CHAPTER 9

Cyclic Generation

Dr. Joris Dormans
Ludomotion

There are many different ways of generating dungeons for roguelike games. The most popular method seems to be "drilling out" the dungeon from an arbitrary starting point. In this way, tunnels, corridors, and rooms are added to the expanding level. The big advantage of this method is that all areas are guaranteed to be accessible from the entry point. The big disadvantage, however, is that the structure of the generated dungeon essentially is a branching tree: it will have many dead ends and force the player to track back frequently.

A good example of this type of level generation can be found in *Brogue*. In an interview on Rock, Paper, Shotgun, *Brogue* developer Brian Walker explains that his way of getting around the branching problem is to randomly add doors between rooms in different branches of the tree.* This strategy seems effective enough; the quality of *Brogue*'s level generator is testimony to that. Yet, in this chapter I discuss a different approach I developed while working on *Unexplored*. This approach does not generate trees, but incorporates cycles from the get-go, allowing the generator to incorporate more sophisticated level design patterns.

Unexplored is a traditional roguelike in the sense that you traverse the Dungeon of Doom trying to retrieve the Amulet of Yendor and escape the dungeon with it. The main difference between *Unexplored* and games like *Rogue*, *NetHack*, or *Brogue* is that it is real time. The gameplay is modeled on top-down *Zelda* games, including real-time melee combat and lock-and-key puzzles. This difference in gameplay also means there are

* https://www.rockpapershotgun.com/2015/07/28/how-do-roguelikes-generate-levels/

different constraints on the levels generated for *Unexplored*. Levels tend to be smaller and more focused on a couple of themes (such as the presence of lava, water, or teleporters), and make more use of lock-and-key puzzles.

An example of one such level can be found in Figure 9.1. It illustrates how a combination of just a couple of design patterns can create a level that feels consistent and creates an exploration challenge beyond a series of individual monsters and hazards. In this case, the patterns are a larger lock-and-key cycle, with a large chasm in the middle, with an embedded gambit pattern. These patterns are discussed in more detail in the "Patterns" section.

CYCLES

The idea of using cycles for level generation arose during a research workshop on computational modeling in games,* and was inspired by previous research in architecture, urban planning, and hypertext structure. In the real world, branching trees are rare. In most cities, buildings, and parks, you can go around in circles. Cycles are also very dominant in handcrafted levels. Simply google dungeon maps for tabletop role-playing games and look for cycles in the most interesting ones. In fact, this is one way I harvested many of the patterns discussed below.

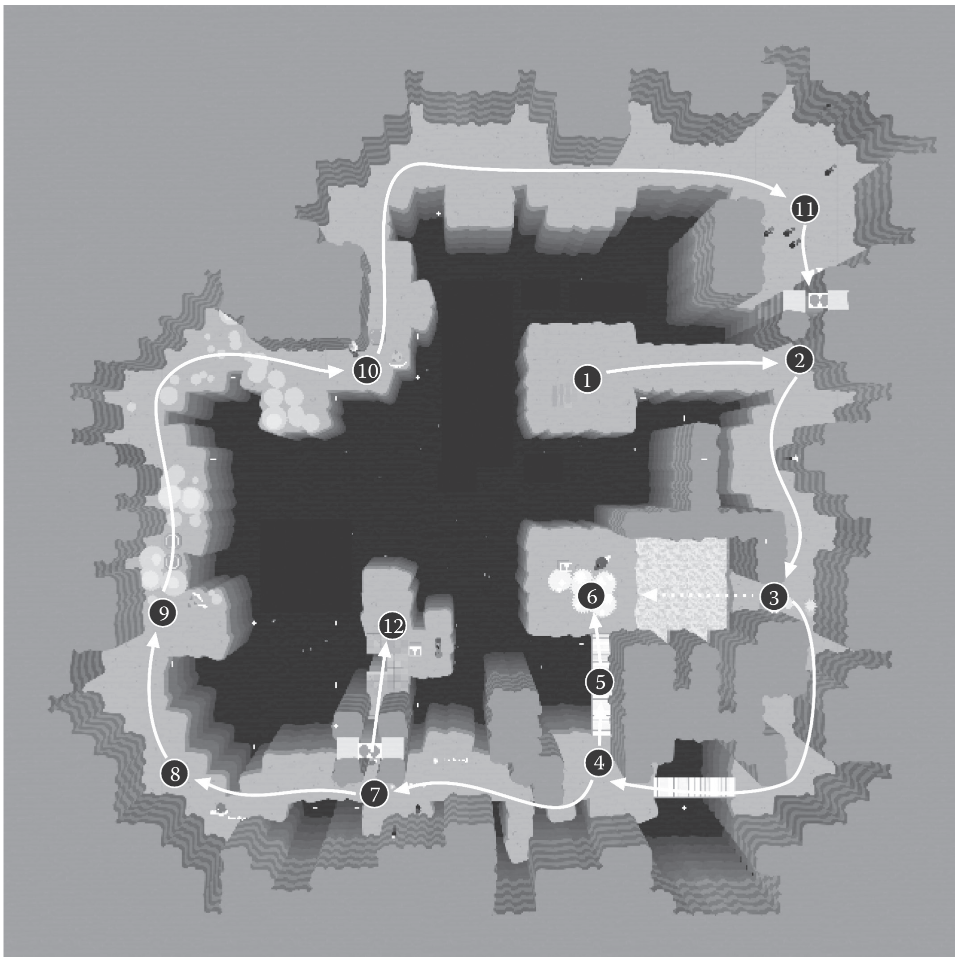
In *Unexplored*, cycles are created by connecting a starting point and a goal by two paths instead of one. Figure 9.2 illustrates how nesting these cycles often is much more effective in creating dungeon levels than simple branching.

There are many different ways of using a cycle. Both paths from the start to goal can offer different challenges to the player. Or maybe if one path is much shorter, it can be much more dangerous, or simply much harder to find. One path may actually only be traversable in one direction, creating a quick route back to the starting point (e.g., after the player has found a key). So far, I have identified a fair number of these types of patterns, which are discussed in more detail in the "Patterns" section.

USING GRAPHS TO EXPRESS CYCLES

Cycles are not easy to work with if your level is represented by a tilemap. In order to get around this problem, *Unexplored* uses graphs to represent level structure during the initial steps of the generation process. It is only halfway through that these graphs are converted to tilemaps.

* http://www.birs.ca/events/2016/5-day-workshops/16w5160

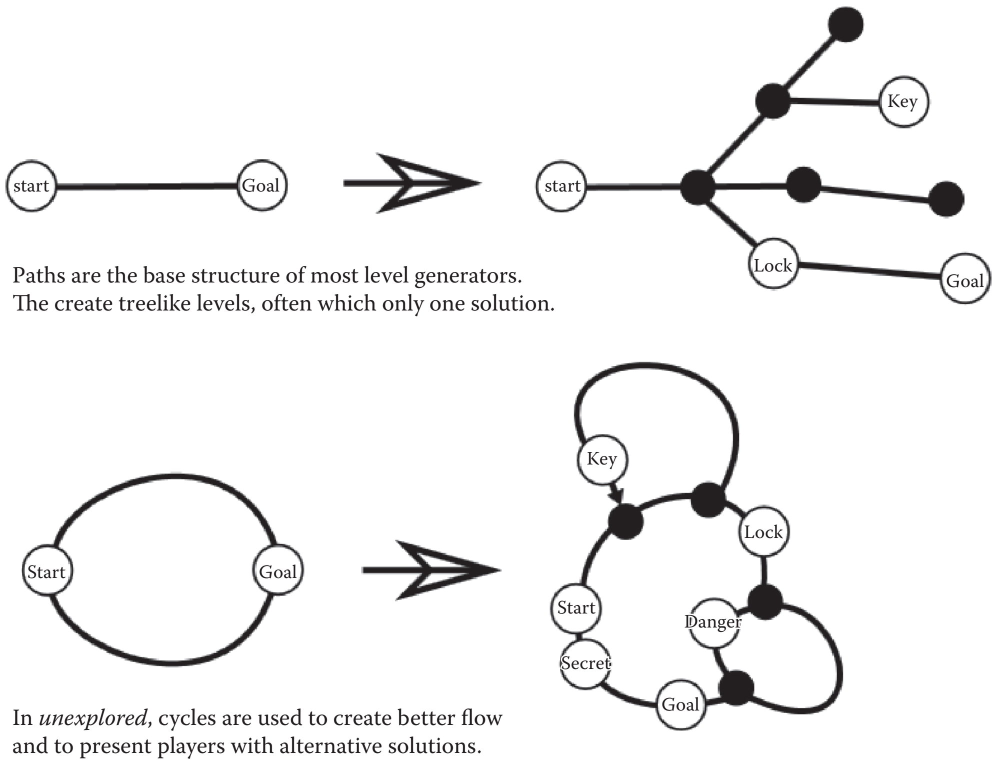


(1) You enter the level at a platform in the center of a huge chasm. (2) The door on the north is barred. The only open route is south. (3) Across the lava, you can spot the platform at 6, but you can not really reach it. (4) You can continue on or check out platform 6, which you might have seen earlier. (5) As you cross the bridge, it collapses. (6) If you make it to the platform, you are trapped there facing a giant rat. The chest contains a scroll of teleportation, which allows you to escape. (7) This door is also barred. (8) The narrow pathway is guarded by a patroling kobold spearman. (9) This section is protected by a sleeping goblin spearman. You can sneak past easily. But did you spot those triggers that set off an alarm? (10) Similar section. A trigger might wake up the goblin bowman. (11) Giant ants protect this area, but it also has two levers operating the doors at 2 and 7. (12) After visiting 11, you can now make your way to this platform and take the staircase down to the next level.

FIGURE 9.1 Pit-level walkthrough.

This process was inspired by the practice of model-driven engineering. In short, model-driven engineering advocates that automated processes use multiple steps to produce the final result. The output of each individual step is a model, preferably a model that makes some sort of sense on its own, that gets passed to the next step for further processing.

In general, when one is generating something as complex as a complete game level, it makes a lot of sense to break down the process into multiple



Paths are the base structure of most level generators.
The create treelike levels, often which only one solution.

In *unexplored*, cycles are used to create better flow
and to present players with alternative solutions.

FIGURE 9.2 Branching vs. nested cycles.

steps. In fact, I assume that all successful level generators do so. However, the advantages of multiple steps get hugely enhanced by using different types of models during the process. *Unexplored* initially uses graphs to represent level structure, as graphs are much better to retain structural information over distances. For example, Figure 9.3 shows one such graph showing connected rooms, containing obstacles, locks, and keys.*

At this stage, all nodes in the graph are deliberately generic. The exact nature and properties of most obstacles, locks, keys, and rooms are defined during later stages.

Unexplored manipulates graphs using transformational graph grammars. These grammars consist of rules that loom for particular patterns in a graph and suggest ways how these graphs can be transformed

* It is important to note that containment in this example is represented as a node embedded in another node. However, this is not different from a node connected with a special containment edge to its parent. In fact, that is exactly how it is implemented.

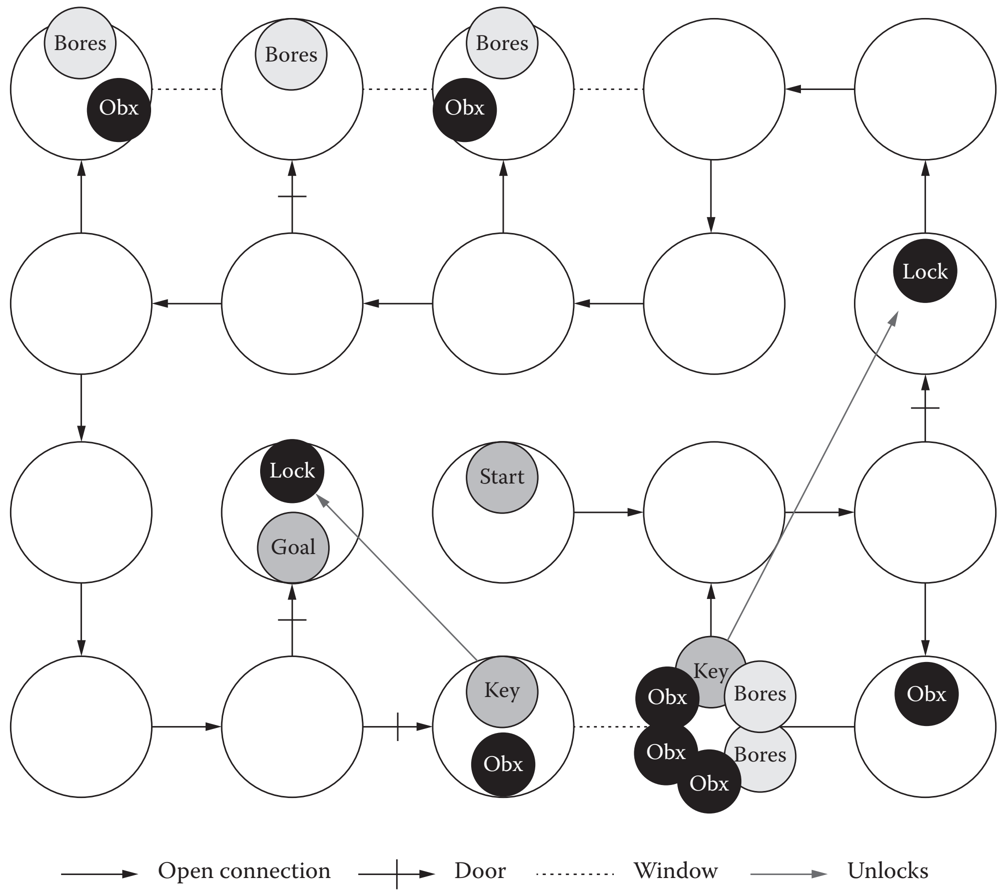


FIGURE 9.3 Graph of a level being generated, with explanations.

by replacing the found pattern with a different pattern. For example, Figure 9.4 illustrates a graph transformation rule that creates a simple cycle, while Figures 9.5 through 9.7 illustrate more rules and the graphs they can generate. It is important to note that the numbers in the rules are used to match up nodes on the left-hand side of the rule (the pattern to be replaced) and the right-hand side of the rule (what the pattern is replaced with).

PATTERNS

Graph transformation rules allow *Unexplored* to implement a number of design patterns. These patterns are common structures found across games. For example, adding a secret shortcut to a long and dangerous level is one such pattern. Each pattern has its own positive and negative effects;

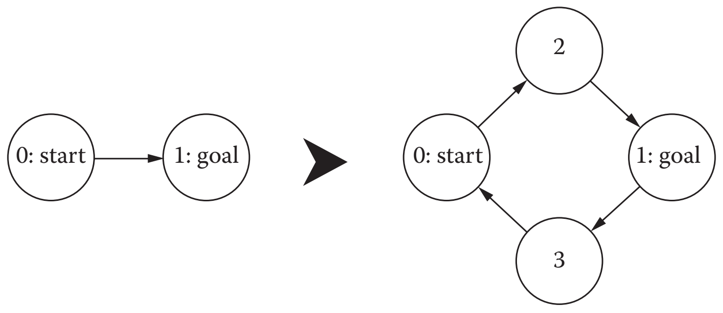


FIGURE 9.4 Graph rule to create a simple graph.

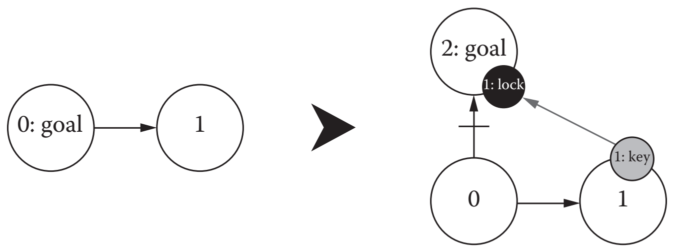


FIGURE 9.5 Graph rule to add a lock and key.

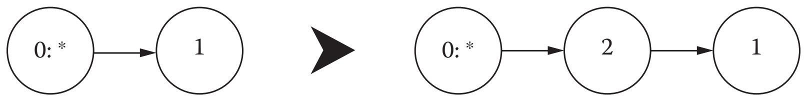


FIGURE 9.6 Graph rule to expand a cycle.

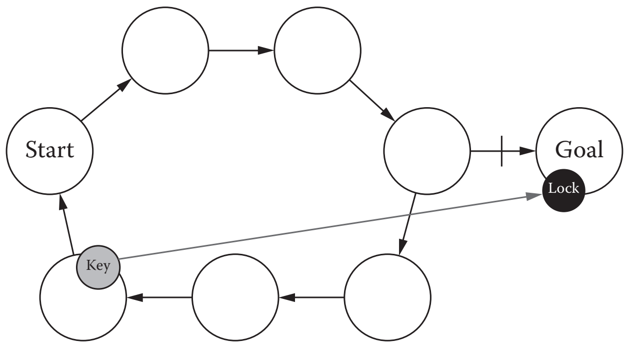


FIGURE 9.7 Sample of what can be generated from these rules.

in this case, the shortcut might be rewarding for players that take the time to explore thoroughly, but it can also bypass a large portion of an otherwise interesting level. However, the important point is that through transformational graph rules, the level generator is able to manipulate these patterns directly: it can make deliberate decisions about their use in order to maximize their positive effects and minimize their potential negative effects.

For example, most of the time, *Unexplored* starts a level by adding a starting point, a goal, and two paths. The first path, called *a*, initially leads from the start to the goal, while the second path, *b*, leads from the goal back to the start. Based on the relative and absolute lengths of *a* and *b*, a number of cycle patterns can be applied. For example, when *a* is short and *b* is long, it can place a locked door at the goal, and place the key at the end of *b* just before a spot where the player can cut back to the main path. This creates a level where players first encounter the locked door, and when they do find the right key, they do not have to go far to reach the door it unlocks. Obviously, this requires that the player cannot reach *b* from the cycle's entrance straightaway, although it might be interesting if they can already see *b* and the key. Figure 9.8 lists a couple more of these patterns.

IMPLEMENTATION

Of course, the devil is in details or, in this case, the implementation. Graph transformation and model-driven engineering are powerful methods, but to make them work in the way we wanted, we still had to add a couple of things. The most important addition to the graph transformation rules are attributes attached to nodes and edges in the graph. For example, a room node can have an attribute that specifies the room type or theme. The graph transformation rules in *Unexplored* can read and write to those attributes. In this way, the generator can activate a different transformation rule for rooms that are marked as a "shrine," "armory," "workshop," and so on.

Attributes turned out to be really important in generating lock-and-key mechanisms. For *Unexplored*, simply adding keys and locked doors was not going to work. There are several game elements that allow players to push ahead early and often involuntarily: they can jump in chasms to the level below, use a scroll of descend to similar effects, or use a scroll of teleportation to teleport to a random location in the level. As the objective

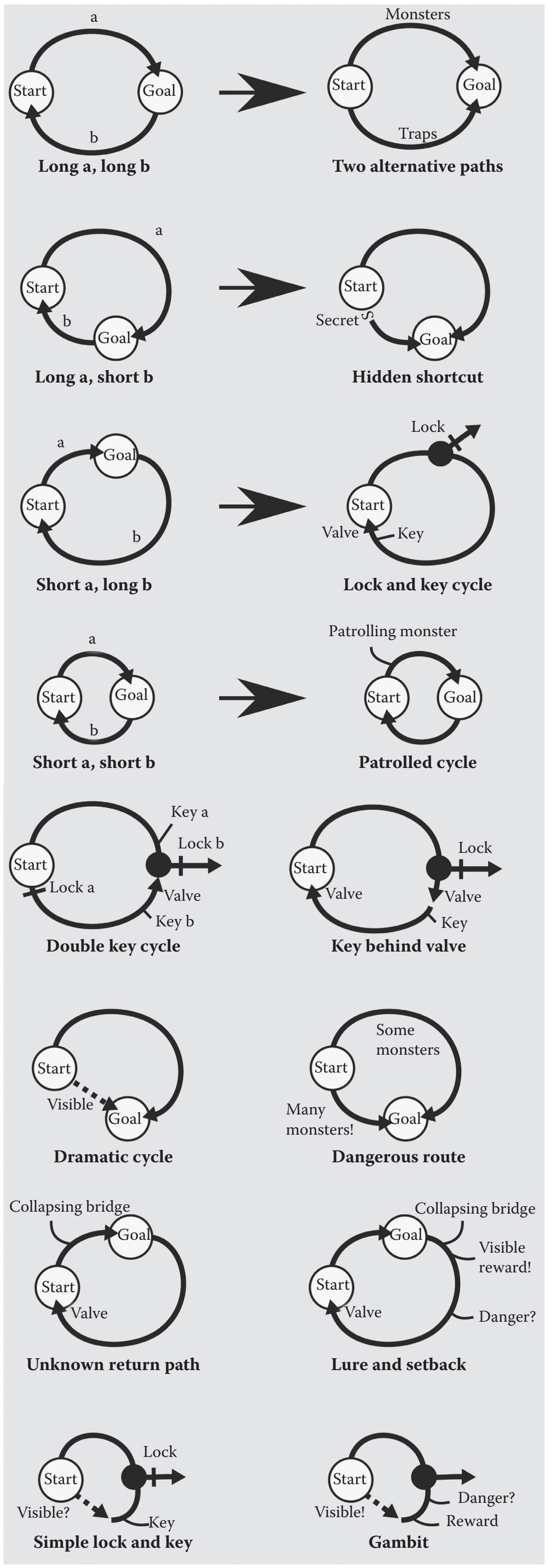


FIGURE 9.8 Level design patterns.

of the game is to get to level 20, retrieve the Amulet of Yendor, and make your way back up, this means that many of the lock-and-key puzzles need to be solvable in two directions.

A simple solution would have been to make all locked doors operated by levers and simply make sure there is a lever on both sides of the door. However, lock-and-key mechanisms can get boring very quickly, especially when there is little variation to their design. One design objective of *Unexplored* is to use as many game mechanisms as possible as lock-and-key mechanisms. For example, a potion of resist fire can serve as a key to a barrier of lava the player needs to cross. Unfortunately, the potion has a temporary effect, and you are never quite sure when the player will drink it, and it definitely will no longer work when the player is returning to the surface.

Using attributes to mark the structural function of different locks and keys proved to be a very useful way to avoid deadlocks and create variety in lock and keys at the same time. For example, locks can be marked as "conditional," indicating a lock for which the player must have a key to pass. The most typical conditional lock is an indestructible door that can be opened only with a particular key. An example of a nonconditional lock would be a room with turrets controlled by a lever. If the player has not found the lever, he or she can still risk running through the room. Nonconditional locks have the advantage of being traversable in two directions automatically, and it is often easier to involve different game mechanics in their design.

ASIDE: LOCK-AND-KEY ATTRIBUTES

Locks Might Be Conditional, Dangerous, or Uncertain

Typically, we think about locks as binary barriers: if you have the key, you can cross the barrier; if you don't have the key, you simply cannot. Locked doors, in whatever way they are unlocked or opened, are the typical example. Locks as binary barriers are conditional locks. However, locks do not need to function in that way. Some locks are barriers that might be navigated without a key, but this crossing the barrier might be uncertain or impose a certain risk. Keys for this type of lock make the crossing less dangerous, or more certain. A simple example of an uncertain lock would be a secret door, which is opened if one is able to find it (a certain matter of uncertainty). A dangerous lock could be a lake of lava that might be crossed at the expense of hit points (a certain risk). A scroll of magic mapping that reveals secret doors or a potion that protects you against fire damage can act as keys for those doors, making it easier to cross the lock in both cases.

Locks Are Permanent, Reversible, Temporary, or Collapsing

When you unlock a door, that door might remain unlocked forever (permanent), for a short period of time (temporary), or until it is relocked (reversible). Sometimes, a lock collapses after use, allowing the player only to pass once. Permanent locks are the safest to use, as once they are opened, nothing can go wrong. Temporary doors typically produce more gameplay at a certain risk; they can turn into valves (see the next section). Reversible locks can create problems, depending on the type of key you are using for them (see the following sections).

Locks Might Be Valves or Asymmetrical

Certain locks allow you to cross only in one direction (valves), while others can only be opened from one direction but traversed in two directions after they are opened (asymmetrical). Valves and asymmetrical locks tend to create more interesting conditional locks. They are frequently used in the patterns above to make sure the player cannot enter a part of the cycle that is intended as a route back. Valves do not always require a key. For example, jumping down a high ledge can be considered to be a valve, as it allows the player only to travel in one direction.

Locks and Keys Can Be Safe or Unsafe

A safe lock is guaranteed to have a solution, while an unsafe lock is not. To a certain extent, a dangerous or an uncertain lock is always safe. In general, you want to make sure all conditional locks on the main path through the game are safe. It is fine to use unsafe, conditional locks on optional paths, although if that lock is also temporary, collapsing, or a valve, you must make sure that the path back is safely locked (if it is locked at all).

Keys Can Be Single Purpose or Multipurpose

Single-purpose keys can only be used to open a lock, and for nothing else, while multipurpose keys can also be used in different ways. Even if a key can open multiple locks, it can be considered to be multipurpose. In general, it is better to make use of multipurpose keys, as single-purpose keys are more boring and frequently end up as dead weight in your inventory. The way *Zelda* dungeons frequently use the bow as a key by hitting switches from a distance is a good example of a multipurpose key. In fact, many of the best keys in *Zelda* also double as a weapons.

Keys Are Particular or Nonparticular

Particular keys are the only thing that unlocks a particular lock, whereas several nonparticular keys might unlock a single lock. Just as conditional locks tend to be more boring than dangerous or uncertain locks, particular keys tend to be more boring than nonparticular keys, often because nonparticular keys tend to be multipurpose as well (although they do not need to be). Particular keys function as do real-life keys. The small keys of *Zelda* are a good example of single-purpose, nonparticular keys. One effect of using nonparticular keys is that the players, for whatever reason, might have one when they encounter the lock. When you have not placed a nonparticular key for each lock you want it to open, this can make conditional locks feel less rigid.

Keys Might Be Consumed or Persistent

Keys that are destroyed somehow in the process of unlocking a door are consumable, while keys that are not are persistent. Keys that are consumed tend to be less safe than persistent keys, especially if those keys are also multipurpose and can be consumed to achieve other goals. Small keys in *Zelda* are consumed, as is a potion of resist fire that is placed as a key to pass a certain fire barrier. Like nonparticular keys, consumable keys might make a conditional lock less rigid, but also a lot less safe.

Keys Might Be Fixed in Place

Levers and switches are the best example of keys that are fixed in place (and typically single purpose and particular as well). One problem of fixed-in-place keys is that if there is only one key on one side of the lock, it makes the lock asymmetrical. Often, it is best to make the locks triggered by keys that are fixed in place open permanently, especially when the player can reach the key only once, which effectively would make the key consumable and probably unsafe as well. When using keys that are fixed in place, it is often important to give some sort of feedback on the status of the door, and to make sure the lock is permanent and not reversible (as often is the case with levers).

TILEMAPS

Another problem we ran into during the implementation of these techniques in *Unexplored*, but also in other games using similar ideas, is how to transform graphs into tilemaps. The problem is a generic one: during the initial stages of the generation process, graphs are used to represent the abstract structure of the level. Graphs are good for this, as they can include edges to represent the structural relationships between nodes. These relationships include adjacencies, but also which key unlocks what lock, what enemy patrols which rooms, and so on. However, at one point the generator needs to build a topological representation detailing the exact locations of wall, doors, keys, and enemies. In *Unexplored*, levels are represented as tilemaps.

Unfortunately, there is no standard solution to this problem. For *Unexplored*, we solved the issue by first creating a graph that is laid out as a grid and stored the topological information in the nodes' attributes (Figure 9.9). All graph transformations are then performed on this graph. This way, when the generator needs to make the transition from graph to tilemap, there are no issues in mapping the graph onto a two-dimensional tilemap. Initially, individual tiles correspond with nodes and edges directly, but the resolution of the tilemap is then increased to allow for

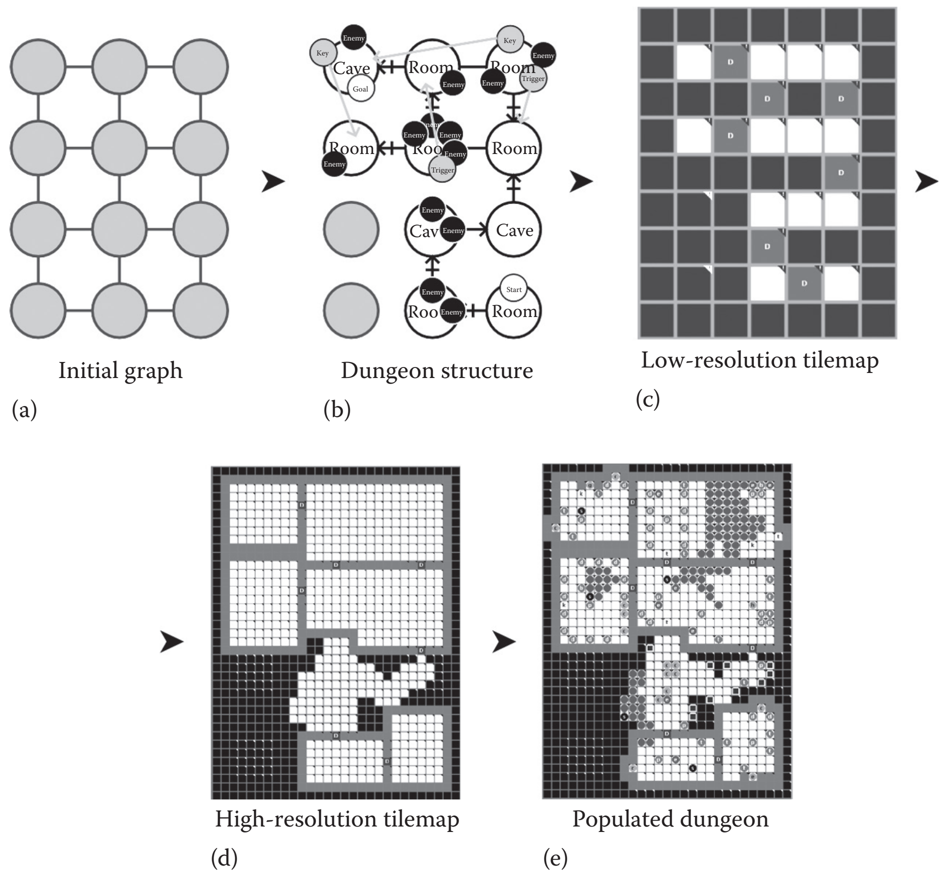


FIGURE 9.9 Graph grid layout that any dungeon level starts from.

more details. During subsequent steps in the generation process, transformation rules, much like the graph transformation rules discussed in an earlier section, are used to expand rooms, add features as dictated by the graph, and decorate dungeons.

DISCUSSION

This chapter does not include every detail of implementation. For example, before even one dungeon level is generated, a plan for the dungeon is generated that specifies what themes, monsters, locks, and keys to use on each level. However, the general idea behind cyclic dungeon generation should be clear. The implementation in *Unexplored* relies heavily on the use of transformational grammars operating on graphs and tilemaps, but it is not hard to imagine that other generation techniques could also be used to generate dungeons based on cycles instead of branching paths.

For *Unexplored*, the advantages of cyclic dungeon generation are many. We have found that levels based on a couple of cycles are the most effective. Two to five cycles give each level a distinct and recognizable shape and character. This presents the player with a nice, but manageable navigational challenge. With just a handful of cyclic patterns, the number of possible combinations is still huge. Adding more cycles just seems to clutter the level.

The reason cyclic dungeon generation works well is that as a structure, cycles are able to express so many more interesting patterns than paths do. There is no reason to assume that this advantage is unique to cycles alone. Structures like T-joins or parallel, symmetrical branches might be able to achieve similar results in addition to, or independent of, cycles, but so far cycles are the most powerful and expressive structure we have come across.

In addition, cyclic generation is not restricted to the generation of sprawling dungeons alone. In fact, the research workshop from which the ideas behind cyclic dungeon generation emerged tried to come up with a good way to generate parks or meditative gardens. In such environments, cycles are also very useful: most people prefer a walk or hike that does not involve any backtracking. Similar ideas can be applied to open game spaces and world maps. Cycles can be drawn around obstacles such as forests, lakes, swamps, and mountains, with dangerous shortcuts connecting two sides of the cycle and prominent sites on locations where two cycles intersect. There are many opportunities that cyclic map generation offers that remain unexplored—for now.