2014. *Soul: City*. Mesh kit for *Unreal Engine 4*. *www.unrealengine.com/marketplace/soul-city.* Accessed: December 13, 2018.

[Britannica] *Encyclopaedia Britannica*. "Cloister." *www.britannica.com/topic/cloister*. Accessed: December 16, 2018.

[Fahmy 1992] Hoda Fahmy and Dorothea Blostein. 1992. A Survey of Graph Grammars: Theory and Applications. 1992. In *Pattern Recognition, 1992, Vol. II Conference B: Pattern Recognition Methodology and Systems, Proceedings, 11th IAPR International Conference on*, IEEE, 294-298.

[Ferrari 2013] Mattia Ferrari. 2013. Automatic Generation of Maps in *In Verbis Virtus*. Technical report. Polytechnic University of Milan, Italy.

[Foerster 2017] Sebastian Foerster. May 2, 2017. "Dark Souls: A Masterclass in Game Design." *Obilisk*. *www.obilisk.co/dark-souls-masterclass-game-design*. Accessed: December 13, 2018.

[Ford 2014] Ian Ford. Nov. 28, 2014 "Dragon Age: Inquisition – Mike Laidlaw on the creative process, part one." *The Guardian*. w*ww.theguardian.com/technology/2014/nov/28/dragon-age-inquisition-mike-laidlaw-creative-process*. Accessed: December 13, 2018.

[Font 2015] Jose M. Font, Roberto Izquierdo, Daniel Manrique, Julian Togelius. 2016. Constrained Level Generation through Grammar-Based Evolutionary Algorithms. In *European Conference on the Applications of Evolutionary Computation*, Springer, Cham, 558-573.

[GLM] *OpenGL Mathematics*. 2005-2018. GitHub repository. *github.com/g-truc/glm*

[Golding 2010] James Golding. 2010. "Building Blocks: Artist Driven Procedural Buildings." Slides from presentation at *Game Developers Conference 2010*, San Francisco, CA. *www.gdcvault.com/play/1012655/Building-Blocks-Artist-Driven-Procedural*. Accessed: November 28, 2018.

[Hahn 2006] Evan Hahn, Prosenjit Bose, and Anthony Whitehead. 2006. Persistent Realtime Building Interior Generation. In *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, 179-186.

[Horswill 2012] Ian Douglas Horswill and Leif Foged. 2012. Fast Procedural Level Population with Playability Constraints. In *AIIDE*.

[HowLongToBeat] *HowLongToBeat.* "Assassin's Creed Origins." *howlongtobeat.com/game.php?id=46402*. Accessed: December 15, 2018.

[Hullet 2012] Kenneth Hullet and Jim Whitehead. 2012. Design Patterns in FPS Levels. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games (FDG)*, ACM, 78-85.

[Johnson 2010] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. 2010. Cellular Automata for Real-Time Generation of Infinite Cave Levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, ACM, 2010.

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

[Karavolos 2016] Daniel Karavolos, Antonios Liapis, and Georgios N. Yannakakis. 2016. Evolving Missions to Create Game Spaces. In *Computational Intelligence and Games,* IEEE, 1-8.

[Kerssemakers 2012] Manuel Kerssemakers, Jeppe Tuxen, Julian Togelius, and Georgios N. Yannakakis. 2012. A Procedural Procedural Level Generator Generator. In *Computational Intelligence and Games*, IEEE, 335-341.

[Knuth 1968] Donald E. Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, no. 2: 127-145.

[Koens 2015] Erik Koens. 2015. *Generating non-monotone 2D platform levels and predicting difficulty*. Master's thesis. Utrecht University, Utrecht, the Netherlands.

[Koster 2005] Raph Koster. 2005. *A Theory of Fun in Game Design*. Paraglyph Press, Scottsdale, Arizona.

[Koster 2018] Raph Koster. Jan. 17, 2018. The cost of games. *Gamasutra*. *www.gamasutra.com/blogs/RaphKoster/20180117/313211/The_cost_of_games.php.* Accessed: December 14, 2018.

[Lavender 2015] Becky Lavender. 2015. *The Zelda Dungeon Generator: Adopting Generative Grammars to Create Levels for Action-Adventure Games*. Bachelors thesis. University of Derby, England.

[Liapis 2013] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. 2013. Sentient Sketchbook: Computer-aided game level authoring. In *Foundations of Digital Games*, 213-220.

[Lifschitz 2008] Vladimir Lifschitz. 2008. What Is Answer Set Programming? In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, 1594-1597.

[van der Linden 2013] Roland Van der Linden. 2013. *Designing Procedurally Generated Levels*. Master's thesis. Delft University of Technology, Delft, the Netherlands.

[van der Linden 2014] Roland van der Linden, Ricardo Lopes, and Rafael Bidarra. 2014. Procedural Generation of Dungeons. In *IEEE Transactions on Computational Intelligence and AI in Games 6*, 1, 78-89.

[Lipp 2008] Markus Lipp, Peter Wonka, and Michael Wimmer. 2008. Interactive Visual Editing of Grammars for Procedural Architecture." In *ACM Transactions on Graphics (TOG) 27*, 3: 102.

[Lohmann] Niels Lohmann. 2013-2018. *JSON for Modern C++ version 3.4.0.* GitHub repository. *github.com/nlohmann/json*

[Lorensen 1987] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm." In *ACM SIGGRAPH Computer Graphics*, *21*, 4, 163-169.

[Mark 2015] Benjamin Mark, Tudor Berechet, Tobias Mahlmann, and Julian Togelius. 2015. Procedural Generation of 3D Caves for Games on the GPU. In *Foundations of Digital Games (FDG)*.

[Mailberg 2016] Emanuel Mailberg. Aug. 21, 2016. "'No Man's Sky' Is Like 18 Quintillion Bowls of Oatmeal." *Motherboard*, Vice. *motherboard.vice.com/en_us/article/nz7d8q/no-mans-sky-review*. Accessed December 13, 2018.

[Mana 2015] Mana Station. 2015. *Multistory Dungeons*. Mesh kit for *Unreal Engine 4*. *www.unrealengine.com/marketplace/top-down-multistory-dungeons.* Accessed: December 13, 2018.

[Martin 2006a] Jess Martin. 2005. Algorithmic Beauty of Buildings: Methods for Procedural Building Generation. Bachelor thesis. Trinity University, San Antonio, Texas.

[Martin 2006b] Jess Martin. 2006. Procedural House Generation: A Method for Dynamically Generating Floor Plans. In *Symposium on interactive 3D Graphics and Games*, *2*.

[Mawhorter 2010] Peter Mawhorter and Michael Mateas. 2010. Procedural Level Generation Using Occupancy-Regulated Extension." In *Computational Intelligence and Games (CIG)*, IEEE, 351-358.

[McGuiness 2011a] Cameron McGuinness and Daniel Ashlock. 2011. Decomposing the Level Generation Problem with Tiles." In *Congress on Evolutionary Computation (CEC),* IEEE, 849-856.

[McGuiness 2011b] Cameron McGuinness and Daniel Ashlock. 2011. Incorporating Required Structure into Tiles. In *Computational Intelligence and Games (CIG),* IEEE, 16-23.

[Merrell 2010] Paul Merrell, Eric Schkufza, and Vladlen Koltun. 2010. Computer-Generated Residential Building Layouts. In *Transactions on Graphics (TOG)*, *29*, 6, ACM, 181.

[Models] *The Models Resource*. "Skurge Challenge." *www.models-resource.com/pc_computer/harrypotterthechamberofsecrets/model/13457.* Accessed: December 17, 2018.

[Mourato 2011] Fausto Mourato, Manuel Próspero dos Santos, and Fernando Birra. 2011. Automatic Level Generation for Platform Videogames using Genetic Algorithms. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*, ACM, 8.

[Müller 2006] Pascal Müller, Peter Wonka, Simon Haegler, et al. 2006. Procedural Modeling of Buildings. In *Transactions On Graphics (TOG)*, *25*, 3, ACM, 614-623.

[Müller 2007] Pascal Müller, Gang Zeng, Peter Wonka, and Luc Van Gool. 2007. Image-Based Procedural Modeling of Facades. In *Transactions on Graphics (TOG)*, *26*, 3, ACM, 85.

[Parish 2001] Yoav I. H. Parish, and Pascal Müller. 2001. Procedural Modeling of Cities. In *Proceedings of the 28th Annual Conference on Computer graphics and Interactive Techniques*, ACM, 301-308.

[Perry 2002] Lee Perry. 2002. Modular Level and Component Design. *Game Developer Magazine*, Nov. 2002, 30-35. *www.gdcvalut.com/gdmag*. Accessed: November 28, 2018.

[Piranesi 1745] Giovanni Battista Piranesi. C. 1745-1750. *Le Carceri d'Invenzione*, *plate XIV, The Gothic Arch*, first edition. Etching. In *Wikimedia Commons*, courtesy of Leiden University. *commons.wikimedia.org/wiki/File:Giovanni_Battista_Piranesi_-_Le_Carceri_d%27 Invenzione_-_First_Edition_-_1750_-_14_-_The_Gothic_Arch.jpg.* Accessed: December 15, 2018.

[Piranesi 1750] Giovanni Battista Piranesi. C. 1750. *Carcere oscura con Antenna pel suplizio dè malfatori*. Etching. In *Wikimedia Commons*, courtesy of Metropolitan Museum of Art. *commons.wikimedia.org/wiki/File:Dark_prison_with_a_courtyard_for_the_punishment_of_criminals_._._.(Carcere_oscura_con_Antenna_pel_suplizio_d%C3%A8_malfatori_._._)_MET_DP827984.jpg*. Accessed: December 15, 2018.

[Prusinkiewicz 2012] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. 2012. *The Algorithmic Beauty of Plants*. Springer Science & Business Media. Online book. *algorithmicbotany.org/papers/#abop*. Accessed: December 13, 2018.

[Shaker 2011] Noor Shaker, Julian Togelius, Georgios N. Yannakakis, et al. 2011. The 2010 Mario AI Championship: Level Generation Track." In *IEEE Transactions on Computational Intelligence and AI in Games 3*, 4, 332-347.

[Shaker 2015] Julian Togelius, Noor Shaker, and Mark J. Nelson. 2015. Constructive Generation Methods for Dungeons and Levels. In *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, Springer. Online book. *pcgbook.com*. Accessed: December 13, 2018.

[Silver 2016] David Silver, Aja Huang, Chris J. Maddison, et al. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search." *Nature 529*, 7587: 484-489.

[Smith 2009] Gillian Smith, Mike Treanor, Jim Whitehead, and Michael Mateas. 2009. Rhythm-based level generation for 2D platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games (FDG)*, ACM, 175-182.

[Smith 2011a] Gillian Smith, Jim Whitehead, Michael Mateas, et al. 2011. Launchpad: A Rhythm-Based Level Generator for 2-D Platformers." *IEEE Transactions on Computational Intelligence and AI in Games 3*, 1, 1-16.

[Smith 2011b] Adam M. Smith and Michael Mateas. 2011. Answer Set Programming for Procedural Content Generation: A Design Space Approach. *IEEE Transactions on Computational Intelligence and AI in Games 3*, 3, 187-200.

[Smith 2014] Anthony J. Smith and Joanna J. Bryson. 2014. A Logical Approach to Building Dungeons: Answer Set Programming for Hierarchical Procedural Content Generation in Roguelike Games." In *Proceedings of the 50th Anniversary Convention of the AISB*.

[Smith 2015] Gillian Smith. 2015. An Analog History of Procedural Content Generation. In *Foundations of Digital Games (FDG)*.

[Sorenson 2010] Nathan Sorenson. 2008. *The Evolution of Fun: A Generic Model of Video Game Challenge for Automatic Level Design*. Master's thesis. Simon Fraser University, Burnaby, British Columbia.

[Stiny 1971] George Stiny and James Gips. 1971. Shape Grammars and the Generative Specification of Painting and Sculpture. In *The Best Computer Papers of 1971*, Auerbach, Philadelphia, 125-135.

[Stiny 1978] George Stiny and William J. Mitchell. 1978. The Palladian Grammar. *Environment and planning B: Planning and design 5*, 1, 5-18.

[Summerville 2015] Adam J. Summerville, Morteza Behrooz, Michael Mateas, and Arnav Jhala. 2015. "The Learning of Zelda: Data-Driven Learning of Level Topology." In *Proceedings of the FDG workshop on Procedural Content Generation in Games*.

[Tippetts 2011] Joshua Tippetts. 2011. Creator of worlds: procedural terrain generation in a sandbox environment. *Game Developer Magazine*, April 2011, 21-27. *www.gdcvalut.com/gdmag*. Accessed: November 28, 2018.

[Togelius 2010a] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. 2010. Search-Based Procedural Content Generation. In *European Conference on the Applications of Evolutionary Computation*, Springer, Berlin, Heidelberg, 141-150.

[Togelius 2010b] Julian Togelius, Mike Preuss, and Georgios N. Yannakakis. 2010. "Towards multiobjective procedural map generation." *Proceedings of the 2010 workshop on Procedural Content Generation in Games*, ACM.

[Togelius 2013] Julian Togelius, Alex J. Champandard, Pier Luca Lanzi, et al. 2013. Procedural Content Generation: Goals, Challenges and Actionable Steps. In *Dagstuhl Follow-Ups*, *6*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[Valtchanov 2012] Valtchan Valtchanov and Joseph Alexander Brown. 2012. Evolving Dungeon Crawler Levels with Relative Placement." In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, ACM, 27-35.

[VGMaps] "A Link to the Past: Ganon's Tower: 3F." *VGMaps.com*, Jonathan Leung. *http://www.vgmaps.com/Atlas/SuperNES*. Accessed: December 12, 2018.

[Whitley 1994] Darrell Whitley. 1994. A Genetic Algorithm Tutorial. In *Statistics and Computing*, *4*, 2, 65-85.

[Wonka 2003] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. 2003. Instant Architecture. In *ACM Transactions on Graphics*, *22*, 4. 669-677.

[Yu 2011] Derek Yu. 2011. The Full Spelunky on Spelunky. Slides from presentation at *Game Developers Conference 2011*, San Francisco, California. *makegames.tumblr.com/post/4061040007/the-full-spelunky-on-spelunky*. Accessed: December 13, 2018.

Ludography

Age of Empires. Ensemble Studios. Microsoft Games, 1997. Microsoft Windows.

Angband. Angband Development Team, originally by Alex Cutler and Andy Astrand. Open source, first release 1990. Unix, Windows, Mac OS. *rephial.org*.

Assassin's Creed Origins. Ubisoft Montreal. Ubisoft, 2017. Microsoft Windows, PlayStation 4, Xbox One.

Civilization. MPS Labs. MicroProse, 1991. MS-DOS and other platforms.

Crash Bandicoot. Naughty Dog. Sony Computer Entertainment, 1996. PlayStation.

Cube 2: Sauerbraten. Wouter van Oortmersse, Lee Salzman, Mike Dysart. Open source, first release 2004. Microsoft Windows, Linux, FreeBSD, OS X, Unix. *sauerbraten.org*.

Dark Souls. From Software. From Software and Namco Bandai Games, 2011. PlayStation 3, Xbox 360, Microsoft Windows.

Diablo. Blizzard North. Blizzard Entertainment and Electronic Arts, 1996. Microsoft Windows, Classic Mac OS, PlayStation.

Dungeons & Dragons. Gary Gygax and Dave Arneson. Tactical Study Rules, 1974. Tabletop role-playing game.

Dungeon Crawl: Stone Soup. DCSS Devteam, originally by Linley Henzell. Open source, first release 2006. Windows, OS X, Android, Linus, web. *crawl.develz.org*.

Dwarf Fortress. Tarn Adams. Bay 12 Games, first release 2006. Microsoft Windows, Linux, OS X.

Dwarf Quest. Wild Card. Apple iStore, 2013. Mobile (iOS).

The Elder Scrolls V: Skyrim. Bethesda Game Studios. Bethesda Softworks, 2011. Microsoft Windows, PlayStation 3, Xbox 360, PlayStation 4, Xbox One, Nintendo Switch.

Fallout 3. Bethesda Game Studios. Bethesda Softworks, 2008. Microsoft Windows, PlayStation 3, Xbox 360.

Harry Potter and the Chamber of Secrets. KnowWonder. EA Games, 2002. Microsoft Windows.

Infinite Mario Bros. Markus Persson. Open source, original date unknown. *github.com/cflewis/Infinite-Mario-Bros*.

In Verbis Virtus. Indomitus Games. Indomitus Games, 2015. Microsoft Windows.

The Legend of Zelda: A Link to the Past. Nintendo EAD. Nintendo, 1991. Super NES.

The Legend of Zelda: Ocarina of Time. Nintendo EAD. Nintendo, 1998. Nintendo 64, GameCube.

The Legend of Zelda: Twilight Princess. Nintendo EAD. Nintendo, 2006. GameCube, Wii, Shield TV.

Minecraft. Mojang, originally by Markus Perrson. Mojang, 2011. Microsoft Windows, Mac OS, Linux, and many other platforms.

NetHack. The NetHack Dev Team, originally by Mike Stephenson. Open source, first release 1987. Windows, Linux, MacOS. *www.nethack.org*.

No Man's Sky. Hello Games. Hello Games, 2016. PlayStation 4, Microsoft Windows, Xbox One.

Pokémon FireRed and LeafGreen. Game Freak. Nintendo and The Pokémon Company, 2004. Game Boy Advance.

Pokémon Super Mystery Dungeon. Chunsoft. Nintendo and The Pokémon Company, 2015. Nintendo 3DS.

Prince of Persia. Brøderbund. Brøderbund, 1989. Apple II and other platforms.

Rogue. Michael Toy, Glenn Wichman, Ken Arnold. Epyx, 1980. Unix and other platforms.

Sonic the Hedgehog. Sonic Team. Sega, 1991. SegaGenesis and other platforms.

Spelunky. Derek Yu. Mossmouth, LLC, 2008. Microsoft Windows and other platforms.

Star Wars: Dark Forces. LucasArts. LucasArts, 1995. MS-DOS, PlayStation, Microsoft Windows.

Super Mario Bros. Nintendo Creative Department. Nintendo, 1985. Nintendo Entertainment System and other platforms.

Super Mario 64. Nintendo EAD. Nintendo, 1996. Nintendo 64.

Appendix A
Rule Operations

This appendix lists and describes the rule operations supported by our room grammar implementation. Many operations have one or more parameters that can be set. When a rule is listed in a grammar, the values of all the parameters for each of its operations must be specified. The parameter values cannot change during rule execution or depending any properties of on the predecessor shape to which the rule is applied.

When rule operations refer to the size of a shape, they are referring to its size in its local coordinate system. Thus, rotating a room does not affect the way its structure is derived by the room grammar.

Broadly speaking, rule operations can be divided into three categories: those which modify the working state, those which add normal shapes to the output, and those that add connectors to the output. The operations that modify the working state are as follows:

- *Push* – A copy of the projected shape is pushed to the stack.
- *Restore* – The projected shape is replaced with a copy of the shape on top of the stack.
- *PopAndRestore* – The projected shape is replaced with a copy of the shape on top of the stack, which is then popped off the stack.
- *Resize* (*Dimension, Anchor, NewSize*) – The size of the projected shape in the dimension given by *Axis* is changed to *NewSize*. The value of the *Anchor* parameter (*Negative*, *Positive*, or *Centre*) determines the projected shape's position after it is resized, as shown in Figure A-1.

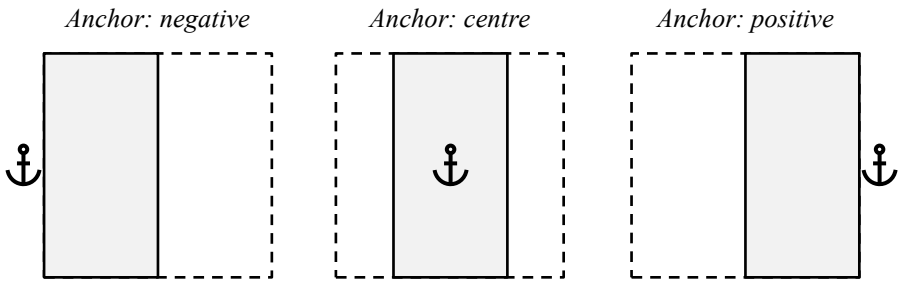


Figure A-1. Effect of anchor when resizing shapes. The dotted line indicates the original size of the shape.

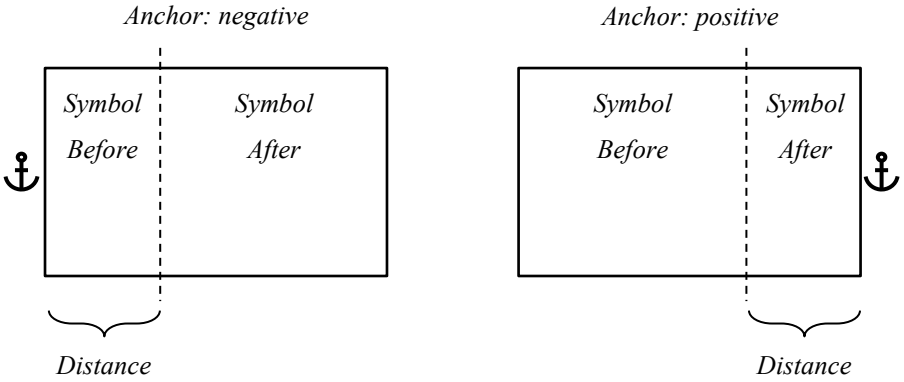


Figure A-2. *SingleSplit* operation.

- *Reduce (Axis, Anchor, Amount)* – As *Resize*, but the new size is found by subtracting *Amount* from the current size of the projected shape. If *Amount* is greater than the current size, the size is reduced to zero.
- *SetAttribute* (*Name*, *Value*) – The attribute named *Name* is set to *Value* on the projected shape. If the projected shape lacks an attribute named *Name*, the attribute is added.
- *ChangeAttribute* (*Name*, *Value*) – As *SetAttribute*, but if the attribute already exists, the new value is found by adding *Value* to the current value.
- *SetAttributeByFunction* (*Name*, *Function*) – As *SetAttribute*, but the new value for the attribute is calculated using a function. *Function* must be a function which takes the current state as its only parameter, returns an integer, and has no side-effects. This operation is used to handle cases where the value of an attribute should depend on another attribute or on a shape's geometric properties.

- *ResizeByAttribute* (*Axis, Anchor, AttributeName*), *ReduceByAttribute* (*Axis, Anchor, AttributeName*) – As *Resize* and *Reduce*, but the value of *NewSize* or *Amount* is given by the appropriate attribute of the projected shape. If the projected shape has no such attribute, a value of zero is used. These operations allow a shape's attributes to affect the size of its descendants.

The operations that place shapes are as follows:

- *PlaceShape* (*Symbol*) – A shape with the symbol *Symbol* is added to the output. The new shape copies all the attributes and geometric properties of the projected shape.
- *SplitSingle* (*Axis*, *Anchor*, *Distance*, *SymbolBefore*, *SymbolAfter*) – Two shapes are added to the output. The two shapes are derived by splitting the projected shape on the specified axis (although the state is not changed). The split point is determined by the values of *Anchor* (*Negative* or *Positive*) and *Distance* (an integer), as shown in Figure A-2. The symbols of the two shapes are given by *SymbolBefore* and *SymbolAfter*.
- *SplitSingleByAttribute* (*Axis, Anchor, AttributeName, SymbolBefore, SymbolAfter*) – As *SplitSingle*, but the value of *Distance* is determined by the appropriate attribute of the projected room.
- *SplitMultiple* (*Axis*, *Symbols*, *Sizes*) – Places multiple shapes by splitting the projected shapes on the specified axis. The *Symbols* parameter is a list of symbols to be used for the placed shapes. The number of sections created is equal to the number of symbols provided. The *Sizes* parameter is a list giving the size of each section. The number of sizes must be equal to the number of symbols. Sizes may be given either in absolute terms (indicated by positive numbers) or in proportionate terms (indicated by negative numbers), or a mix of the two may be used. After deducting the absolute sizes from the size of the projected shape, the remainder is divided among the proportionate sections according to their relative sizes. For example, if sizes of {1, 2, -1, -2} are specified, the first and second sections will each have a size of one. The remaining space will then be split so that the third section gets one third of it and the third section gets the other two thirds.
- *SplitRepeat* (*Axis*, *Anchor*, *SizePerSection*, *Symbol*) – Places multiple shapes by splitting the projected shape on the specified axis. Each shape has a length of *SizePerSection*

along that axis. The sections are measured starting from the end specified by *Anchor* (*Positive* or *Negative*); the last section may be smaller than the others if the total size of the shape is not divisible by *SizePerSection*. All the shapes have the symbol *Symbol*.

We use different operations to place connectors because they need to be placed on the outside of existing shapes, rather than within an existing shape or by subdividing one. The operations for placing connectors are as follows:

- *PlaceConnectors* (*Symbol*, *AnchorY*, *AnchorZ*, *SpacingY*, *SpacingZ*) – Places connectors with the symbol *Symbol* on the positive-*x* face of the projected shape. The size of each connector is determined by the symbol. The first connector to be placed is aligned to the edge or centre of the face in the *y* and *z* dimensions, according to the *AnchorY* and *AnchorZ* parameters. Additional connectors are then placed at intervals based on the *SpacingY* and *SpacingZ* parameters. If both spacing parameters are equal to zero, only a single connector is placed. If one of the spacing parameters is non-zero, a line of evenly spaced connectors is placed in that dimension. If both parameters are non-zero, a grid of evenly spaced connectors is placed. When placing multiple connectors, connectors are only placed up to the edge of the projected shape in each direction.
- *Reorient* (*RotationMatrix*) – Rotates the local coordinate system of the projected shape according to *RotationMatrix*, without rotating the shape itself. This operation can be used to make a different face of the projected shape the positive-*x* side. Thus, connectors can be placed on any face of the projected shape.

Appendix B

Sample Room Grammar

This appendix presents rules from the room grammar used throughout many of the examples in this thesis. The rules are written using the operations defined in Appendix A. The full grammar we developed contains rules to generate rooms of six types: cloisters, cathedrals, large rooms with raised ceilings, multi-level rooms with balconies, hallways, and stairways. In the interest of brevity, we provide here only the rules used to generate cloisters and cathedrals. The rules for other rooms are similar in character.

Shape Symbols

Symbol	Minimum Size	Maximum Size	Required Factor
Cloister	*x*: 3, *y*: 3, *z*: 2		
Colonnade	*x*: 1, *y*: 1, *z*: 2		
TerminalColonnade			
TerminalGroinVault		*z*: 1	
Courtyard			
HalfCourtyard			
TerminalCourtyardPath			
TerminalCourtyard			
Cathedral	*x*: 6, *y*: 4, *z*: 4		
NaveAndSpace	*x*: 2, *y*: 2, *z*: 3		
Nave	*x*: 2, *y*: 2, *z*: 3		
TerminalLargeColonnade	*x*: 2, *y*: 2		*x*: 2, *y*: 2
TerminalLargeGroinVault	*x*: 2, *y*: 2, *z*: 2	*z*: 2	*x*: 2, *y*: 2

Connector Symbols

Symbol	Size	Flags	Required Flags
SmallConnector	*x*: 1, *y*: 1, *z*: 2	none	none
LargeConnector	*x*: 2, *y*: 2, *z*: 3	none	none

Valid Axiom Symbols

Symbol	Minimum Size	Size Increment	Maximum Increments
Cloister	*x*: 5, *y*: 5, *z*: 2	*x*: 2, *y*: 2, *z*: 1	*x*: 2, *y*: 2, *z*: 2
Cathedral	*x*: 6, *y*: 4, *z*: 4	*x*: 2, *y*: 2, *z*: 1	*x*: 6, *y*: 6, *z*: 6

Rules for Generating Cloisters

---

rule *CloisterConnectors*
predecessor symbol: *Cloister*
conditions: *PlacedConnectors* = 0 (true if undefined)
operations:
SetAttribute (*PlacedConnectors*, 1)
PlaceShape (*Symbol*: *Cloister*)
PlaceConnectors (*Symbol*: *SmallConnector*, *AnchorY*: *Centre*, *AnchorZ*: *Negative*, *SpacingY*: 1, *SpacingZ*: 0)
Reorient (90° around *z*-axis)
PlaceConnectors (*Symbol*: *SmallConnector*, *AnchorY*: *Centre*, *AnchorZ*: *Negative*, *SpacingY*: 1, *SpacingZ*: 0)
Reorient (90° around *z*-axis)
PlaceConnectors (*Symbol*: *SmallConnector*, *AnchorY*: *Centre*, *AnchorZ*: *Negative*, *SpacingY*: 1, *SpacingZ*: 0)
Reorient (90° around *z*-axis)
PlaceConnectors (*Symbol*: *SmallConnector*, *AnchorY*: *Centre*, *AnchorZ*: *Negative*, *SpacingY*: 1, *SpacingZ*: 0)

---

rule *DecomposeCloister*
predecessor symbol: *Cloister*
conditions: none
operations:
Push
Resize (*Dimension*: *y*, *Anchor*: *Negative*, *NewSize:* 1)
PlaceShape (*Symbol*: *Colonnade*)
Restore
Resize (*Dimension*: *y*, *Anchor*: *Positive*, *NewSize:* 1)
PlaceShape (*Symbol*: *Colonnade*)
PopAndRestore
Reduce (*Dimension*: *y*, *Anchor*: *Centre*, *Amount*: 2)
SplitMultiple (*Axis*: *x*, *Symbols*: {*Colonnade*, *Courtyard*, *Colonnade*}, *Sizes*: {1,-1,1})

---

rule *DecomposeColonnade*
predecessor symbol: *Colonnade*
conditions: none
operations:
SplitSingle (*Axis*: *z*, *Anchor*: *Positive*, *SymbolBefore*: *TerminalColonnade*, *SymbolAfter*: *TerminalGroinVault*)

---

rule *DecomposeCourtyard*
predecessor symbol: *Courtyard*
conditions: none
operations:
SplitMultiple (*Axis*: *x*, *Symbols*: {*HalfCourtyard*, *TerminalCourtPath*, *HalfCourtyard*}, *Sizes*: {1, -1, 1})

---

rule *DecomposeHalfCourtyard*
predecessor symbol: *HalfCourtyard*
conditions: none
operations:
SplitMultiple (*Axis*: *y*, *Symbols*: {*TerminalCourt*, *TerminalPath*, *TerminalCourt*},
Sizes: {1, -1, 1})

Rules for Generating Cathedrals

---

rule *DecomposeCathedral*
predecessor symbol: *Cathedral*
conditions: none
operations:
SetAttributeByFunction (*Name*: *TranseptWidth*, *Function*: *GetTranseptWidth*)
SetAttributeByFunction (*Name*: *NaveLength*, *Function*: *GetNaveWidth*)
SplitMultiple (*Axis*: *x*, *Symbols*: {*NaveAndSpace*, *Transept*, *NaveAndSpace*},
Sizes: {*NaveLength*, *TranseptWidth*, -1})

---

function *GetTranseptWidth* (shape *Projected*)
Width ← *Projected.GetSize*(*y*) / 2
if *Width* > *Projected.GetSize*(*y*) – 4:
Width ← *Projected.GetSize*(*y*) – 4
if *Width* > *Projected.GetSize*(*x*) / 4:
Width ← *Projected.Size*(*x*) / 4
if *Width* < 2:
Width ← 2
round *Width* **down to nearest factor of** 2
return *Width*

---

function *GetNaveLength* (shape *Projected*)
Remaining ← *Projected*.*GetSize*(*x*) – *Projected*.*AttributeValue*(*TranseptWdith*)
Length ← 2 × *Remaining* / 3
if *Length* **is even**:
Length ← *Length* – 1
return *length*

---

rule *DecomposeNaveAndSpace*
predecessor symbol: *NaveAndSpace*
conditions: none
operations:
SetAttributeByFunction (*Name*: *NavePadding*, *Function*: *GetNavePadding*)
SetAttributeByFunction (*Name*: *AislesToAdd*, *Function*: *GetAislesToAdd*)
ReduceByAttribute (*Axis*: *y*, *Anchor*: *Negative*, *Attribute*: *NavePadding*)
ReduceByAttribute (*Axis*: *y*, *Anchor*: *Positive*, *Attribute*: *NavePadding*)
PlaceShape (*Nave*)

---

function *GetNavePadding* (shape *Projected*)
if *Projected*.*GetSize*(*y*) < 6:
return 0
else
return *floor*(*Projected.GetSize*(*y*) / 5)

---

function *AislesToAdd* (shape *Projected*)
if *Projected*.*AttributeValue*(*NavePadding*) **is odd**:
return 1

```
    else:
        if Projected.GetSize(y) ≥ 6 and Projected.GetSize(z) ≥ 4:
            return 2
        else
            return 0
```

```
function AisleHeightReduction (shape Projected)
    if Projected.HasAttribute(AisleHeightReduction):
        Previous = Projected.GetAttribute(AisleHeightReduction)
        if (Previous ≤ 2):
            return Previous – 2
        else if Previous is even:
            return Previous / 2
        else:
            returnn (Previous + 1) / 2
    else:
        return Projected.GetSize(z) / 2
```

---

rule *NaveSideConnectors*
predecessor symbol: *Nave*
conditions: *PlacedConnectors* = 0 (true if undefined)
operations:
SetAttribute (*PlacedConnectors*, 1)
PlaceShape (*Symbol*: *Nave*)
Reorient (90° around *z*-axis)
PlaceConnectors (*Symbol*: *SmallConnector*, *AnchorY*: *Negative*, *AnchorZ*: *Negative*, *SpacingY*: 1, *SpacingZ*: 0)
Reorient (180° around *z*-axis)
PlaceConnectors (*Symbol*: *SmallConnector*, *AnchorY*: *Negative*, *AnchorZ*: *Negative*, *SpacingY*: 1, *SpacingZ*: 0)

---

rule *GreatWestDoor*
predecessor symbol: *Nave*
conditions: *PlacedConnectors* = 1 (false if undefined)
operations:
SetAttribute (*PlacedConnectors*, 2)
PlaceShape (*Symbol*: *Nave*)
Reorient (180° around *z*-axis)
PlaceConnectors (*Symbol*: *LargeConnector*, *AnchorY*: *Centre*, *AnchorZ*: *Negative*, *SpacingY*: 3, *SpacingZ*: 0)

---

rule *AddAisle*
predecessor symbol: *Nave*
conditions:
AislesToAdd > 0 (false if undefined)
y-size ≥ 4
z-size ≥ 3
operations:
SetAttributeByFunction (*Name*: *AisleHeightReduction*, *Function*: *GetAisleHeightReduction*)
ChangeAttribute (*AislesToAdd*, -1)
Push
Resize (*Axis*: *y*, *Anchor*: *Negative*, *NewSize*: 1)
Reduce (*Axis*: *z*, *Anchor*: *Negative*, *Amount*: *AisleHeightReduction*)

PlaceShape (*Symbol*: *Colonnade*)
Restore
Resize (*Axis*: *y*, *Anchor*: *Positive*, *NewSize*: 1)
Reduce (*Axis*: *z*, *Anchor*: *Negative*, *Amount*: *AisleHeightReduction*)
PlaceShape (*Symbol*: *Colonnade*)
PopAndRestore
Reduce (*Axis*: *y*, *Anchor*: *Centre*, 2)
PlaceShape (*Nave*)

---

rule *DecomposeNave*
predecessor symbol: *Nave*
conditions:
AislesToAdd = 0 (true if undefined)
operations:
SplitSingle (*Axis*: *z*, *Anchor*: *Positive*, *Distance*: 1,
SymbolBefore: *TerminalLargeColonnade*,
SymbolAfter: *TerminalLargeGroinVault*)

Appendix C

Sample Mesh Placement Rules

These mesh placement rules correspond to the room grammar presented in Appendix B. They place meshes from the *Multistory Dungeons* [Mana 2015] mesh kit.

Rules for Pass 2 – Shape Processing

---

if *Shape* **has symbol** *TerminalGroinVault*, **use function**:
    for each grid position *GridPosition* **inside** *Shape*:
        CellSlot ← cell slot **at** *GridPosition*
        add mesh "SM_Ceiling_04" **translated by** (0, 0, -130) **at** *CellSlot*
    for each *x* or *y* face slot *FaceSlot* **on the boundary of** *Shape*:
        add flag *SmallArch* **to** *FaceSlot*
        add flag *CheckWall* **to** *FaceSlot*
    for each edge slot *EdgeSlot* **inside or on the boundary of** *Shape*:
        add flags *SmallPlinth*, *ColumnBase* **to** *EdgeSlot*

---

if *Shape* **has symbol** *TerminalGroinVaultLarge*, **use function**:
    AABB ← *Shape.BoundingBox* **as** axis-aligned bounding box
    for each integer *X* **at intervals of** 2 **from** *AABB.Min.x* **to** *AABB.Max.x* – 1:
        for each integer *Y* **at intervals of** 2 **from** *AABB.Min.y* **to** *AABB.Max.y* – 1:
            CellSlot ← cell slot **at** position (*X*, *Y*, *AABB.Min.z*)
            add mesh "SM_Ceiling_05" **translated by** (300, 300, 0) **at** *CellSlot*
    for each integer *X* **at intervals of** 2 **from** *AABB.Min.x* **to** *AABB.Max.x*:
        for each integer *Y* **at intervals of** 2 **from** *AABB.Min.y* + 1 **to** *AABB.Max.y* – 1:
            XFaceSlot ← *x* face slot **at** position (*X*, *Y*, *AABB.Min.z*)
            add flag *LargeArch* **to** *XFaceSlot*
    for each integer *X* **at intervals of** 2 **from** *AABB.Min.x* **to** *AABB.Max.x*:
        for each integer *Y* **at intervals of** 2 **from** *AABB.Min.y* **to** *AABB.Max.y* – 1:
            XFaceSlot ← *y* face slot **at** position (*X*, *Y*, *AABB.Min.z*)
            add flag *LargeArch* **to** *YFaceSlot*
    for each *x* or *y* face slot *FaceSlot* **on the boundary of** *Shape*:
        add flag *CheckWall* **to** *FaceSlot*
    for each edge slot *EdgeSlot* **inside or on the boundary of** *Shape*:
        add flags *SmallPlinth*, *ColumnBase* **to** *EdgeSlot*

---

if *Shape* **has symbol** *TerminalColonnade*, **use function**:
    for each x or y face slot *FaceSlot* **on the boundary of** *Shape*:
        add flags *CheckRailing*, *CheckWall* **to** *FaceSlot*
    for each *z* face slot *FaceSlot* **on the bottom of** *Shape*:
        add flag *CheckFloor* **to** *FaceSlot*
    for each edge slot *EdgeSlot* **inside or on the boundary of** *Shape*:
        add flag *Column* **to** *EdgeSlot*

---

if *Shape* **has symbol** *TerminalLargeColonnade*, **use function**:
    for each x or y face slot *FaceSlot* **on the boundary of** *Shape*:
        add flags *CheckRailing*, *CheckWall* **to** *FaceSlot*
    for each *z* face slot *FaceSlot* **on the bottom of** *Shape*:
        add flag *CheckFloor* **to** *FaceSlot*
    for each edge slot *EdgeSlot* **inside or on the boundary of** *Shape*:
        add flag *Column* **to** *EdgeSlot*

---

if *Shape* **has symbol** *TerminalCourtyard* **or** *TerminalCourtyardPath*, **use function**:
    for each *z* face slot *FaceSlot* **on the bottom of** *Shape*:
        add flag *CheckFloor* **to** *FaceSlot*

Rules for Pass 3 – Side Face Processing

---

if *FaceSlot* **with adjacent cell slots** *A*, *B* **has flag** *SmallArch*, **use function**:
    add mesh "SM_Arch_03" **translated by** (0, 0, -200) **at** *FaceSlot*
    if (*A* **does not have flag** *GroinVault* **and** *A* **does not have no flag**) **or**
            (*B* **does not have flag** *GroinVault* **and** *B* **does not have no flag**):
        # patch hole between groin vault and open space inside a room
        add mesh "SM_Arch_04" **translated by** (0, 0, -200) **at** *FaceSlot*

---

if *FaceSlot* **with adjacent cell slots** *A*, *B* **has flag** *LargeArch*, **use function**:
    add mesh "SM_Arch_Big_02" **translated by** (300, 0, -200) **at** *FaceSlot*
    if (*A* **does not have flag** *GroinVaultLarge* **and** *A* **does not have no flag**) **or**
            (*B* **does not have flag** *GroinVaultLarge* **and** *B* **does not have no flag**):
        # patch hole between groin vault and open space inside a room
        add mesh "SM_Arch_Big_03" **translated by** (0, 0, -200) **at** *FaceSlot*

---

if *FaceSlot* **with adjacent cell slots** *A*, *B* **has flag** *CheckRailing*, **use function**:
    if (*A* **has flag** *TerminalCourtyard*) **or** (*B* **has flag** *TerminalCourtyard*):
        add mesh "SM_Railing_01" **translated by** (300, 0, -110) **at** *FaceSlot*

---

if *FaceSlot* **with adjacent cell slots** *A*, *B* **has flag** *CheckWall*, **use function**:
    if (*A* **has no flag**) **and** (*B* **has any flag**):
        add mesh "SM_FP_Wall_04" **translated by** (0, 0, -200) **at** *FaceSlot*
    else if (B **has no flag**) **and** (*A* **has any flag**):
        add mesh "SM_FP_Wall_04" **translated by** (0, 0, -200) **and**
                rotated by (180° around *z*-axis) **at** *FaceSlot*

Rule for Pass 4 – Bottom Face Processing

---

if *FaceSlot* **has flag** *CheckFloor*, **use function**:
    Above ← cell slot **at grid position of** *FaceSlot*
    Below ← cell slot **at** (**grid position of** *FaceSlot*) + (0, 0, -1)
    if (*Abobe* **has any flag**) **and** (*Below* **has no flag**):
        add mesh "Mesh_Base_01" **translated by** (0, 0, -400) **at** *FaceSlot*

Rule for Pass 5 – Edge Pre-Processing

```
if EdgeSlot has flag Column, use function:
    Below ← edge slot at (grid position of EdgeSlot) + (0, 0, -1)
    if (Below does not have flag Column):
        add flag ColumnBase to EdgeSlot
    else if (Below does not have flag ColumnBase):
        add flags SmallPlinth, ColumnBase to EdgeSlot
```

Rule for Pass 6 – Edge Processing

```
if EdgeSlot has flags Column and ColumnBase, use function:
    Above ← edge slot at (grid position of EdgeSlot) + (0, 0, 1)
    if (Above does not have flag Column) or (Above has flag ColumnBase):
        if (EdgeSlot has flag SmallPlinth):
            add mesh "Mesh_Column_12" translated by (0, 0, -200) at EdgeSlot
        else
            add mesh "Mesh_Column_11" translated by (0, 0, -200) at EdgeSlot
    else if (Below does not have flag ColumnBase):
        if (EdgeSlot has flag SmallPlinth):
            add mesh "Mesh_Column_10" translated by (0, 0, -200) at EdgeSlot
        else
            add mesh "Mesh_Column_08" translated by (0, 0, -200) at EdgeSlot
```