DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Story Guided Procedural Generation of Complex Connected Worlds and Levels for Role Play Games

Theobald Beyer, B. Sc.

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Story Guided Procedural Generation of Complex Connected Worlds and Levels for Role Play Games

Handlungsbasierte prozedurale Generierung von komplexen verbundenen Welten und Leveln für Rollenspiele

Author:	Theobald Beyer, B. Sc.
Supervisor:	Prof. Gudrun Klinker, Ph.D.
Advisor:	Sven Liedtke, M. Sc.
Submission Date:	15.12.2017

I confirm that this master's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15.12.2017 Theobald Beyer, B. Sc.

Acknowledgments

As with all works, a lot of thanks is due to people without whom this thesis wouldn't exist in the way it does.

To my advisor Sven Liedtke for all the help and critique he had for me and for making it possible for me to write about this topic in the first place.

To my father Thomas Beyer and my friend Leon Goldmann for proofreading.

And to my dear Bonny for bearing with me and encouraging me during all the times I might have been a little grumpy and/or stressed. And for kicking my butt, whenever I started to procrastinate.

Thank you.

Abstract

Procedural content generation had not only a rise in the games industry in recent years but also became a topic in dedicated scientific work. Overlooking past game releases, however, one might get the impression, that procedural content generation of whole worlds on one hand and a strong story for the player to follow on the other hand are mutually exclusive.

This work presents an overview of established methods, concepts, and techniques in the field of procedural content generation. Furthermore, it presents an approach to implement a generator for a whole, complex and interconnected world for a role-playing game. This partly includes the storyline the player follows. The story is used as a scaffolding for the world to be wrapped around so that the world is best suited to accommodate the story.

The approach is divided into three main steps: The generation of the storyline, the generation of the world, according to the storyline, and the generation of single levels according to the world, including detailed information on every step.

Contents

1. Introduction

1.1. Procedural Content Generation

As Richard Bartle states in his work about player types *Hearts, clubs, diamonds, spades: Players who suit MUDs* from 1996 [Bar96], there are four types of players to be found in multi-player games. These four types, the *achiever*, *explorer*, *killer*, and *socialiser*, are the extremes on a two dimensional chart defined by two axis: acting vs. interacting and players vs. world. While *killers* and *socialisers* act on or interact with other players, *achievers* and *explorers* act on or interact with the games world and levels.

To prevent a player from getting bored, his needs according to Bartles player type taxonomy need to be met. The *achiever's* and *explorer's* focus on the game world instead of on other players leads to a high demand for new content. *Achievers* gain satisfaction by completing given challenges. They need new challenges, as soon as the old ones are achieved. Therefore, a great amount of content is necessary to prevent *achievers* from getting bored. *Explorers* on the other hand get bored when they feel like they know all the secrets of the world they are playing in. In conclusion, every new world or level is a new mystery to explore for *explorers*.

Unfortunately the manual creation of quality content for games is expensive and time-consuming. Procedural generation can help to make this process less time-consuming for humans or even replace human work effort completely. Literally interpreted, procedural creation means to create something following a given procedure. But this definition is very broad and would include baking an apple pie according to a recipe. Since this work is about procedural content generation for games, a slightly extended definition from Shaker and coworkers' *Procedural Content Generation in Games, A textbook and an overview of current research* [STN16] is used:

"Procedural generation is the algorithmic creation of content with limited or indirect user input [, opposed to manual creation]."

Procedural generation can be separated into sub-sections by several attributes, as done so by Togelius et al. in *Search-based Procedural Content Generation* [Tog+11]. One of these axes in which PCG can be defined is online versus offline generation. Togelius et al. describe the difference as *"whether content generation is performed online during* *the runtime of the game, or offline during game development"* [Tog+11]. The mentioned intermediate cases, where the generation doesn't happen during the development of the game, but also not while directly playing the game, are here counted as offline generation as well. This work will only consider offline generation of content, which does not mean, that the described techniques cannot also be suitable for online content generation. Procedural content generation will further be abbreviated as PCG.

An example for a way to meet the above mentioned needs of both player types, *achievers* and *explorers*, with procedural generation is the *Universal Pokemon Game Randomizer* [Dab]. This is a user-written program that allows other users to randomize the occurrence of Pokémon in some Pokémon games. The randomized versions present a new challenge for *achievers* since they now have to adapt to the current world and find new ways to beat the game with the given choice of Pokémon. It also presents a world in which the player has no information about the occurrence of Pokémon and therefore gives the *explorer* content to explore. This program also was part of the inspiration for this work.

PCG can not only be used to holistically generate content, but also only partially generate the content, modify given content or work as a tool in the development process. Explained on the example of level-generation, a holistic generator would be able to create a whole, functioning level without any given user input. On the other end, a generator simulating erosion mechanisms could be used as a tool during development, while everything else is done manually by a designer. In this work a mostly holistic approach is considered, that works without a designer reviewing the results. But since replacing single steps of a generator with manual work is done easily, all techniques can also be used for more partial approaches.

1.2. Statement of the Problem

This work presents an approach to generate a narrative and the corresponding world including level layouts, non-player characters (NPCs), dialogues and other additional information. This approach is shown on the example of generating a game in the style of the early Pokémon games for the Game Boy. These games are role-playing games published by Nintendo between 1998 and 2004 [Gam99] [Gam00] [Gam01b] [Gam01a] [Gam99] [Gam04] [Gam05].

All games follow the same principles: The player navigates the protagonist in a top-down view through a tile-based world. The goal is to become the best Pokémon trainer and to catch all Pokémon. To achieve that, the player travels through the world, catches wild Pokémon and battle NPCs to train them. During this journey different narrative events happen.

1.2.1. A Story-guided Approach

The storyline is the core of every role-playing game. By design, the world is created to serve as a space for the story to take place in. Therefore the story has to be taken into account when a world is generated for a role-playing game. In this work, the (procedurally generated) story is used as a basic framework around which the world is modeled.

1.3. Related Work

Procedural content generation is used in many games for various purposes and to different extents. The most popular example is terrain generation, as it is done in games like *No Man's Sky* [Hel16] or *Minecraft* [Moj11]. In those games the 3D-landscapes the game takes place in are completely generated. In other games only components of the game level are randomized, like in random dungeons in *Diablo III* [Bli12], where the base level is fixed, but free spaces are filled in with pre-factored modules. Tile-based level maps are also often generated, like the large world maps in the *Civilization* series [Fir10] [Fir14] [Fir16]. Other games feature procedural enemies, often seen in strategy games where the player can equip his units with weapons and skills or compose them of given components. The AI the player battles has to equip and construct units, like the player, resulting in generated enemies. Examples for this are *Endless Legend* [Amp14] and *Earth 2150* [Rea00].

Despite the above mentioned, there are many other examples of games featuring some sort of PCG. The online distribution platform *Steam* lists over 250 games in the category *procedural generation*, but in terms of story generation or story-based generation of worlds, not much can be found. One example is Kitfox's *Moon Hunters* [Kit16]. It generates the world and story while playing and reacts to the player's choices and actions. Though the attention here is on the generated story, it's concept is very different from the story-guided generation of a whole world before playing. *Moon Hunters* generates an environment in which stories could take place, but the generation of the worlds is not guided by any story elements.

In recent years procedural generation in games was approached in scientific ways, including a taxonomy in 2010 [Tog+11] and a whole book to provide an overview in 2016 [STN16]. Many works on specific problems were published, but barely any connected different PCG disciplines in one project.

This work explains several concepts and techniques and combines them to a holistic approach to generating a story, world, and levels for a role-playing game.

2. Basic Concepts and Techniques

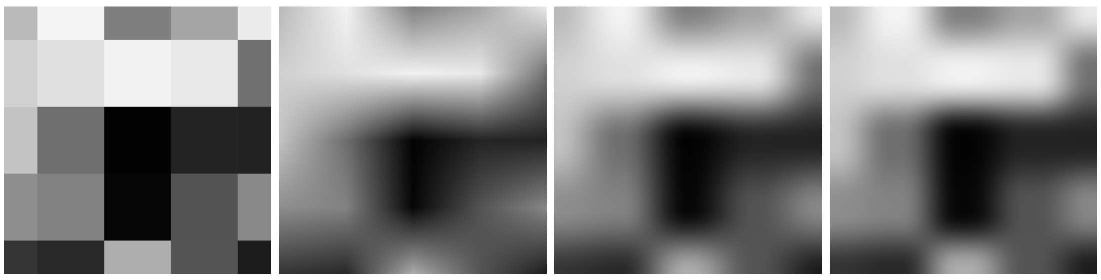
2.1. Noise

Often in PCG natural patterns are mimicked in one, two or more dimensions. Such natural patterns are seldom completely random. In most cases points close to each other are somehow related. To generate such patterns, coherent noise is used. There exist several techniques to procedurally generate a variety of coherent noise patterns. The most established techniques are explained in this section.

Value Noise

Value noise is defined by random values on a grid. Every point in the domain can be interpolated with various interpolation methods. Different interpolation methods result in different noise textures. In value noise the grid structure is often visible to the human eye. These artifacts often outweigh the benefits of cheaper computation compared to gradient noise.

In the following examples three different interpolation functions are used: Linear interpolation $f(t) = t$, cosine interpolation $f(t) = (cos(t \cdot \pi + \pi) + 1)/2$ and a smoothstep interpolation $f(t) = 6t^5 - 15t^4 + 10t^3$. This smoothstep function is chosen, because its 1st and 2nd derivative is 0 for $t = 0$ and $t = 1$. Figure 2.1 shows all three interpolation functions and an image without interpolation.



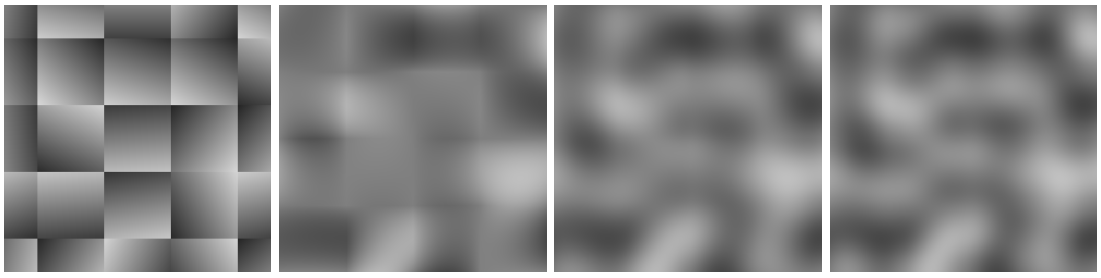
(a) no interpolation (b) linear (c) cosine (d) smoothstep

Figure 2.1.: Value noise with different interpolation methods.

Gradient Noise

To overcome the grid-based artifacts of value noise, Ken Perlin invented *Perlin Noise* in 1985 [Per85]. This is the first known implementation of gradient noise. Instead of a single value, every grid point holds a gradient. This can be seen in figure 2.2a. The value of any point p in the domain for the gradient belonging to grid point g can be computed with the dot product between the gradient vector and the vector pointing from g to p, $p - g$. Though the linear interpolation still shows some artifacts, the cosine (figure 2.2c) and smoothstep (figure 2.2d) interpolations directional artifacts are highly decreased and they have a very natural feel.

Alongside Perlin Noise there are two established gradient noise algorithms: Simplex noise, which was also invented by Ken Perlin in 2001 [Per01], and OpenSimplex, which was developed as a patent-free alternative to Perlin's Simplex noise.



(a) no interpolation (b) linear (c) cosine (d) smoothstep

Figure 2.2.: Gradient noise with different interpolation methods.

Worley Noise

Worley noise was invented by Steven Worley in 1996 [Wor96]. Unlike value or gradient noise, Worley noise is not defined by values on a grid, but by an arbitrary set of points. Every position in the domain is defined by the distance to the closest point of the set. The distribution of points thus has a significant influence on the outcome. One way to distribute these points is Poisson disk sampling, as described in chapter 2.3.

Worley noise, often called cellular noise, has a cellular, organic look to it. It is closely related to Voronoi diagrams since every position in the domain is associated with the closest point. Figure 2.3 shows two Worley noise textures with different distribution methods.

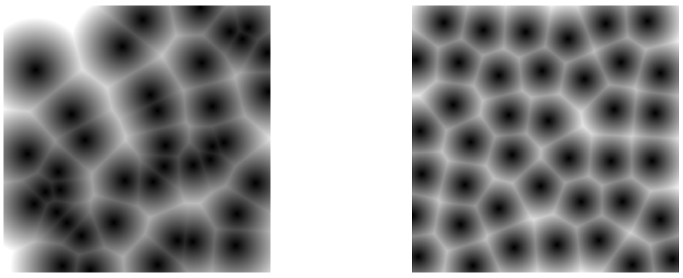

(a) random distribution (b) Poisson disk sampling

Figure 2.3.: Worley noise from different point sets.

Fractal Noise

Fractal noise describes the process of summing up several layers (*octaves*) of noise, rather than an own form of noise. Usually the scale of the noise increases per octave, whereas the intensity decreases. This leads to similar patterns on different scales, hence the name *fractal noise*.

Domain Warping

Until now, all described noise functions computed their values with unaltered coordinates. These coordinate values, however, can be changed, so that position (x, y) in the final texture holds the noise value of position (x', y'). This process is called domain warping. When done with linear transformations, like translation or rotation, the outcome is visually similar to the original. But non-linear transformation functions can have an interesting effect on a noise texture.

The images in figure 2.4 show warped gradient and Worley noise textures as well as their unwarped bases. As warp function a circular function $f(x, y) = (x + sin(g(x, y)), y + cos(g(x, y)))$ was used, where $g(x, y)$ is a gradient noise function returning the noise value at position (x, y) scaled to $[0 - 2\pi]$.

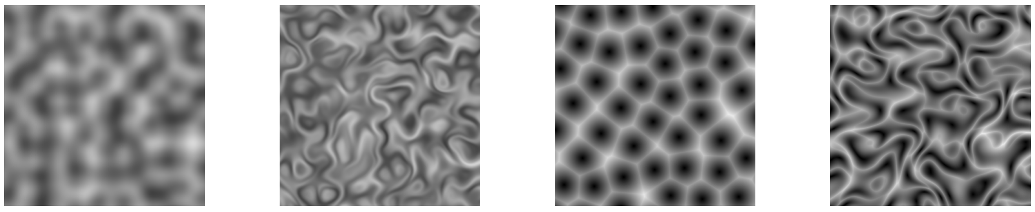

(a) gradient noise (b) warped gradient (c) Worley noise (d) warped Worley

Figure 2.4.: Example of warped noise.

2.2. Mazes

"A maze is a path or collection of paths, typically from an entrance to a goal" [Wik17]. Mazes can be classified by the following attributes [Pul15]:

branching	mazes with junctions and possible dead-ends
perfect	mazes with exactly one solution path between any two points (no cycles in the graph) and no unreachable areas
braided	branching mazes without dead-ends

The attribute *braided* is sometimes used quantitatively to indicate to what degree, the maze has loops and no dead-ends. But since this quality is not directly measurable, the attribute *braided* is here only used as defined above. The here presented algorithms produce perfect, branching mazes, but can also return single-path mazes as a special case. All these algorithms work on a graph. The examples are all shown on graphs representing orthogonal grids, but all of them work on arbitrary graphs (with slight adjustments). The graph a maze algorithm works on is here referred to as *base graph* and the resulting graph representing the maze as *result*. If no further sources are specified, the algorithms are described as in Buck's book *Mazes for Programmers: Code Your Own Twisty Little Passages* [Buc15].

Recursive Backtracking

The recursive backtracking algorithm simulates an individual traversing the graph. The algorithm starts at a starting position. From the current node it walks on a randomly selected edge to a neighboring node that has not been visited yet. If there are no unvisited neighbors available, the algorithm walks back in its history until it finds a node that has unvisited neighbors.

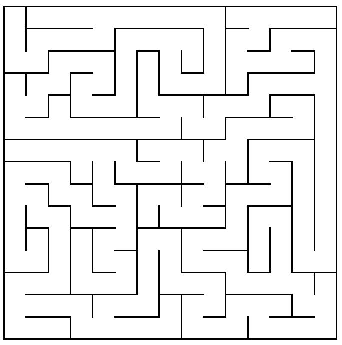


Figure 2.5.

This algorithm is often implemented as a stack since a stack simplifies the backtracking. This algorithm can be performed on any arbitrary graph, is guaranteed to be perfect and in most cases branching. The maze layout can be influenced by biases. On a grid, the algorithm could e.g. be biased to prefer horizontal connections by raising the chances of picking a horizontal connection compared to a vertical connection.

Characteristic for this algorithm are long, winding passages, since it only creates a junction if it runs into a dead-end. An example can be seen in figure 2.5.

Kruskal's Algorithm

Kruskal's algorithm [Kru56] is an algorithm to determine the minimal spanning tree of a graph with weighted edges. The algorithm starts with a list of all edges, sorted by weight, and a list of trees, each a subgraph of the base graph containing one node from the base graph. The algorithm then repeatedly selects the edge with the lowest weight. If the selected edge connects two separate trees, it is added to the result. If it introduces a cycle in the graph, it is deleted. This step is repeated until all nodes are in one tree. This is the minimal spanning tree of the base graph.

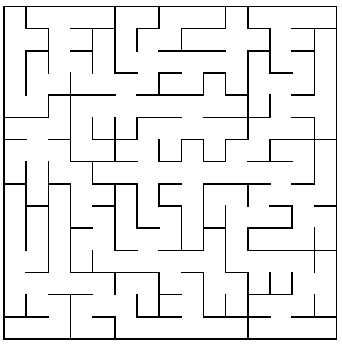


Figure 2.6.

To generate a maze with it, edges are not selected based on weight, but randomly. As per definition of a minimal spanning tree, the resulting graph is guaranteed to be a perfect and in most cases a branching maze. The results can be biased by changing the possibility with which each edge is selected. In Kruskal's algorithm, this can be done via the weight of the edges.

Kruskal's algorithm, if unbiased, typically results in a maze with a lot of short dead-ends. An example can be seen in figure 2.6.

Prim's Algorithm

Prim's algorithm [Pri57] is, like Kruskal's algorithm, an algorithm to determine the minimal spanning tree of a graph with weighted edges. Starting at any node in the graph, the algorithm adds this point to the result and all adjacent nodes to a list of possible new steps, here referred to as *frontier*. As long as there are possible steps in the frontier, the algorithm adds the one it can reach with the lowest cost. In general, Prim's algorithm can be implemented to run in linear time provided a sufficiently dense base graph [Tar].

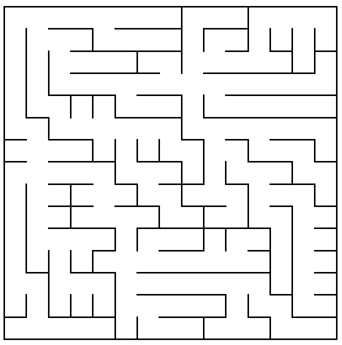


Figure 2.7.

Like Kruskal's algorithm, Prim's algorithm can generate random mazes if the next step is not selected based on weight, but randomly. The result is guaranteed to be a perfect and in most cases branching maze and the result can be biased by changing the possibility with which each edge is selected. The unbiased algorithm results in a maze with a lot of short dead-ends. An example is shown in figure 2.7.

Tree Growing Algorithm

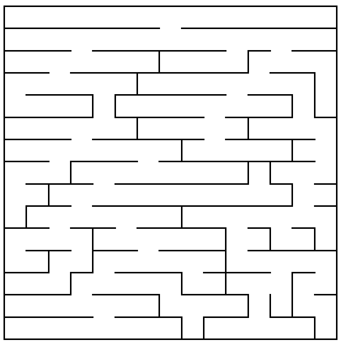


Figure 2.8.

The tree growing algorithm is a generalization of the recursive backtracking and Prim's algorithm. Like Prim's algorithm, it uses a frontier of possible next steps it constantly updates when taking a step. Unlike in Prim's algorithm, however, the position of the step in this frontier list, or other criteria, can be used to influence the possibility of that step to be taken next. If a random next step is selected, it works like Prim's algorithm, if always the newest entry in the frontier is selected, it works like recursive backtracking.

Apart from those two special cases, any other rule to select the next step can be used. This can lead to a variety of different characteristics and sometimes very regular patterns. Figure 2.8 shows a result in which horizontal edges were heavily favoured. All properties are identical to Prim's algorithm: It produces perfect and nearly always branching mazes.

Hunt-and-Kill Algorithm

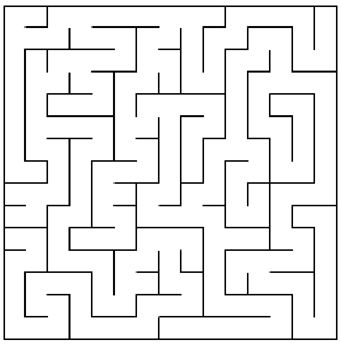


Figure 2.9.

The Hunt-and-Kill algorithm is a variant of the recursive backtracking. If it runs into a dead-end, it searches iteratively for the next node along the graph. A requirement is, that the graph is not necessarily finite, but denumerable. If it runs into a dead-end, the algorithm searches for the first node that is not part of the result yet, but adjacent to a node that's part of the result. The found node is added and connected to the result and used as starting point for the algorithms random walk.

Characteristically the Hunt-and-Kill algorithm is identical to the recursive backtracking: A perfect maze with long, winding corridors, branching in most cases. An example can be seen in figure 2.9.

On an infinite, enumerable graph a potentially infinite maze could be generated with this algorithm. For that, a boundary has to be set up to which point the maze needs to be generated. Then the algorithm generates as long as it has nodes in this set that are not in the result. Only if the player moves too close to the edge, the area is increased and the maze is further generated. This technique, however, bares the problem, that potentially a lot of the maze could be generated outside the required area before the required area is filled completely. This can lead to memory issues.

Eller's Algorithm

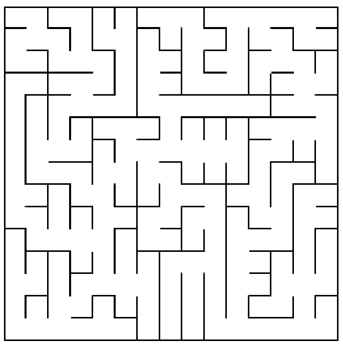


Figure 2.10.

Eller's Algorithm, invented by Marlin Eller and published by Pullen [Pul15] is a more complicated algorithm, but it provides an even combination of long winding corridors and small dead-ends. It also efficiently generates infinite mazes, since it builds them row-wise with just the knowledge of the last generated row.

In this example, a maze with fixed width but infinite height is built row by row. As a starting condition every node is assigned to its own separate set. Now randomly selected, adjacent nodes in the first row are joined into one set. The following steps are then repeated for every following row:

1. Randomly determine connections from the last row down to the current row. These are vertical connections.
2. Assign each node to the corresponding set if it has a vertical connection. Otherwise, assign it its own new set.
3. Join random adjacent nodes in the current row if they are not in the same set. These are horizontal connections.
4. The row is finished. Start for the next row at step 1.

To end the generation in any row, follow the steps as described above, but in step 3 join all adjacent nodes, that are not in the same set.

This algorithm not only creates possibly infinite mazes, it also requires a very low amount of memory since only the last and the current row have to be in memory for the generator instead of the whole maze. It produces perfect, branching mazes, as the example in figure 2.10 shows. If in step 3 nodes within the same set are joined, the resulting maze has cycles and thus is non-perfect.

The algorithm can be adjusted to work on an arbitrary graph, but in this case, this graph has to be created step-by-step in a possibly infinite manner. In this base graph every node in layer n needs to be connected to at least one node in layer $n - 1$. Also, in step 1 only one vertical connection can be added per node in the current layer if the resulting maze shall be perfect. As an example this could be done on any planar graph, that is repeated in a third dimension. Starting with the lowest layer, the algorithm works upwards layer by layer, where every node is connected to its counterpart in the last layer.

Sidewinder Algorithm

An alternative to Eller's algorithm is the sidewinder algorithm. Compared to Eller's algorithm, the sidewinder algorithm has one advantage in implementation: No sets need to be saved. To ensure a perfect maze, all nodes in the first row are in the same set and thus connected somehow. For every following row, the row is separated into adjacent subsets, which all get exactly one connection to the previous row. This way every node is connected to the first row and thus all nodes are in the same set. The downside of this algorithm is, that dead-ends can never go in the direction opposite to the generation direction because after a row is finished generating, there can not be any nodes not connected to the first row. An example is shown in figure 2.11 (the generation direction was top-to-bottom).

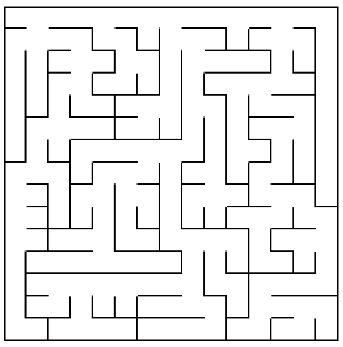


Figure 2.11.

Recursive Subdivision

The recursive subdivision algorithm is the only one on this list that doesn't work on a graph per se, since it does not carve out a path on a graph but builds walls instead. It can, however, be adjusted to work on a graph, as described below.

The recursive subdivision algorithm subdivides a given area into smaller areas while ensuring the resulting areas are connected. This process is done recursively with the resulting areas again until all areas are too small to be divided further. On a rectangular grid, for example, this could be to divide the rectangle into two smaller rectangles along the longer axis. After adding this wall, somewhere in the wall a passage is created. One noticeable feature in the results often are the first few walls, that stretch across the whole domain, as the wall between columns 4 and 5 in figure 2.12.

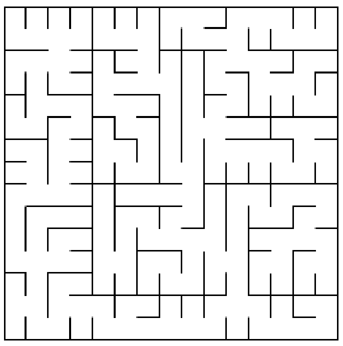


Figure 2.12.

Though it depends on the actual division rules of the algorithm, it can be very easily configured in a way, that the results are not only guaranteed to be perfect but also to be branching. Non-perfect mazes can be generated simply by adding more than one passage per split, which creates cycles.

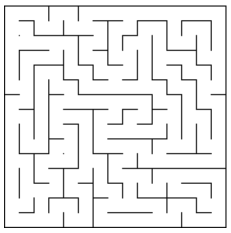
The characteristics of this algorithm are very hard to describe since they are heavily dependent on the division rules used. An area doesn't have to be split according to an orthogonal axis or a straight line. One could e.g. split a given area with a circular arc.

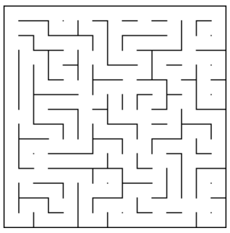
To use this algorithm on a graph, a spatial relation between the nodes has to be created. One could simply give every node a coordinate in a space of arbitrary dimensions. Then the spatial splits can be performed and every edge intersecting with the split is deleted except for one building the passage. Since in most cases, the nodes already have a spatial position when a maze should be created, this way might be the simplest to implement.

Another way to divide a graph into two subgraphs only connected by one edge is to randomly delete edges until two separate graphs exist. The edge deleted last is added to the result and is the passage between the two subgraphs and thus cannot be deleted in further divisions. This process is repeated on the subgraphs until no edges are left to delete.

Creating Braided Mazes

To create a braided maze (no dead-ends), one can use any branching maze, e.g. one generated with one of the algorithms above. To create a braided maze, every dead-end has to be eliminated. This is accomplished the following way: All dead-ends, being defined as a node with only one connection to another node, are stored in a list. For every dead-end, a new connection between the dead-end and an adjacent node is added to the result until no dead-ends are left in the list. If the number of circles shall remain small, any dead-end should preferably connect with an adjacent dead-end. This way two dead-ends can be removed with just one new connection, thus only adding one new cycle to the graph. Three examples of braided mazes created with recursive backtracking, Kruskal's algorithm, and a biased tree growing algorithm can be seen in figure 2.13.


(a) recursive backtracking


(b) Kruskal's algorithm

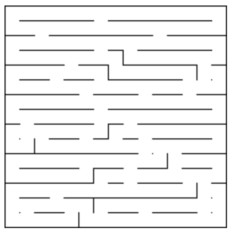

(c) biased tree growing

Figure 2.13.: Examples of mazes turned into braided mazes.

Converting Non-Perfect Mazes Into Perfect Mazes

To convert a non-perfect maze into a perfect one, all cycles need to be removed from the initial maze. For that, a flood-fill algorithm can be utilized. Starting from an arbitrary node the algorithm works its way in a breadth-first search across the graph. Whenever it would walk onto a node that has already been visited via another way, the edge between this node and the current node is deleted. The breadth-first search ensures, that the loop is separated the furthest away from the starting point as possible. This property can be helpful if one wants to extend the length of blind-alleys (passages that end in dead-ends) to be more equally distributed.

Creating Non-Branching Mazes

A non-branching or *unicursal* maze is a maze without junctions. It consists of just one, often winding, path and is often referred to as labyrinth. There are several ways to create a unicursal maze.

To create a unicursal maze out of a branching one, the base maze first needs to be converted to a perfect maze, as described below. After that, all dead-ends need to be sealed. Though this is a fairly simple way to create a unicursal maze, it is not guaranteed to be a winding path nor to traverse a maximum of the given nodes. Figure 2.14 shows the unicursal path between the top left and the bottom right corner. All dead-ends are sealed and marked grey. The base maze was the maze shown in figure 2.5.

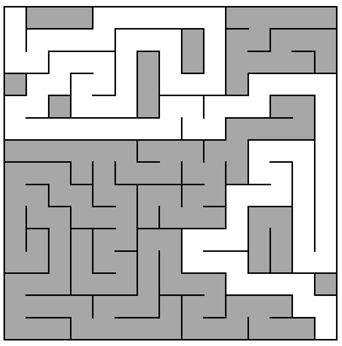


Figure 2.14.

To achieve a unicursal path that traverses a maximum of nodes, the best start is a braided maze. This base maze is traversed by a breadth-first search, that marks a step level on every node indicating after how much steps it got there. If the algorithm steps on the goal, the edge the algorithm came from is deleted, except it is the last edge connecting the goal node to the rest of the graph. If the algorithm steps on a node it already visited on another way, one of two things can happen: 1) The stepped on node has a lower step level than it would get from the current step. Then the algorithm seals up all connections from that node to adjacent nodes with a lower step level than its own. 2) The node has the same step level it would get from the current step, which means there is more than one path with equal distance to this node. In this case, all except one (randomly chosen) passage to adjacent nodes with lower step level are deleted. After that process, the algorithm starts anew. This process is repeated until the maze has no cycles left. Then all blind alleys are sealed off. The solution is guaranteed to be the longest possible path in the given braided maze, however, it is not guaranteed to be perfect, since some nodes might not be reachable.

A perfect, unicursal maze on any graph is a path that visits every node exactly once. Such a graph is called a Hamiltonian path and not every graph has one. The determination of the existence of such a path in a given graph is known as the *Hamiltonian path problem* and is NP-complete. Even if one solution is found, it has not necessarily its start and end points at the desired start and end points of the maze.

Classifying Maze Parts

On any graph structure, that is used to traverse over it, nodes can be classified into categories:

start	The starting point.
end	The end point.
dead-end	A path between a node with just one connection and a node with more than one connection and every node on the way has exactly two connections.
noose	A path connecting a junction node (three or more connections) to itself with no other junctions on the path.
blind alley	All nodes that when traversed coming from the start have to be traversed again to reach the end. (Every dead-end or noose is a blind alley)
critical node	A node that has to be traversed when going from start to end.

Not every node in a maze graph falls in one of those classes. Nodes that don't, are on one of several solution paths. If a graph has no such nodes, it is a perfect graph, since every node is either on the critical path or in a blind alley. An example of classified maze parts can be seen in figure 2.15.

Blind Alleys

To find all blind alleys, those areas can be 'filled up' recursively. This is done by a depth-first search starting from every junction node along every connection. The search only follows nodes with exactly two connections. The start and end point count as junctions in this scenario. If it finds a junction, that is not the node it started from, the followed path is neither a dead-end nor a noose. If it, however, finds a node with just one connection or the node it started from, the followed path is a dead-end or a noose and can be filled. This process is repeated recursively until no changes are made anymore. All that's left of the maze now is the sum of all the possible solutions from the start to the end. Everything else are blind alleys.

Knowing blind alleys is helpful in situations, where certain gameplay elements have to be placed, e.g. optional objectives or items to reward a player for exploration.

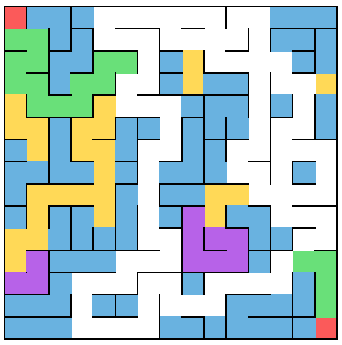


Figure 2.15.: red: start and end nodes
green: critical path
blue: dead-ends
yellow: blind alleys
purple: nooses
white: no special class

Critical nodes

Critical nodes can be found by running a left-hand solver and a right-hand solver on the sum of all possible solutions, as found after filling in all blind alleys as described above. A left-hand solver works the following way: starting at the start node it simulates a human walking through the maze while always touching the wall to its left side, taking always the leftmost option at junctions. This technique, however, can only solve mazes in which the start and end point are connected through a wall somehow. If the end point is on an island, the left-/right-hand solver will eventually come back to the starting point.

Knowing the critical nodes is also helpful when positioning gameplay elements with the knowledge that the player has to cross this particular node to reach the end. This could be a barrier the player can only cross with a certain tool or a story element the player has to find.

2.3. Poisson Disk Sampling

The positioning of objects in a given space is crucial to the feel of that area. Besides the average number of objects per area, the most important criterion is the choice of positions. If objects are positioned completely random, it often has an unnatural, messy feeling to it. If positioned on a grid, it is too regular for naturally occurring objects. Even positions on a jittered grid often seem unnatural. Poisson disk sampling is a method to get a natural and adaptable positioning of objects.

Poisson disk sampling is a method of sampling points in an arbitrary finite domain. Only a distance function has to be defined for the domain the points are sampled in. The result is a set of points within that domain that are *"tightly packed together, but not no closer to each other than a specified minimum distance"* [Tul08].

To generate a Poisson distribution, several techniques can be used, some more efficient than others. The easiest but most inefficient way is to randomly select points in the domain and check if they are too close to any existing point. This technique, however, might not find spaces where new points could be positioned due to its completely random nature.

A more efficient way is to systematically place points in proximity to existing points. The algorithm works with two lists, a *result* and a *processing* list. Starting with one randomly set point, this point is added to both lists. Then the following steps are repeated until the *processing* list is empty.

1. Select a random point p from the processing list.
2. Select n random points $[q_1, ..., q_n]$ in a ring defined by a minimal and maximal radius around the selected point p.
3. For each point q_i in $[q_1, ..., q_n]$ do the following:
   If q_i is closer to any point in the result list than the minimal distance, delete it.
   Otherwise, add q_i to the result and processing list.
4. Delete the selected point p from the processing list.

To check if the distance between two points in an n-dimensional space is smaller than an amount x is the same process as to test if two circles (or their n-dimensional equivalent) with radius $x/2$ collide. To accelerate this process, suitable acceleration data structures, like a quadtree or a bounding volume hierarchy, can be used.

One particularly interesting property of this way of sampling points is that parameters like the minimal and maximal distance can vary over a domain dependent on external influences. So could e.g. the distance of trees be dependent on the height of the terrain they are placed on. The sampling can also be done in several layers. One could place trees with some minimal distance, then with the tree points as starting points place shrubs with smaller minimal distance. The results from one layer can even be fed to the next layer. E.g. after placing large boulders the distance to the closest boulder is directly used to place smaller rocks to ensure those are grouped around the boulders. Examples of a regular Poisson disk sampling and one with varying minimal distance are shown in figure 2.16.

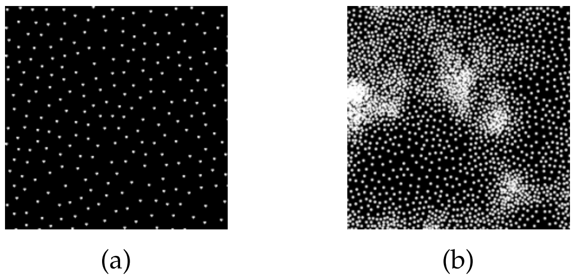


Figure 2.16.: Examples of Poisson disk sampling with fixed minimum radius (2.16a) and minimum radius defined by Perlin noise (2.16b). Images taken from `http://devmag.org.za/2009/05/03/poisson-disk-sampling/`

2.4. Cellular Automata

Cellular automata (CA) are a form of finite-state machines (FSM). A Cellular automaton consists of a regular grid of cells that each have a state (often just *on* or *off*). The next state of a cell (often called next generation) is defined by the states of the neighboring cells. The concept was first discussed by John v. Neumann and Stanislaw Ulam in the 1940s and published in von Neumann's work *The General and Logical Theory of Automata* [von51].

The neighborhood relation between two cells can, of course, be defined however one pleases. Two neighborhood types on orthogonal grids are most common: The *Moore* and the *von Neumann* neighborhood. The *von Neumann* neighborhood encompasses the four directly adjacent cells, that share an edge with the center cell. The *Moore* neighborhood additionally contains the four diagonally adjacent cells.

The most famous cellular automaton, which was also the one that brought attention to the topic in 1970, is Conway's *Game of Life* [Gar70]. The rules are the following: Possible states are *dead* and *alive*. A cell that is alive dies if less than two or more than three of its eight neighbors (*Moore* neighborhood) are alive. A dead cell becomes alive with exactly three living cells in its neighborhood, otherwise it stays dead.

These rules were constructed to simulate life with the factors of over- and underpopulation and are very unstable. When given the right rules, cellular automata can converge to stable structures. These structures often have a natural look and are usually used to create cave-like environments.

To generate a cave, the states used are *solid* and *traversable*. One converging rule set is the following: If the cell is solid, it becomes traversable if it has five or more traversable neighbors. If the cell is traversable, it becomes solid with five or more solid neighbors. Cells that are not in the grid are assumed always to be solid. An example for this rule set can be seen in figure 2.17. Figure 2.17a shows the initial state. It has a solid edge and the remaining cells are randomly filled with a 45% chance. Figure 2.17b shows the state after 16 iterations. This state is stable and wouldn't change according to the given rule set. Figure 2.17c and 2.17d show the results of initial states that were filled with a 42% or 48% chance.

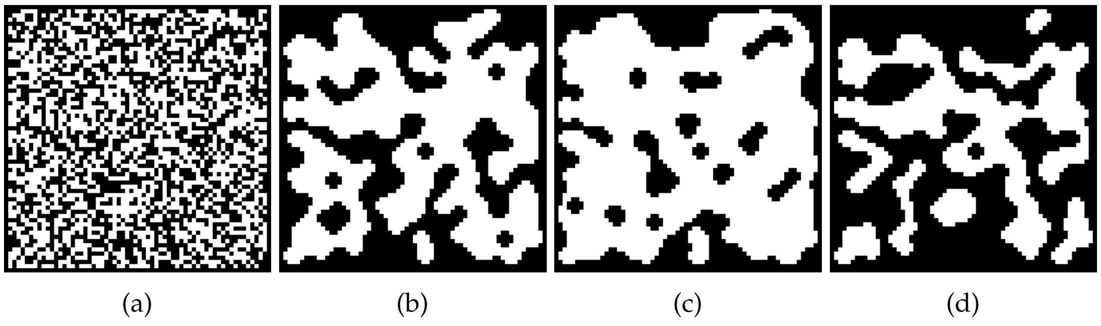


Figure 2.17.: Different states of cellular automata; white: *traversable*, black: *solid*

As seen in figure 2.17, the initial state is crucial for the outcome of the cellular automaton. If the input is not completely random, a cellular automaton can convert unnatural, clean visuals into natural looking ones. In figure 2.18 a maze generated with the recursive subdivision algorithm on a 4×4 grid with entrances at the upper left and lower right corner was used as input as seen in figure 2.18a. The maze was stretched to a size of 108×108 cells. The areas where solid and traversable cells meet were marked in figure 2.18b. This process also can be done by a cellular automaton. A possible rule set would be: A cell is marked if it finds a cell with a different state than itself in a radius of x. In figure 2.18c the marked cells are set to solid or traversable randomly. Figure 2.18d shows the outcome after 7 iterations. The general maze structure including the entrances is preserved, but it has an overall natural, cave-like feel to it.

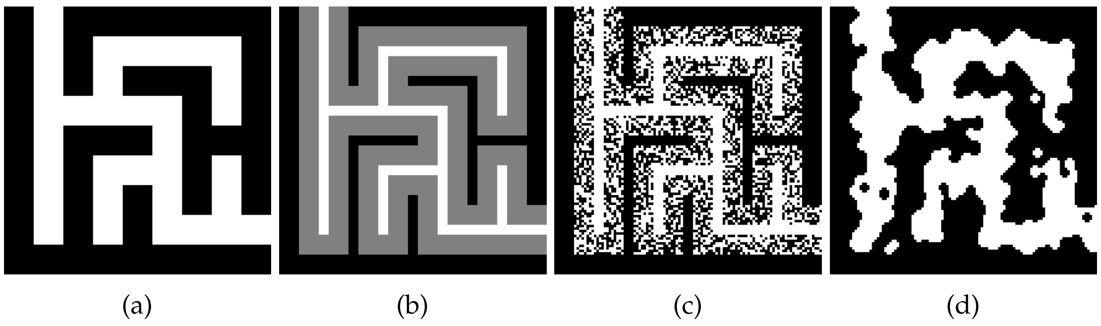


Figure 2.18.: A cave-like structure generated from a maze as basis.

2.5. Grammars

Formal grammars are a set of production rules that generate strings, starting with a single start symbol. Though these systems were initially introduced to generate text, they can be adapted to generate a variety of structures, such as objects like trees or houses as well as graph structures or whole game levels.

The basic principle of a generative grammar is a search and replace mechanism, that replaces patterns in the domain with (sometimes random) other patterns. These rules are defined in the form `a -> b`, which means `a` is replaced with `b`. Randomness can be added with options: `a -> b|c`. `a` is replaced by `b` or `c`. A grammar with the rule `s -> AsB|AB` and a start symbol `s` could generate strings with a random number of `A`s followed by the same number of `B`s. The same principle can be used on other structures than strings of symbols, too. As shown in the chapter *Grammars and L-systems with applications to vegetation and levels* of *Procedural Content Generation in Games (Computational Synthesis and Creative Systems)* [TSD16], such grammars work very well on graphs, too. An example is shown in appendix C.

Classical grammars usually have the problem, that they lack the ability to adjust according to context. As an example, if information about a person is generated by the following rules:

```
name -> Alice|Bob|Charly
gender -> male|female
age -> 1|2|3|...|100;
```

and start symbol `'name, age, gender'`, there are undesirable combinations of names and genders, e.g. `'Bob, 37, female'`. These cases would need to be described in a more complicated fashion to guarantee no undesired outcomes like so:

```
person -> mperson|fperson
mperson -> mname, age, male
fperson -> fname, age, female
mname -> Alice|Bob|Charly
fname -> Alice|Charly
age -> 1|2|3|...|100;
```

This inconvenient way of circumscribing problematic outcomes can be prevented with the addition of simple conditional rules. An example for that is *Forlog*, a grammar processing syntax with support for conditional statements and simple memory functions [Bey16]. In Forlog a *production rule* is written in capital letters, e.g. `RULE_NAME`. Possible outcomes are defined in the following lines starting with `'>'`. A production rule is called with square brackets, e.g. `[RULE_NAME]`. For every outcome a weight can be set, that determines the possibility of that particular outcome. This is done with a number sign followed by an integer, e.g. `#2>my option`. 'my option' will be selected twice as often as other rules. Besides these basics there are several other functionalities[1]. An example of a very simple Forlog grammar to generate cave names would be the following:

```
CAVE_NAME
>[PROPERTY] [CAVE]

PROPERTY
>dusty
>dark

CAVE
>cave
>cavern
```

The possible outcomes for calling the `CAVE_NAME` rule would be *'dusty cave'*, *'dusty cavern'*, *'dark cave'* and *'dark cavern'*. If a production rule is defined more than once, the production rule is the sum of all defined outcomes. An example of a Forlog grammar can be seen in appendix B.

[1]Further functionalities can be looked up at `http://deosis.de`.

3. Implementation

3.1. Basic Concept

In this chapter, a system is presented to generate a whole game world for a game similar to a Pokémon game, as explained in section 1.2. The generator starts by generating a simple narrative. Though it is linear in this case, the algorithms could be adapted quite easily to allow for branching, non-linear narratives. From that storyline, stored as a sequence of events, a world map is generated. The world map defines spatial relations between locations in the storyline. Once the world map is finished, the single level maps are generated. These are the actual tile maps the player can walk on and interact with.

During this whole process more and more data is collected and generated. All generated data is stored as entities in a database. Though a relational database would work or might even be necessary on larger projects, in this case native C# data structures are used to store the entities.

Data Structures

A lot of information is stored in lists, like NPCs or cities. But apart from this unsorted information, some data sets can be stored more efficiently in other data structures. In many cases data stored is associated with a coordinate on a map since both the world map and the level maps are based on orthogonal grids. In cases where data is distributed sparsely on the map, a lookup table (`Dictionary<TKey, TValue>` in C#, `std::map<Key, T>` in C++, etc.) can be used with a coordinate data structure as key. For dense data on the grid, a 2-dimensional array is better suited. For both a wrapper class can be written that simplifies working on the data and provides helper functions. More importantly, this wrapper class can also manage random content, that doesn't necessarily have to be generated all at once. Every time a value is requested, the wrapper class checks if it has been generated already and if not, generates it before it returns the value. This potentially saves a lot of computation time if out of a random consistent field just a few samples are taken.

3.2. Entity Management and Generation

Before the single generation steps can be implemented, all needed entities, all necessary information and the connections between each need to be planned and designed. Each entity holds several properties, including references to other entities. Some of those properties influence other properties. The type of Pokémon in a cave, for example, influences the naming possibilities. If a cave is e.g. occupied by fire type Pokémon, it can have a fire specific name, like *Mt. Ember*. Or a cave, that has two entries, can have names suggesting it is traversable, like *Rock Tunnel* or *Ice Path*. In both cases, however, the cave could also have a neutral name, like *Granite Cave*.

To allow such special cases, but also keep everything consistent, every property of an entity can be generated individually with respect to previously generated properties. This not only keeps consistency, but also allows the generator to set some properties, if necessary, while keeping others open. A few examples for entities are:

storyline event	`eventType, location, entity`
city	`name, gymLeader`
gym leader	`name, gender, pokemonType`
person	`name, gender`
dungeon	`name, isDeadEnd, dungeonType, pokemonType`
...	...

3.3. Story Generation

The story is generated in form of a sequence of events. These events can be designed in different ways, depending on the desired story. An example set of event types, that allows the modeling of a broad range of different stories is the following:

start	The player starts his adventure. Any preparation that is necessary is done here.
boss	The player has to beat a boss.
interact	The player has to interact with an NPC or another entity.
cross	The player has to cross a dungeon to the other side.
end	The player finishes his adventure and story is wrapped up.

These event types can also be used to describe a Pokémon game. They are adapted to suit the terminology. The following events are specified for the example:

start	The player acquires his first Pokémon and starts the journey.
gym	The player has to win the next gym fight.
interact	The player has to speak to an NPC (or interact with another entity).
cross	The player has to cross a dungeon to the other side.
league	The player has to win the battle against the elite four and the champion. This is also the end of the story.

The *start* and *league* events follow a similar structure in all reference games: In the *start* event, the player selects his first Pokémon out of three. One each of the types Fire, Water, and Grass. Along with it he gets a Pokedex and starts his journey.

The *league* event is the end of the storyline, in which the player first fights the elite four, four very strong trainers with a specific type of Pokémon, and the current Pokémon Champion. These five matches must be won in a row with the same team of Pokémon and without healing this team in between fights.

The objective in *cross* events is simply to cross a cave, forest or similar location. This kind of location will be called *dungeon* for the rest of this work. Examples are the *Viridian Forest*, the *Rock Tunnel* or the *Seafoam Islands* in the original games. The difficulty is to cross the dungeon without fainting, which happens when all carried Pokémon are beaten in battle, and to reach the next *Pokémon Center*, which functions as a kind of checkpoint. Though the *cross* events wouldn't need to be written down, since they are all necessary to complete before the next event, it is easier to plan and balance the story with these events as own entries. As an example for this, see the storyline of *Pokémon Red & Blue Edition* below. The player can only reach *Pewter City*, where he has to defeat the gym leader, by crossing the *Viridian Forest*.

In *interact* events the objective is to interact with an NPC. If a battle arises from the conversation, the event is only completed once the battle is won. If the battle is lost, the player faints and is set back to the last *Pokémon Center*. The entity to interact with does not necessarily have to be a person, it could e.g. be a switch or a computer. Most often the entity in question is found at the end of a dead-end dungeon, to present a further challenge in reaching it. An example is the *Berry Forest*, where the player has to rescue a girl named Lostelle. The girl waits at the end of a forest with a maze layout. By talking to her, the event is completed. Another example is the *Silph Co. Tower*, where the player has to reach the 10th floor to win a battle against *Team Rocket* boss *Giovanni*. This event is only completed if the player wins the fight.

In most cases the structure of a storyline in a Pokémon game is an alternating pattern of gym battles and other events (*interact*, *cross*). The beginning of the storyline of the games *Pokémon Red & Blue Version* [Gam99] looks the following:

```
start             Pallet Town
interact          Shop Keeper
interact          Prof. Oak
cross             Viridian Forest
gym               Pewter City
cross             Mt. Moon
gym               Cerulean City
interact          Bill
interact          Captain
gym               Vermilion City
...               ...
```

The following characteristics are equal in every Pokémon game: The first/home city has no gym; no city has more than one gym; there is a total of eight gyms. From these characteristics the following base structure can be derived: `(start; town1)`, `(gym; town2)`, `(gym; town3)`, `...`, `(gym; town9)`, `(league, town10)`. Every town at this point gets a name and, if it has a gym, a gym leader, including name and Pokémon type. By determining the type first, the city's and the gym leader's name can be generated in accordance. From all 18 available Pokémon types[1] eight are selected randomly. The named cities are saved to the list of cities, while the gym leaders are saved to the list of NPCs.

Now, between every existing event several *interact* and *cross* events can be put randomly. The number of events between each *gym* event should be between 1 and 3, with a maximum of one *cross* event. Given that `i` stands for *interact* and `c` for *cross*, the following nine combinations are possible: `i`, `c`, `ii`, `ic`, `ci`, `iii`, `iic`, `ici`, `cii`.

When a *cross* event is generated it gets a dungeon that needs to be crossed. A dungeon is created with `isDeadEnd` set to `false`, a `dungeonType` is selected, then the `pokemonType` and as a last step a suitable name is generated. For *interact* events one of several pre-defined interact events is selected randomly, to set the story background. An example type would be a *rescue mission*, where the player has to rescue a person at the end of a dungeon or a *acquire item mission*, in which the player has to find an item. These types of events are further specified to concrete events, e.g. to find a lost child in a forest or to liberate a warehouse from a group of gangsters by defeating

[1]Bug, Dark, Dragon, Electric, Fairy, Fighting, Fire, Flying, Ghost, Grass, Ground, Ice, Normal, Poison, Psychic, Rock, Steel, Water

their leader. If necessary, a suitable dungeon and necessary NPCs are generated. In the aforementioned cases, this would be a forest with a child NPC in it or a warehouse with several gangsters and one gangster boss.

Though it is not done in this example, a non-linear storyline could be represented as a directed graph and could be generated with a graph grammar.

3.4. World Generation

This section explains, how the storyline can be worked into a world map. Though the storyline is linear, the world does and should not be a non-branching graph. It rather should give the impression of a realistic world that is not built specifically for the storyline.

Though this task might seem fairly easy to a human, it is subject to a lot of subtle restrictions. Finding a solution to such a highly constrained problem is complex since every choice influences the possibilities of other choices. A step-by-step generator that chooses steps based on fitness functions usually can find very suitable steps at first but gets worse in the later process, since the first chosen steps limit the available options drastically.

3.4.1. Answer Set Programming

A way of solving such constraint-based systems is answer set programming (ASP). In ASP, the program is told *what* to achieve, without specifying *how* to achieve this goal [Bar03]. This goal is defined by an answer set, a list of facts and logical rules. A solver program is then used to find a set of facts that fulfills all requirements of the answer set. As Nelson states in *Procedural Content Generation in Games*: *"Answer set programming has a well-developed programming language with reliable existing tools, which can be used to specify both game logic and constraints within the same language. Therefore it serves as a good general-purpose choice for programming logic- and constraint-based PCG systems."* [NS16]. The *well-developed programming language* to define answer sets he mentions is *AnsProlog*. To solve the answer sets in this chapter the *clingo* solver version 4.5.4 of the *Potassco (Potsdam Answer Set Solving Collection)* [Geb+11] of the University of Potsdam was used.

In AnsProlog, a statement is defined either as a simple fact, a single <head>, or as a rule, a <head> followed by a condition as <body>. The `':-'` can be read as *'if'*:

```
<head>.
<head> :- <body>.
```

Declared facts simply evaluate to true. Parameters can be used to evaluate facts only for certain conditions. The following facts e.g. define that *Azure City* is a city and Azure City's shopkeeper is Mark.

```
city(azureCity).
shopkeeper(azureCity, mark).
```

An example for a conditional rule, one could define that every city, that has a shopkeeper, also has to have a shop:

```
hasShop(X) :- city(X), shopkeeper(X, _).
```

The underscore '_' is used as a wildcard for any input. The conditions separated by a comma ',' are combined by a logical AND ($\wedge$). In logical terms this rule could be written as $city(X) \wedge shopkeeper(X, \_) \Rightarrow hasShop(X)$.

The `(n..m)` notation is syntactic sugar and can be used to cover all integers between `n` and `m`.

```
p(1..4).    % is equal to:
p(1). p(2). p(3). p(4).
```

With these rules it would be possible to define a whole world. But doing this step by step would bring no benefit compared to every other step-by-step generator. Instead, in AnsProlog, a larger possibility space can be generated and after that, through rules, be trimmed down to all possibilities fulfilling the given rules. Such a possibility space is created by choice rules. The following rule defines, that between 6 and 8 cities shall have gyms, but it doesn't define which cities, nor exactly how much.

```
6 {hasShop(X) : city(X)} 8.
```

This statement can be written in words as follows: The number of `hasShop(X)` facts for which `X` is a city has to be at least 6 but not more than 8. An upper or lower bound can be undefined by simply not giving any number. A set to choose from can also be stated directly with semicolons separating the possibilities. The following line defines that exactly two elements out of the set $\{a; b; c\}$ shall be selected.

```
2 {a; b; c} 2.
```

Logically this evaluates to $(a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$. General constraints can be formalized as a rule without a `<head>`. So if the solver should reject all solutions in which *Berry Town* has a shop, the according rule would be the following:

```
:- hasShop(berryTown).
```

With these rules it is possible to generate a possibility space of valid worlds. *"By intermixing rules that create generative spaces, and others that prune them back down to interesting subsets, we can achieve strong control over the kind of content that is generated."* [NS16].

3.4.2. Example

With the basics of AnsProlog explained in the previous section, this section shows and explains an example of generating a world map with towns and cities on it, connected by routes. The structure of cities and routes is saved as a graph, in which both cities and routes are nodes and the edges are transitions between those places. A world map with only 5 cities and 6 routes is used to illustrate the procedure.

Though this whole process could be defined in one answer set, the possible solution space would grow very big resulting in an immense computational effort as well as vast amounts of used memory. Therefore the generation of the graph structure and the generation of the world are done in two separate answer sets.

Since the second step - the generation of the world map - is better suited to illustrate the procedure, the generation of the graph structure from the storyline will not be explained in detail. The used answer set can be found in appendix A. It ensures the following rules:

- Every city has to be reachable from the start city (here *Azure City*).
- The cities can be visited in the order in which they appear in the story.
- One city can have a maximum of three connections.
- The graph contains a minimum of `minCycles` cycles.

The output generated by this answer set is a graph structure with named cities but unnamed routes. The routes are then named in the order in which the player will use them when following the story. The single connection between cities and routes are extracted from the graph and fed into the second answer set, which is explained in detail below.

All cities and routes are saved as simple facts:

```
city(azureCity). city(berryTown). city(crimsonCity).
city(denimCity). city(ebonyTown).

route(route1). route(route2). route(route3).
route(route4). route(route5). route(route6).
```

Every city and every route also is a place.

```
place(P) :- city(P).
place(P) :- route(P).
```

Connections between the places are defined.

```
connection(azureCity, route1). connection(route1, berryTown).
connection(berryTown, route2). connection(route2, crimsonCity).
connection(crimsonCity, route3). connection(route3, denimCity).
connection(crimsonCity, route4). connection(route4, route5).
connection(route5, ebonyTown).
connection(ebonyTown, route6). connection(route6, berryTown).
```

Then the dimensions of the map and the underlying structure are defined. In this case a grid of 8×8 fields is used.

```
gridSize(1..8).
tile((X, Y)) :- gridSize(X), gridSize(Y).
```

A `contains(tile, place)` relation defines which place is at which tile on the grid and that one tile can contain at most one places.

```
0 {contains(T, P) : place(P)} 1 :- tile(T).
```

Every city has to have exactly one tile it sits on, while a route goes over several tiles that are connected in a straight line. For both a start and end point are defined. For a city, the start and end point are identical. For every route, a length between 2 and 5 is set.

```
% CITY
1 {placeStartTile(P, T) : contains(T, P)} 1 :- city(P).
placeEndTile(P, T) :- city(P), placeStartTile(P, T).

% ROUTE
1 {routeLength(R, (2..5))} 1 :- route(R).
:- routeLength(R, L), not L {contains(T, R)} L.
1 {entityStartTile(E, T) : contains(T, E)} 1 :- route(E).
1 {entityEndTile(E, T) : contains(T, E)} 1 :- route(E).
```

Now the program can generate some output, though a lot of constraints are still needed to generate a world map. An example output can be seen in figure 3.1. The cities are displayed as their initials, the routes as the respective numbers.

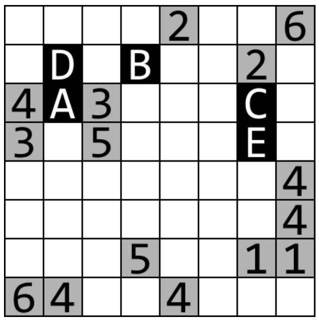


Figure 3.1.

At this stage, all pieces are scattered randomly on the map without any relation to each other. Route tiles are neither connected, nor in a straight line and cities are placed too close together. To overcome these problems, neighboring tiles are defined. The `illegalNeighbors(P1, P2)` rule is defined in a way, that two neighboring tiles containing two different places are allowed only if they are the start and end tile of an existing connection. Also, route tiles belonging to one route have to be connected and in a straight line, which is achieved by prohibiting the existence of two route tiles with different X *and* Y coordinates.

```
neighbors((X1, Y1), (X2, Y2)) :-
    tile((X1, Y1)),
    tile((X2, Y2)),
    |X1 - X2| + |Y1 - Y2| == 1.

% defining illegal neighbors
illegalNeighbors(P1, P2) :-
    P1 != P2,
    place(P1), place(P2),
    contains(T1, P1), contains(T2, P2),
    neighbors(T1, T2),
    not connection(P1, P2), not connection(P2, P1).
illegalNeighbors(P1, P2) :-
    P1 != P2,
    place(P1), place(P2),
    contains(T1, P1), contains(T2, P2),
    neighbors(T1, T2),
    connection(P1, P2),
    1 {not placeEndTile(P1, T1); not placeStartTile(P2, T2)}.

% prohibiting the existence of illegal neighbors
:- not 0 {illegalNeighbors(_,_)} 0.
```

```
% define connected route tiles...
connectedRouteTiles(R, T1, T2) :-
    route(R),
    contains(T1, R),
    contains(T2, R),
    neighbors(T1, T2).
connectedRouteTiles(R, T1, T3) :-
    connectedRouteTiles(R, T1, T2),
    connectedRouteTiles(R, T2, T3),
    T1 != T3.
% ...to define not connected route tiles and prohibit them
notConnectedRouteTiles(R, T1, T2) :-
    T1 != T2,
    route(R),
    contains(T1, R),
    contains(T2, R),
    not connectedRouteTiles(R, T1, T2).
:- not 0 {notConnectedRouteTiles(_,_,_)} 0.

% prohibiting the existence of non-straigt routes
noStraightLineRouteTiles(R, (X1, Y1), (X2, Y2)) :-
    route(R),
    contains((X1, Y1), R),
    contains((X2, Y2), R),
    |X1 - X2| * |Y1 - Y2| != 0.
:- not 0 {noStraightLineRouteTiles(_,_,_)} 0.
```

Figure 3.2 shows an example outcome of the current state. Cities are no direct neighbors, routes are all connected and in straight lines and the only direct neighbors of different kind are the ones with an existing connection (Crimson City and Route 3, Crimson City and Route 4, Ebony Town and Route 6).

The next steps ensure that the start and end points of the routes are at the actual ends, not somewhere in the middle of the route. Also, if a connection between Place $P1$ and Place $P2$ exists, the end point of $P1$ has to be a direct neighbor of the start point of $P2$.

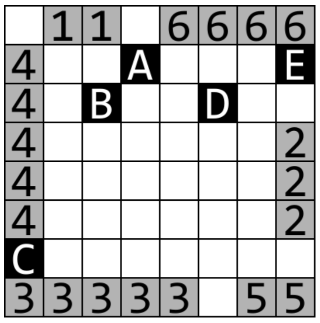
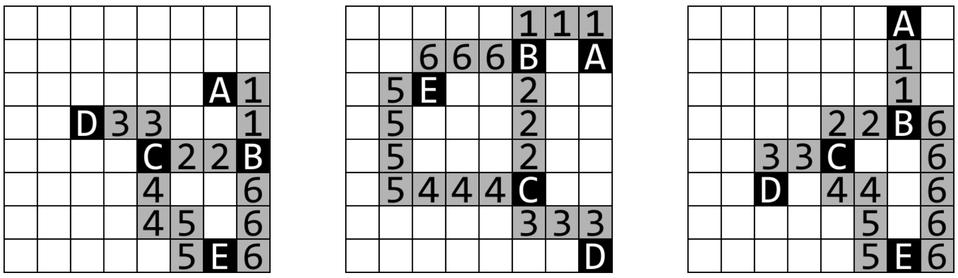


Figure 3.2.

```
% ensure route start and end points are the actual ends
1 {tileInRoute (R, (X,Y)) :
    contains((X, Y), R),
    placeStartTile(R, (XS, YS)),
    placeEndTile(R, (XE, YE)),
    |XS - XE| == |XS - X| + |XE - X|,
    |YS - YE| == |YS - Y| + |YE - Y|
} :-
    contains((X, Y), R), route(R).

% ensure connection
noConnection(E1, E2, T1, T2) :-
    connection(E1, E2),
    placeEndTile(E1, T1),
    placeStartTile(E2, T2),
    not neighbors(T1, T2).
:- not 0 {noConnection(_,_,_,_)} 0.
```

With these last points ensured, the solver can find working world maps, as seen in figure 3.3. The use of a logic solver additionally allows for explicit user input, as long as it is solvable with the other rules. If needed, the user could e.g. define that the length of Route 6 has to be 5 (`routeLength(route6,5).`). This statement would not stand in conflict with the actual rule for the route length (`1 {routeLength(R, (2..5))} 1 :- route(R).`), since this only ensures, that there is exactly one route length per route, which is here given by the user. An example is seen in figure 3.3c. Of course all other parameters can be set in accordance with the requirements for the world, too. Figure 3.3b shows an example in which the minimal route length is set to 3 instead of 2.

From the generated world map single level maps can be generated. Besides the world map layout, some additional information needs to be generated. Since a single level map has no information about other level maps it is connected to, all data regarding the transitions between level maps is generated as part of the world map. This is mainly the offset and size of the transitions at each edge and the dimensions of single fields in the grid. Both could be set to a fixed standard, but this bears the risk of unnatural repetition and patterns, that can be seen by the user and reduce the immersion. Further details to this are given in chapter 3.5.



(a) (b) (c)

Figure 3.3.: Three examples of generated world maps.

3.5. Level Generation

As the next step, the actual tile maps have to be generated. In this context, they are called levels or level maps interchangeably. While the algorithms to generate particular level maps are very different, all level maps share a set of properties.

Like the world map, the base structure of the tile maps is a regular orthogonal grid. One square field (also called *tile*) in the grid is defined via cartesian coordinates. Other than on the world map, a lot of the data associated with a level is not sparse, but mandatory for every tile. Therefore it is beneficial to implement a data structure to manage data on a 2D grid. For this work, the generic class `DataMap<T>` was written. It functions as a wrapper class to a 2-dimensional array, that implements commodities like access via coordinates, iterating over the whole map or providing information about neighboring tiles.

Level maps can be distinguished into two groups: independent maps and overworld maps. Independent maps have no direct connection to other maps and thus don't have to fit given layouts. The player reaches such maps via a teleport functionality. An example is the entrance to a house. The moment the player walks into the door of the house from the outside, he is teleported to the inside map of the house. The level map of the interior of a house doesn't need to be exactly the same size as the house has on the outside. This applies mostly to indoor levels like caves or the indoors of buildings.

Overworld maps, on the other hand, are connected and while generating overworld maps one has to make sure that the seams between maps are correct and not noticeable. To achieve this several values are saved. To mask the regular grid structure a little, every x and y value on the world map is associated with a random size value. These values can be represented as functions $width(x)$ and $height(y)$, where $width(x)$ is the width of a given x coordinate and $height(y)$ the height of a given y coordinate. The dimensions of one world map tile (x, y) are defined as $(width(x), height(y))$.

Also, every connection from one to another tile is defined as an offset between 0 and 1. Each tile stores two offsets, one horizontal and one vertical. These again can be represented as functions $offsetH(x,y)$ and $offsetV(x,y)$, with $offestH(x,y)$ being the offset for a connection in horizontal direction (along the x-Axis), not the horizontal offset of a connection in vertical direction (and $offsetV$ respectively). Every level map has a border of $b = 2$ tiles to prevent the player from walking out. The size of the connection between the two world map tiles is defined as $l_{con} = 4$ with a starting offset o_s in positive direction of the respective axis. The level map tiles included in the connection to the neighboring world map tile are thus $[o_s, o_s + 1, ..., o_s + l_{con} - 1]$. As an example, let the world map tiles $(1,1)$ and $(2,1)$ be connected, but not part of the same place. The connection is in a horizontal direction. The height of the cells is defined by $height(1)$, which is here assumed to be 20. Now $2 \times b$ has to be subtracted from the height, since the connection has to be between the borders of a level map. The length of the opening between the world map tiles l_{con} has to be subtracted from the remaining height to get the number of possibilities to position the traversable tiles along the edge of the level map. In this case it would be:

$$height(1) - (2 \times b) - l_{con} = 20 - (2 \times 2) - 4 = 12$$

The actual offset o_s of the connection is defined by the offset for horizontal connections of $offsetH$ the world map tile $(1,1)$. Assuming $offsetH(1,1)$ is 0.37, the following offset can be computed:

$$b + \lceil offsetH(1,1) \times (height(1) - (2 \times b) - l_{con}) \rceil =$$

$$2 + \lceil 0.37 \times 12 \rceil =$$

$$2 + \lceil 4.4 \rceil = 7$$

An efficient and easy way to store this kind of random value is a lookup table. Every time a value is requested from the lookup table, it checks whether this particular value already exists - generates it, if not - and returns it.

The Level Data Structure

The properties every level map needs are the following:

map type	The type of the map.
dimensions	Size of the map; number of tiles in x and y direction.
tileset	A set of tiles used to compose the map.
tile data	Three layers of references to the tileset.
traversal data	Traversal information for each tile.
entity data	A list of all entities on the map.
trigger data	A list of all triggers on the map.
restrictions	A list of requirements the map has to meet.

Tilesets

Though the basic layout of the tileset is defined by the map type, the tileset itself can be randomized. The easiest way to do this is to provide different tilesets with the same layout and randomly choose one. In figure 3.4 four tilesets and example level maps are shown taken from *Pokémon FireRed & LeafGreen Version* [Gam04].

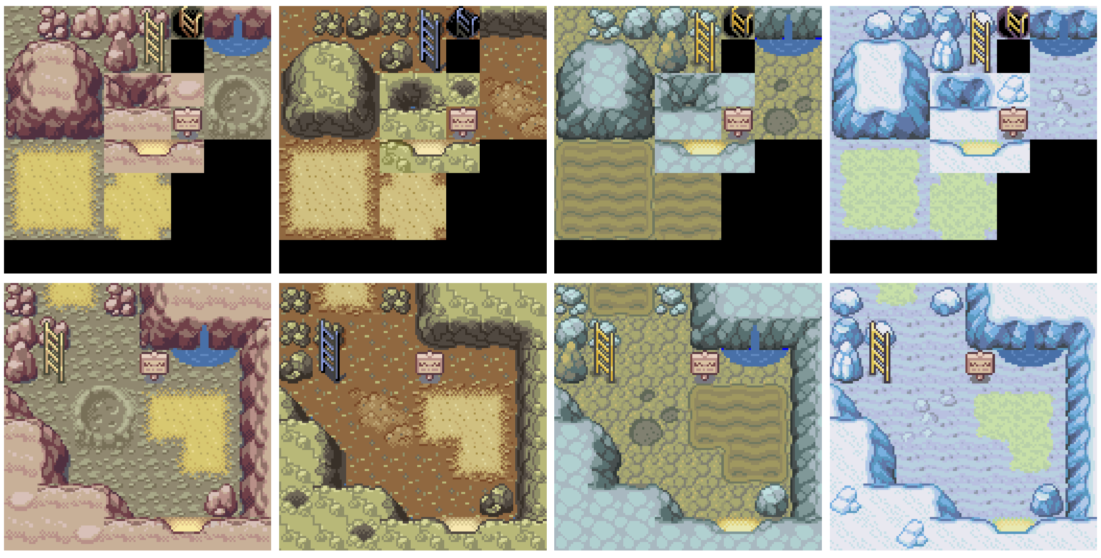

(a) *Mt. Moon* (b) *Diglet's Cave* (c) *Rock Tunnel* (d) *Seafoam Islands*

Figure 3.4.: Example tilesets from *Pokémon FireRed & LeafGreen Version* [Gam04].

A way to randomize tilesets even more is to construct them from single sub-tilesets. E.g. select one of several rock tilesets, tree tilesets, and ground tilesets. While dungeon tilesets are completely independent, the overworld maps are all connected and need to look coherent. To achieve that, some parts of the overworld tileset are fixed, while others, like details, are free for every map. To further randomize the tilesets in use, the color palettes can be generated, too.

Dimensions

The dimensions of a map depend on various factors. If the level is a place on the world map, the dimensions are defined by the grid tiles the place occupies on the world map. So the dimensions of a level that occupies coordinates $(2,4)$ and $(3,4)$ on the world map would be $(width(2) + width(3), height(4))$. For the Pokémon example a suitable minimum value for world map tile dimensions is 16, a good maximum 24.

If a level map is not resembling a place on the world map, e.g. the inside of a building or a cave, the dimensions can be chosen randomly. A guidance is given by the map type. For example the inside of a normal house is rather small, e.g. around $6-12$ in width and height. For dungeons the size is much bigger, e.g. between 30 and 50 tiles in width and height.

In this example, the levels are restricted to be rectangular. This was also done in the original Pokémon games to save memory since a non-rectangular shape would lead to empty spaces in the grid structure and thus to wasted memory. The system could be adapted to work with arbitrary combinations of grid tiles, like L-shapes.

Tile Data

The tile data defines what tiles are at every position of the level grid. This works with simple integers pointing to the respective position in the tileset. One level stores three layers of tile data, that are rendered on top of each other. Layer 0 is used for the floor or ground, layer 1 for obstacles up to the height of the player character and layer 2 for parts of obstacles that are in front of the player but through the perspective of the viewer on the same tile as the player. The player character and other entities are displayed between level 1 and 2. Figure 3.5 shows a layer-wise build up of an example map.

Traversal Data

The traversal data determines where an entity (including the player character) can walk. Besides *non-traversable* tiles, there are several traversable states. Traversable tiles are sorted into layers with respective states *traversable0*, *traversable1*, and so on. These are

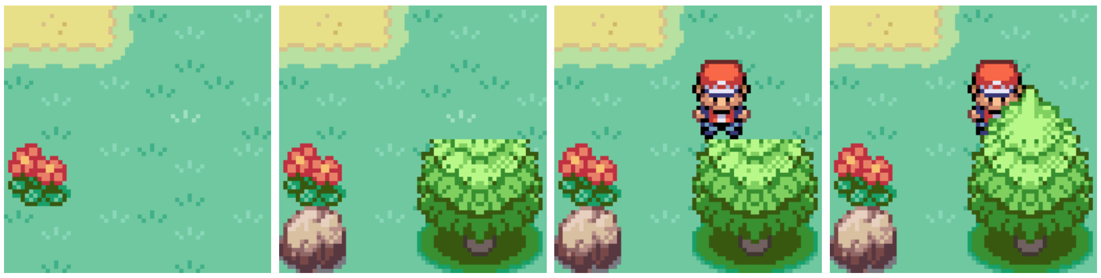


Figure 3.5.: Tile data layers

used if two tiles next to each other are traversable, but the player can not walk from one to the other, because they are on different heights. An example can be seen in figure 3.6. To traverse from one to another traversal level a *transition* is used. Another example for traversal types is *water*. It is not traversable on foot, but can be traversed with the help of certain Pokémon. In a matrix structure connections between every traversal type are defined. For the traversal types explained above the matrix would be:

		1	2	3	4	5
non-traversable	1	✗	✗	✗	✗	✗
traversable0	2		✓	✗	✓	✓
traversable1	3			✓	✓	✗
transition	4				✓	✓
water	5					✓

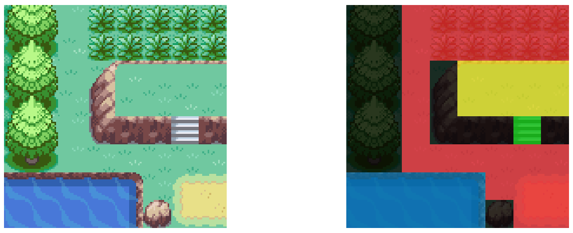


Figure 3.6.: Traversal data example; black: *non-traversable*; blue: *water*; green: *transition*; red: *traversable0*; yellow: *traversable1*;

Entity and Trigger Data

Entities and triggers are both placed on tiles on the map. Since there are few entities and triggers on a level map compared to the number of tiles, both are stored in separate lookup tables to save memory. The information stored in these lookup tables are pointers to the respective entity or trigger in the database.

3.5.1. Generating a Route

One of the most used map types in a Pokémon game is the *Route*. Routes are overworld levels that connect places of interest, e.g. cities. The player spends a lot of time on routes during the game, not only to reach the next point of interest, but also to catch wild Pokémon or fight NPC trainers to train his own Pokémon. The given example is a route which occupies world map tiles $(0,0) - (2,0)$ with dimensions 60×20, an edge of 2 tiles and entrances given from world map tile $(0,-1)$ with offset 6 and to world map tile $(3,0)$ with offset 12.

Traversal Data

The first step in Route generation is the traversal data. Since routes are not meant to be a puzzle-like challenge, like e.g. caves, a maze algorithm with long winding dead-ends as a base for a route seems not fit. Instead, a kind of recursive subdivision is used to generate the base traversal data. The algorithm searches for the largest free rectangle on the map and subdivides it. In this step, the chance of a subdivision along the axis in which the rectangle has a larger dimension is more likely. Other than in the recursive subdivision algorithm explained in section 2.2, this algorithm adds a wall of a certain thickness. The maximum thickness is selected in a way, that on both sides a minimum of space remains (in this example 4 units). The passages through the walls always are at the edges of the rectangle. Another way to describe this process would be: A random rectangle is added into the larger free one in a way that it shares one side with the larger rectangle but has on all other three sides a minimum distance to the larger rectangle. After each added rectangle a flood-fill algorithm checks if the player is still able to reach every entrance from every other entrance. If that is not the case, the inserted rectangle is deleted and the step is repeated. An example can be seen in figure 3.7 (black = *non-traversable*; white/beige = *traversable*).

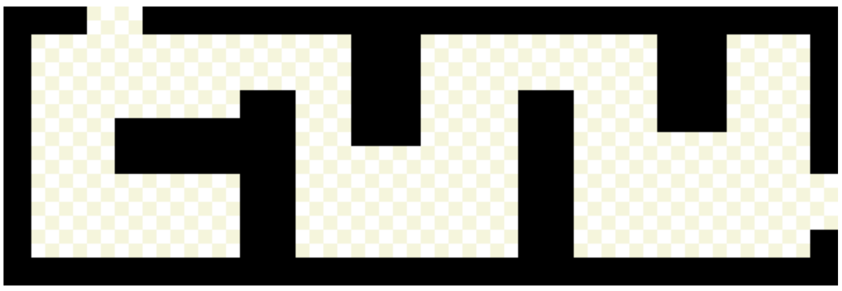


Figure 3.7.: Traversal data after adjusted recursive subdivision algorithm.

As a next (optional) step, water can be added with the same algorithm. If the route is at a point in the storyline, where the player has the ability to swim with a Pokémon (use attack *surf* outside of battle), no traversal checks have to be performed for added waters. Another method could be to use a noise map to determine the places of water. Both examples can be seen in figure 3.8.

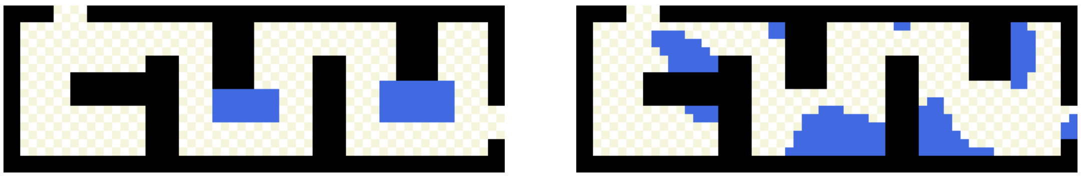


Figure 3.8.: Added bodies of water (blue) as walls of a recursive subdivision (a) or with Perlin noise (b).

In the next step, a cellular automaton is used to mask the rectangular nature of the walls. It has a random component and is not convergent. For every *traversable* cell it counts the number of *non-traversable* cells in its Moore-Neighborhood, gives the current cell a probability to change to *non-traversable* and decides randomly with the given probability whether the cell actually is changed from *traversable* to *non-traversable*. Again a traversal test in form of a flood-fill is performed after every change to ensure the level is traversable.

Depending on how ofter the cellular automaton traverses the map the outcome is more or less subtle. Examples can be seen in figure 3.9. The following probability p depending on the number of *non-traversable* neighbors n was used by the cellular automaton:

$$p = \begin{cases} 0\% & n \leq 2 \\ 2\% & n = 3 \\ 10\% & n = 4 \\ 20\% & n = 5 \\ 70\% & n = 6 \\ 100\% & n \geq 7 \end{cases}$$

Now that the traversal data is generated, high grass is added. In the Pokémon games, wild Pokémon hide in high grass. The player can encounter such wild Pokémon with a certain chance every step he takes in wild grass. If a wild Pokémon appears, the player has to fight it. This mechanism is a significant part of making the traversal of a route challenging to the player. Therefore the amount of high grass determines the difficulty of a route.

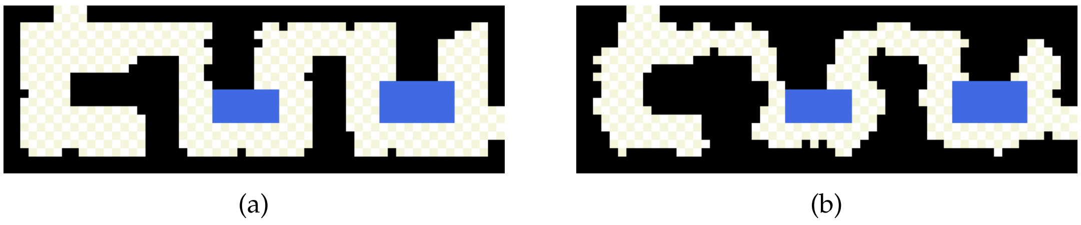


Figure 3.9.: Effect of the cellular automaton after 5 (a) and 20 (b) iterations.

Taking the original games as inspiration, the fields of high grass are usually rectangular structures and often enclosed between *non-traversable* tiles on at least two sides. To achieve a similar effect, random points are chosen on the level map. From those points the next *non-traversable* tile in all four directions is looked up. The rectangle containing the starting point and the closest three walls is filled with high grass.

Again, this is not the only method of adding grass to a level map. Other methods could be a noise map or a cellular automaton that lets grass grow. Examples for the explained method and Perlin noise can be seen in figure 3.10.

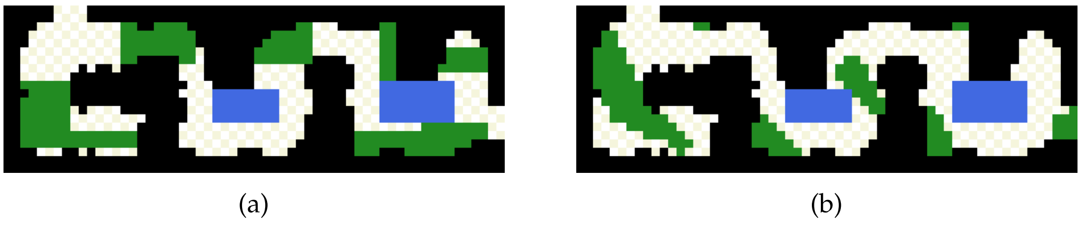


Figure 3.10.: Added high grass (green) with a custom method (a) and Perlin noise (b).

Now, all necessary other restrictions can be implemented. These are most often obstacles that hinder the player from traversing the route before he has finished the required story events. This could be natural obstacles that can be overcome with a special Pokémon attack, like water (overcome by attack *surf*) or a bush (destroyed by attack *cut*), or simply an NPC that won't let the player pass before a certain event is finished, as e.g. the old man in *Viridian City* in *Pokémon Red & Blue Version* [Gam99], who won't let the player player pass until he got his coffee, which incidentally is the case after a certain unrelated event is finished.

Tile Data

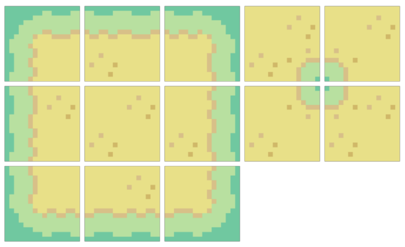


Figure 3.11.: the sand tileset

Next, the tile map is generated. Many parts of a level map are constructed from Wang tiles [Wan61], including dirt, water, and rock formations. One challenge when working with Wang tilesets is, that every edge of a tile has to fit the edge of the neighboring tile. Often not every arbitrary combination of data on a grid can be represented by every tileset. A lot of tilesets have no representation for a single tile surrounded by no further tiles of the same type. In some cases, e.g. the rocks, there is a substitute tile for a single element, but for the rest, another solution to deal with these cases has to be found. In all these cases a binary data set (e.g. dirt / no dirt) has to be converted in a way that every cell can be represented by the tileset.

In the example of the dirt, every neighbor in a cell's Moore-Neighborhood is given a value 1 or 0 for *dirt* or *no dirt*. These bit values can be interpreted as a binary number. It needs to be defined which numbers are possible combinations and which are not. Additionally, some cells have to be *no dirt*, dependent on different other data (e.g. the ground under high grass is always grass). A completely valid distribution of *dirt* and *no dirt* tiles can be generated by a cellular automaton. Every cell that is marked as *dirt* but has an illegal number or has to be *no dirt* to fulfill other dependencies, is set to *no dirt*. This is done until the cellular automaton converges.

To add variance a grammar or cellular automaton can be used to replace combinations of tiles with alternative combinations that have the same traversal data and thus are purely cosmetic. Figure 3.12 shows a tile map based on the traversal data seen in figure 3.10a. In it, grass was replaced with either dirt, based on a noise map or flowers. Inspired by the original Pokémon games the flowers are arranged in a checkerboard pattern combined with a noise map.

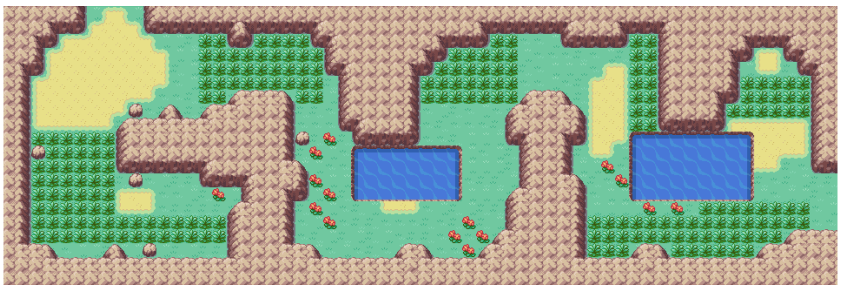


Figure 3.12.: Example tile map of the finished route.

Of course the presented techniques are not the only possibility to generate levels. Through combination of techniques presented in chapter 2 many different styles can be achieved.

Another example can be seen in figure 3.13. It shows a 60 × 32 cave based on a maze generated by Kruskal's algorithm. The entrances are in the top left and bottom right corner and a given restriction was to only be possible to cross the cave by swimming. It used the *Mt. Moon* tileset with two different single tile stones, noise based sand fields and randomly inserted craters.

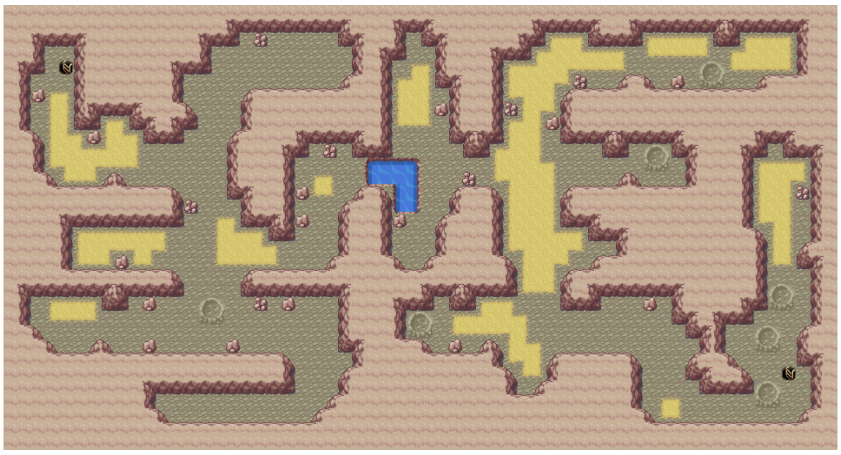


Figure 3.13.: Example tile map of a cave.

Entities & Triggers

On every map entities such as NPCs or items need to be placed as well as triggers. As with the rest of the map generation, the placement of NPCs varies from level type to level type. At first, the triggers and necessary entities need to be set. These are e.g. a *teleport* trigger when the player steps onto a tile like a ladder to the next floor or an NPC that needs to be interacted with defined by a story event. The positioning of these entities and triggers is not very complicated since they are most often fixed by the level map layout. The *teleport* triggers e.g. need to be at exactly the tiles on which the ladders are, the NPC to interact with is at the farthest dead end in the maze, etc.

Other entities can be set by fitness functions for the tiles that determine the possibility of a certain entity being spawned on that tile. The selection of a tile according to the fitness function can vary, depending on the desired outcome.

The first possibility is to just select the tile with the highest fitness value. This, however, can lead to a very similar distribution on every level map and leaves no room for interesting outcomes. Another possibility is to simply select a threshold, e.g. 70% of the maximum fitness value, and select a tile randomly from all the tiles with a fitness value greater than that threshold. A third way is to directly apply the fitness values as a possibility for that tile to be chosen. This can be done with stochastic universal sampling (SUS) [Bak87]. To do that for n tiles $t_0, ..., t_{n-1}$ with fitness values $f(t_0), ..., f(t_{n-1})$ the sum of all fitness values F has to be computed: $F = \sum_{i=0}^{n-1} f(t_i)$. Now the algorithm selects a random number r in the range of $[0, F]$. This number is used as a pointer on a number line on which every tile occupies space according to its fitness value. To find the respective tile the algorithm walks through the tiles in a fixed order and adds the fitness value of each tile to a counter. The tile at which the counter exceeds r is the selected one. With SUS every tile can be selected as long as its fitness value is larger than 0, which can lead to interesting outcomes and help to mask certain patterns on the one hand, but also may cause problems on the other hand. SUS can also be combined with a threshold fitness so that the set to choose from only contains tiles above a certain threshold.

Since in most cases a roughly equal distribution is desired, the fitness values of tiles can be adjusted after each spawned entity. Often the fitness values of tiles in proximity to other entities need to be decreased. Though in some cases the Manhattan or Euclidean distance is the right choice to compute such proximity, in a lot of cases when dealing with level maps the more interesting distance is the length of the shortest path between two tiles according to the traversal data, which here will be called *walking distance*. One way to find (one of) the shortest path(s) between two points on a graph is Dijkstra's algorithm [Dij59]. In this case, often all tiles with a certain walking distance need to be found. For that, the data gathered by Dijkstra's algorithm can be used, which works like a flood-fill algorithm on unweighted graphs, outgoing from the tile in question and spreading according to the traversal data.

On a route one typically finds a few trainers standing around and a few items laying on the ground. The gameplay mechanics of trainers works as follows: If the player walks into their line of sight, which is usually around four tiles far, the trainer challenges the player to a Pokémon battle the player can't refuse. That is why trainers are usually positioned outside of high grass and in a way, that it is hard or even impossible to not walk into their line of sight. Some trainers stand still and always look in one direction, while others change their direction randomly or in patterns. Items usually are placed in dead-ends of the level map, far of the main path. They function as a reward for players exploring the area and counteract the frustration of a dead-end.

To apply the explained techniques to the positioning of trainers, a suitable fitness function needs to be found. Factors influencing this value are e.g. the proximity to walls, the presence of high grass, and how well the player could navigate around the trainer to avoid a battle. Then trainers are generated and placed until the number matches the desired difficulty of the route. The more trainers a route occupies, the more difficult it is for the player to traverse it. So the number of trainers can indirectly be determined by the position in the storyline.

The fitness function for items should mainly be based on dead-ends. If the level is based on a maze, it is easy to determine dead-ends with one of the algorithms in chapter 2.2. If the level map is not based on a maze, the dead-end like parts have to be determined in other ways. One would be to use the distance to one of the shortest paths between all entrances. This, again, can be done with Dijkstra's algorithm [Dij59]. An adapted flood-fill starting from the paths determines the shortest distance between every tile on the level map and the paths. The distance itself can be used as fitness value, or all tiles that are above a certain threshold distance to the main paths can be seen as dead-ends and get the same fitness value.

If dialogue is needed, it can be generated with adjusted Forlog grammars. The example grammar in appendix B produces sentences A trainer can say before battle. The grammar is adjusted to the following facts: *Adam* is the name of the closest gym leader, the *emerald forest* is close to the trainers location, in the *emerald forest* live wild *Venonat*, *Tangela* and *Scyther*, the trainer trains bug type Pokémon and he possesses a *Metapod* and a *Kakuna*. These are a few example outputs:

Adam is so great! I tried to fight him, but I am too weak.

Look how cool my Kakuna is.

My bug Pokémon are going to crush you!

I heard you can find Scyther in emerald forest.

I quite like my grandma's cake.

Wild Monsters

As the last step, the Monsters encountered in the wild need to be set. There are several ways to encounter wild Pokémon, and every one has its own set of Pokémon. The first way is to encounter them by walking. On overworld maps, wild Pokémon are found in high grass, while in dungeons, like caves, wild Pokémon can appear everywhere. The second way of encountering Pokémon is like first one by moving, but in this case by swimming (using attack *surf*) on water. The third way is by using a fishing rod. For all three types suitable Pokémon have to be selected. These are randomly chosen out of the ones, that fulfill certain criteria. First, the level range in which the Pokémon occurs has to overlap with the level range wild Pokémon should have on the respective map. Second, the Pokémon should thematically fit in the level map and the way they are encountered. This is done by classifying Pokémon into several classes. Every Pokémon type has its own class, but there are also groups like *in water*, *near water*, *cave* etc. One Pokémon can have several classes. This is necessary since the type itself gives not enough information to select fitting Pokémon. One would e.g. expect a *Water* type Pokémon when using the fishing rod, as well as when encountering a wild Pokémon on a beach. The seagull Pokémon *Wingull* and the goldfish Pokémon *Goldeen* are both of type *Water*, but it would be as odd to find a fish in high grass as it would be to catch a bird with a fishing rod.

4. Conclusion & Future Work

Procedural content generation for games holds the potential to create high amounts of content for players, though it is most often only used for the generation of single levels. This work presents not only many different basic techniques, but also how they can be combined. In the examples given an approach to generate a complete world from the world map layout down to details like single dialogues is presented. The world, as well as all the single levels, are generated to embed a storyline, which is generated beforehand under certain constraints.

To actually write a complete generator, all examples would need to be extended to hold more than a few options. E.g. level maps for cities or forests are generated differently than level maps for routes and caves, though they use the same ideas, techniques, and approaches. For proving this concept, these generators were not written for this work.

In many examples alternatives to the taken approach are explained. This shows, that the presented approach is not the single and only solution to this generation problem. Depending on the desired outcome, different techniques can be combined in various ways. The overall structure to generate complex data sets step by step without generating inconsistencies, can be applied to all projects of that kind. Answer set programming is especially helpful to avoid inconsistencies while generating content.

Compared to the solutions created with this approach, hand-crafted stories and worlds show more love for detail. Especially if the storyline gets more complex than the storyline of a classical Pokémon game. This has to be taken into consideration when planning a project. Nonetheless, generated worlds bare high replayability for role-playing games. As a rule of thumb, a game, that sells a story, is not suited to be generated with this approach. A game, that sells an adventure is suited to have its story and world generated with this approach.

This approach can also be used during production of a game, to e.g. generate several storylines and refine them by hand before releasing them. In general, any part of a procedural generator can be used as a tool during development. E.g. could a maze generator be used as basis for a level designer, who does all following production steps by hand. The possibility to generate only the smallest part of a project up to the whole world makes PCG a very powerful tool.

In conclusion, the usage of PCG in role-playing games can drastically reduce the time and money used for handcrafting worlds and stories once the generators are written. The creation of a generator can be a time-consuming task in itself. Whether it is worthwhile depends on how many different generated outputs are needed and whether the creation by hand would take longer than the creation of a generator.

Besides the given approach, this work hopefully shows, that practically everything, no matter how extensive, can be generated. It just has to be broken down into small pieces, that are easy to manage for a computer.

List of Figures

Bibliography

[Amp14] Amplitude Studios. *Endless Legend*. [PC Software]. 2014.

[Bak87] J. E. Baker. "Reducing Bias and Inefficiency in the Selection Algorithm." In: *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. Cambridge, Massachusetts, USA: L. Erlbaum Associates Inc., 1987, pp. 14–21. ISBN: 0-8058-0158-8.

[Bar03] C. Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. New York, NY, USA: Cambridge University Press, 2003. ISBN: 0521818028.

[Bar96] R. Bartle. "Hearts, clubs, diamonds, spades: Players who suit MUDs." In: *Journal of MUD research* 1 (1996).

[Bey16] T. Beyer. *Forlog*. [accessed 19.11.2017]. 2016. URL: `http://deosis.de/`.

[Bli12] Blizzard Entertainment. *Diablo III*. [PC Software]. 2012.

[Buc15] J. Buck. *Mazes for Programmers: Code Your Own Twisty Little Passages*. 2015. ISBN: 978-1680500554.

[Dab] Dabomstew. *Universal Pokemon Game Randomizer*. [accessed 6.11.2017]. URL: `http://pokehacks.dabomstew.com/randomizer/`.

[Dij59] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs." In: *Numer. Math.* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0029-599X. DOI: `10.1007/BF01386390`.

[Fir10] Firaxis Games. *Sid Meier's Civilization V*. [PC Software]. 2010.

[Fir14] Firaxis Games. *Sid Meier's Civilization: Beyond Earth*. [PC Software]. 2014.

[Fir16] Firaxis Games. *Sid Meier's Civilization 6*. [PC Software]. 2016.

[Gam00] Game Freak. *Pokémon Yellow Version*. [Game Boy Color]. 2000.

[Gam01a] Game Freak. *Pokémon Crystal Version*. [Game Boy Color]. 2001.

[Gam01b] Game Freak. *Pokémon Gold & Silver Versions*. [Game Boy Color]. 2001.

[Gam04] Game Freak. *Pokémon FireRed & LeafGreen Versions*. [Game Boy Advance]. 2004.

[Gam05] Game Freak. *Pokémon Emerald Version*. [Game Boy Advance]. 2005.

[Gam99] Game Freak. *Pokémon Red & Blue Versions*. [Game Boy]. 1999.

[Gar70] M. Gardner. "Mathematical Games – The fantastic combinations of John Conway's new solitaire game "life"." In: *Scientific American* (1970).

[Geb+11] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. "Potassco: The Potsdam Answer Set Solving Collection." In: *AI Communications* 24.2 (2011), pp. 107–124.

[Hel16] Hello Games. *No Man's Sky*. [PC Software]. 2016.

[Kit16] Kitfox Games. *Moon Hunters*. [PC Software]. 2016.

[Kru56] J. B. Kruskal. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem." In: *Proceedings of the American Mathematical Society* (1956).

[Moj11] Mojang. *Minecraft*. [PC Software]. 2011.

[NS16] M. J. Nelson and A. M. Smith. "ASP with applications to mazes and levels." In: *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Ed. by N. Shaker, J. Togelius, and M. J. Nelson. Springer, 2016.

[Per01] K. Perlin. "Chapter 2: Noise Hardware." In: 2001.

[Per85] K. Perlin. "An Image Synthesizer." In: *SIGGRAPH Comput. Graph.* 19.3 (July 1985), pp. 287–296. ISSN: 0097-8930. DOI: `10.1145/325165.325247`.

[Pri57] R. C. Prim. "Shortest Connection Networks and Some Generalizations." In: *The Bell System Technical Journal* 36 (1957).

[Pul15] W. D. Pullen. *Maze Classification*. [Online; accessed 7-November-2017]. 2015. URL: `http://www.astrolog.org/labyrnth/algrithm.htm`.

[Rea00] Reality Pump Studios. *Earth 2150*. [PC Software]. 2000.

[STN16] N. Shaker, J. Togelius, and M. J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

[Tar] R. E. Tarjan. "6. Minimum Spanning Trees." In: *Data Structures and Network Algorithms*, pp. 71–83. DOI: `10.1137/1.9781611970265.ch6`. eprint: `http://epubs.siam.org/doi/pdf/10.1137/1.9781611970265.ch6`.

[Tog+11] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. "Search-based Procedural Content Generation." In: *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)* 3 (2011).

[TSD16] J. Togelius, N. Shaker, and J. Dormans. "Grammars and L-systems with applications to vegetation and levels." In: *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Ed. by N. Shaker, J. Togelius, and M. J. Nelson. Springer, 2016.

[Tul08] A. H. Tulleken. "Poisson Disk Sampling." In: *Dev.Mag* (21 2008).

[von51] J. von Neumann. "The General and Logical Theory of Automata." In: *Cerebral Mechanisms in Behavior – The Hixon Symposium*. Ed. by L. A. Jeffress. John Wiley & Sons, 1951, pp. 1–31.

[Wan61] H. Wang. "Proving theorems by pattern recognition — II." In: *The Bell System Technical Journal* 40.3 (Jan. 1961), pp. 1–41. DOI: `10.1002/j.1538-7305.1961.tb03975.x`.

[Wik17] Wikipedia. *Maze — Wikipedia, The Free Encyclopedia*. [Online; accessed 7-November-2017]. 2017. URL: `https://en.wikipedia.org/w/index.php?title=Maze&oldid=807338595`.

[Wor96] S. Worley. "A Cellular Texture Basis Function." In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pp. 291–294. ISBN: 0-89791-746-4. DOI: `10.1145/237170.237267`.

Appendix

A. Answer Set to Generate a Graph from a Storyline

This answer set is used to generate the graph structure from the storyline.

```
#const routeNr = 6.
#const minCycles = 1.
param("routeNr", routeNr).
param("minCycles", minCycles).

city(azureCity).
city(berryTown).
city(crimsonCity).
city(denimCity).
city(ebonyTown).

startCity(azureCity).

route(1..routeNr).

relation(azureCity, berryTown).
relation(berryTown, crimsonCity).
relation(crimsonCity, denimCity).
relation(denimCity, ebonyTown).

relation(P1, P2) :- place(P), relation(P1, P), relation(P, P2).

place(X) :- city(X).
place(X) :- route(X).

:- connection(P1, P2), connection(P2, P1).
:- connection(P1, P2), city(P1), city(P2).
```

```
1 {connection(C, P) : place(P); connection(P, C) : place(P)} 3 :- city(C).
1 {connection(C, P) : place(P); connection(P, C) : place(P)} 2 :- startCity(C).
1 {connection(P, C) : place(P)} :- city(C), not startCity(C).
1 {connection(C, P) : place(P)} :- startCity(C).

1 {connection(R,P) : place(P), R!=P} 1 :- route(R).
1 {connection(P,R) : place(P), R!=P} 1 :- route(R).

cityConnection(C1, C2) :-
    C1 != C2,
    city(C1),
    city(C2),
    route(R),
    connection(C1, R),
    connection(R, C2).

cityConnection(C1, C2) :-
    C1 != C2,
    city(C1),
    city(C2),
    route(R1),
    route(R2),
    connection(C1, R1),
    connection(R1, R2),
    connection(R2, C2).

indCityConnection(C1, C2) :-
    C1 != C2,
    city(C1),
    city(C2),
    city(CM),
    1 {cityConnection(C1, CM); indCityConnection(C1, CM)},
    1 {cityConnection(CM, C2); indCityConnection(CM, C2)}.
```

```
relCityConnection(C1, C2) :-
    C1 != C2,
    city(C1),
    city(C2),
    route(R),
    connection(C1, R),
    connection(R, C2),
    relation(C1, C2).
relCityConnection(C1, C2) :-
    C1 != C2,
    city(C1),
    city(C2),
    route(R1),
    route(R2),
    connection(C1, R1),
    connection(R1, R2),
    connection(R2, C2),
    relation(C1, C2).
relCityConnection(C1, C2) :-
    C1 != C2,
    city(C1),
    city(C2),
    city(CM),
    1 {relCityConnection(C1, CM)},
    1 {relCityConnection(CM, C2)},
    relation(C1, C2).

notConnectedToStart(C):-
    not startCity(C),
    city(C),
    {relCityConnection(azureCity, C)} 0.
:- not 0 {notConnectedToStart(_)} 0.
```

```
illegalConnection(C1, C2) :-
    C1 != C2,
    city(C1),
    city(C2),
    1 {indCityConnection(C1, C2); cityConnection(C1, C2)},
    {relCityConnection(C2, C1)} 0,
    relation(C2, C1).
:- not 0 {illegalConnection(_,_)} 0.

cycle(C1, C2) :-
    C1 != C2,
    city(C1),
    city(C2),
    1 {indCityConnection(C1, C2); cityConnection(C1, C2)},
    1 {indCityConnection(C2, C1); cityConnection(C2, C1)},
    relation(C2, C1).
:- not minCycles {cycle(_,_)}.

doubleRoute(R1, R2) :-
    R1 != R2,
    route(R1),
    route(R2),
    place(P1),
    place(P2),
    connection(P1, R1),
    connection(R1, P2),
    connection(P1, R2),
    connection(R2, P2).
doubleRoute(R1, R2) :-
    R1 != R2,
    route(R1),
    route(R2),
    place(P1),
    place(P2),
    connection(P1, R1),
    connection(R1, P2),
    connection(P2, R2),
    connection(R2, P1).
:- not 0 {doubleRoute(_,_)} 0.
```

```
doubleCityConnection(C1, C2) :-
    cityConnection(C1, C2),
    cityConnection(C2, C1).
:- not 0 {doubleCityConnection(_,_)} 0.
```

B. Example Forlog Grammer

A Forlog grammar to generate example sentences for trainers to say before battle. Rules like the [CHAMP] or [OWN_POKEMON] rule are adjusted according to the circumstances.

```
>[SENTENCE]

SENTENCE
>[X_IS_COOL] [I_WANT_TO_BE_LIKE]
>[MY_POKEMON]
>[LOVE_MY_POKEMON]
>[SMALLTALK]
>[NEAR_LOCATION]

X_IS_COOL
>[CHAMP] is so [COOL]!
>[CHAMP] is the best Trainer!
>[CHAMP]'s Pokémon beat anyone!
>Noone can beat [CHAMP]!

I_WANT_TO_BE_LIKE
>I wish I would be as good as he is.
>I tried to fight him, but I am too weak.
>One day I want to be as good as he is.
>I want to be like him.

COOL
>cool
>great
>strong
```

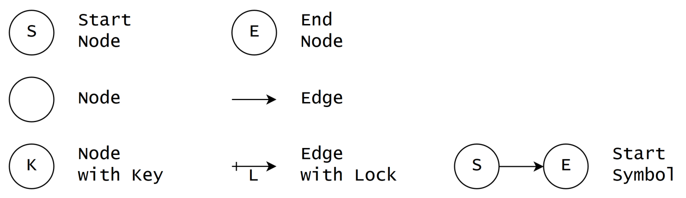
```
NICE
>nice
>beautiful
>great

MY_POKEMON
>My [TYPE] Pokémon are going to crush you!
>My Pokémon are going to crush you!
>Your Pokémon have no chance against my [TYPE] Pokémon.
>Your Pokémon have no chance against mine.
>Prepare to lose!

LOVE_MY_POKEMON
>I love my Pokémon!
>I love my [OWN_POKEMON]. It's the best.
>Look how [COOL] my [OWN_POKEMON] is.
>Look how [NICE] my [OWN_POKEMON] looks.

SMALLTALK
>Isn't the weather [NICE] today?
>I haven't seen you here before.
>Are you new to this area?
>I [LOVE_HATE] [OBJECT].
>I think [OBJECT] is the [BEST_WORST].
>Where are you going?
>Do you like [OBJECT]? I [DO_DONT]!

NEAR_LOCATION
>Have you been to [LOCATION]?
>I heard you can find [NEAR_POKEMON] in [LOCATION].
>I'm on my way to [LOCATION]. I want to catch a [NEAR_POKEMON].
>My [FRIEND] saw a [NEAR_POKEMON] in [LOCATION].

LOVE_HATE
>love
>hate
>quite like
```

```
DO_DONT
>do
>don't

BEST_WORST
>best
>worst

OBJECT
>high grass
>my grandmas cake
>cloudy days

FRIEND
>friend
>cousin
>brother
>sister

CHAMP
>Adam

TYPE
>bug

LOCATION
>emerald forest

NEAR_POKEMON
>Venonat
>Tangela
>Scyther

OWN_POKEMON
>Metapod
>Kakuna
```

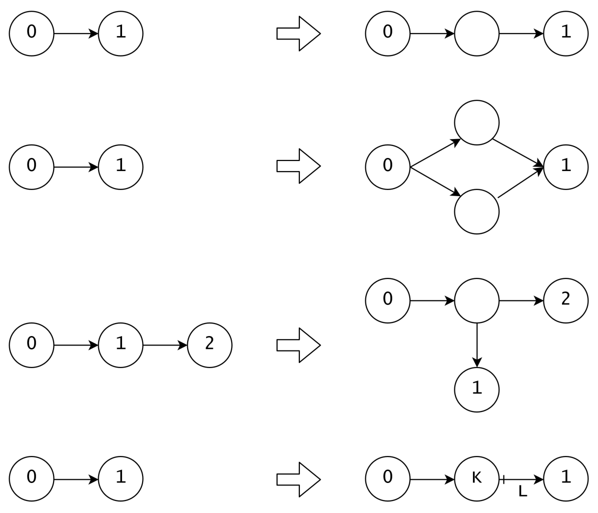
C. Example Graph Grammar

An example for a graph grammar to generate level layouts with lock/key mechanics.

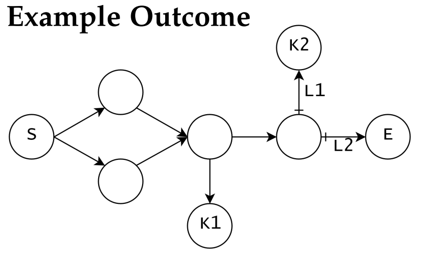
Legend



Rules



Example Outcome



K2
L1
S
L2
E
K1