Content Generation in Dungeon Run

Dungeon Run 2: Dual Wield is an action-adventure roguelike. Basically, the game takes the common tropes of dungeon crawling in roguelikes and translates these into a real-time action game, not unlike a 2D *Zelda* game. The latter type of game is characterized by fairly tight level design with many lock and key puzzles. Although *Dungeon Run* is somewhat looser in design, and as a roguelike depends also on strategic choices and the ability of the player to improvise. Still, the type of gameplay demands high standards for the generated dungeons. This means that in designing the dungeon generator I had to go beyond the techniques commonly found in roguelikes, to create fun dungeons with challenging quests and puzzles that span multiple dungeon levels.

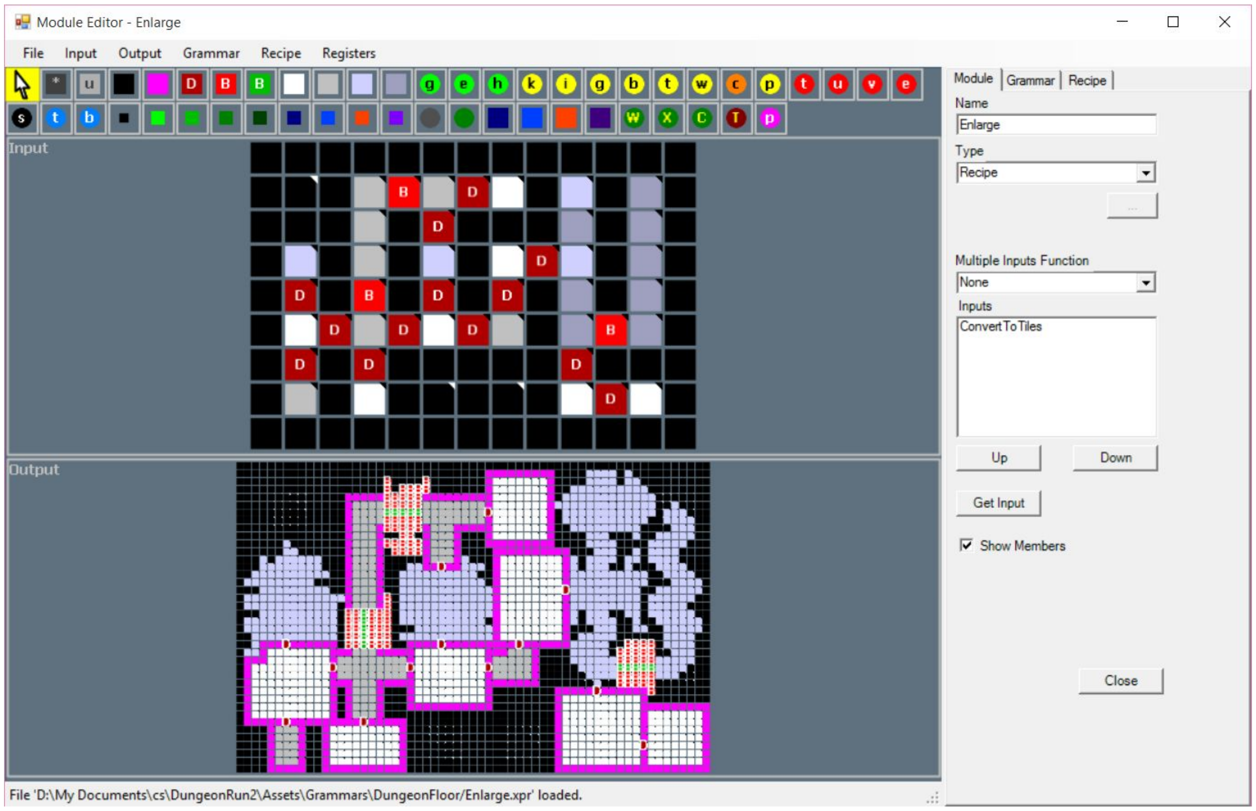
This article describes my approach to content generation for this game. It outlines the general approach (Model Driven Engineering) and talks about the importance of working with the right type of models. It goes into considerable detail where lock and key generation are concerned. I feel the lessons I learned in developing this game (and the techniques behind it) apply to many game beyond *Dungeon Run*.

Disclaimer: At the moment of writing at least 90% of the stuff I write about is actually implemented in the game. The remaining 10% are plans of which I pretty sure that will get implemented. That being said, the game remains in development, and I am sure I will add stuff to the game not covered in this article.

Approach: Model Driven Engineering

To start with, the general approach I used comes from Model Driven Engineering. I have been researching this technique on and off for several years. The toolset I have developed for this research (Ludoscope) is tailored towards Model Driven Engineering (see figure 1 and sidebar below). Needless to say, without this tool, *Dungeon Run* would not have been possible. The idea behind Model Driven Engineering is quite simple: do not try to generate something, in this case a dungeon, in one huge step, but rather break the process down into many individual steps that each produces a model, which it passes on to the next step in the process. Every model in the process is a more refined representation of the thing you want to generate, and every step and model must be meaningful in themselves.

This approach has a number of clear advantages. It is far easier to manage than one big black box type algorithm. Each step can be designed and debugged individually. But more importantly, using different models as the target output for each step allows you to use the right type of model to solve the right problems. For example, the logic of lock and key puzzles is far easier to represent with graphs, whereas the layout and geographic structure of a dungeon is better captures using tilemaps or 2D shapes.



Name: Enlarge

Type: Recipe

Multiple Inputs Function: None

Inputs: ConvertToTiles

☑ Show Members

File 'D:\My Documents\cs\DungeonRun2\Assets\Grammars\DungeonFloor/Enlarge.xpr' loaded.

Figure 1: Ludoscope

What is Ludoscope?

Ludoscope is a general purpose tool to design content generation. The tool is designed around the use of transformational grammars to create content, and set up to follow the ideas of Model Driven Engineering. Ludoscope can handle different types of transformational grammars: string grammars, graph grammars, tile grammars and shape grammars, and includes a number of operations that manipulate and translate between different types of models.

Ludoscope allows you to set up any number of modules, each which executes a grammar, or follows a predefined script (mixing the application of transformation rules and other operations) and creates a new model based on its input. Ludoscope allows multiple inputs and outputs between modules and has some basic branching logics that allows you to create loops and logical gates.

In addition, Ludoscope extends some basic notions of transformational grammars. Most importantly, it allows symbols to have arbitrarily defined member values, and can manipulate and filter on those members when applying transformation rules. For example a string grammar rule in Ludoscope might read:

```
A(n>0) => A(n--) B
```

Applying this rule to the expression A(n=6) will always result in:

```
A(n=0) B B B B B B
```

I have been using Ludoscope and its associated content generation library in several games.

Abstract and Geographic Models

I like to break down the types of models I use into two main categories: abstract and geographic. Abstract models are often graphs or strings representing the logic embedded into a dungeon, whereas the geographic models I used are mostly tile maps that describe how the dungeon is actually constructed. For example, the graph figure 2 represent the room structure of a level in *Dungeon Run*, including the placement of locks and keys, triggers that operate traps and turrets, which enemies patrol or guard which areas, and so on. While the tile map in figure 3 is the finalized model for the same level.

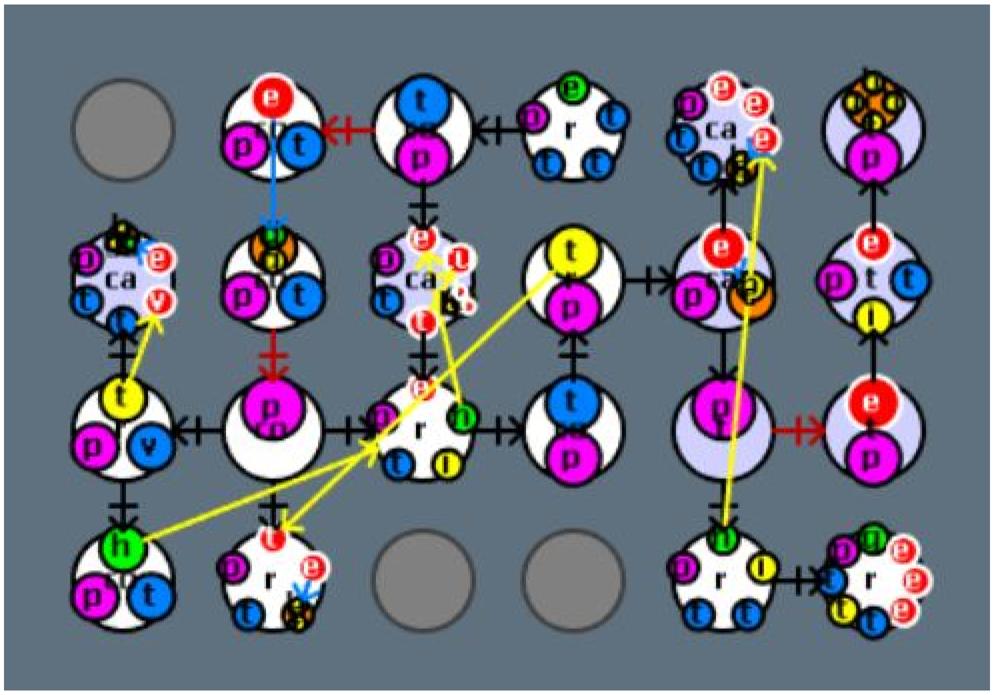


Figure 2: space structure of a dungeon, nodes represents rooms that contain enemies and hazords (red), decoration (blue), hints (green), chests (orange) and items (yellow). The green 'e' and 'g' and are the entrance and exit respectively.

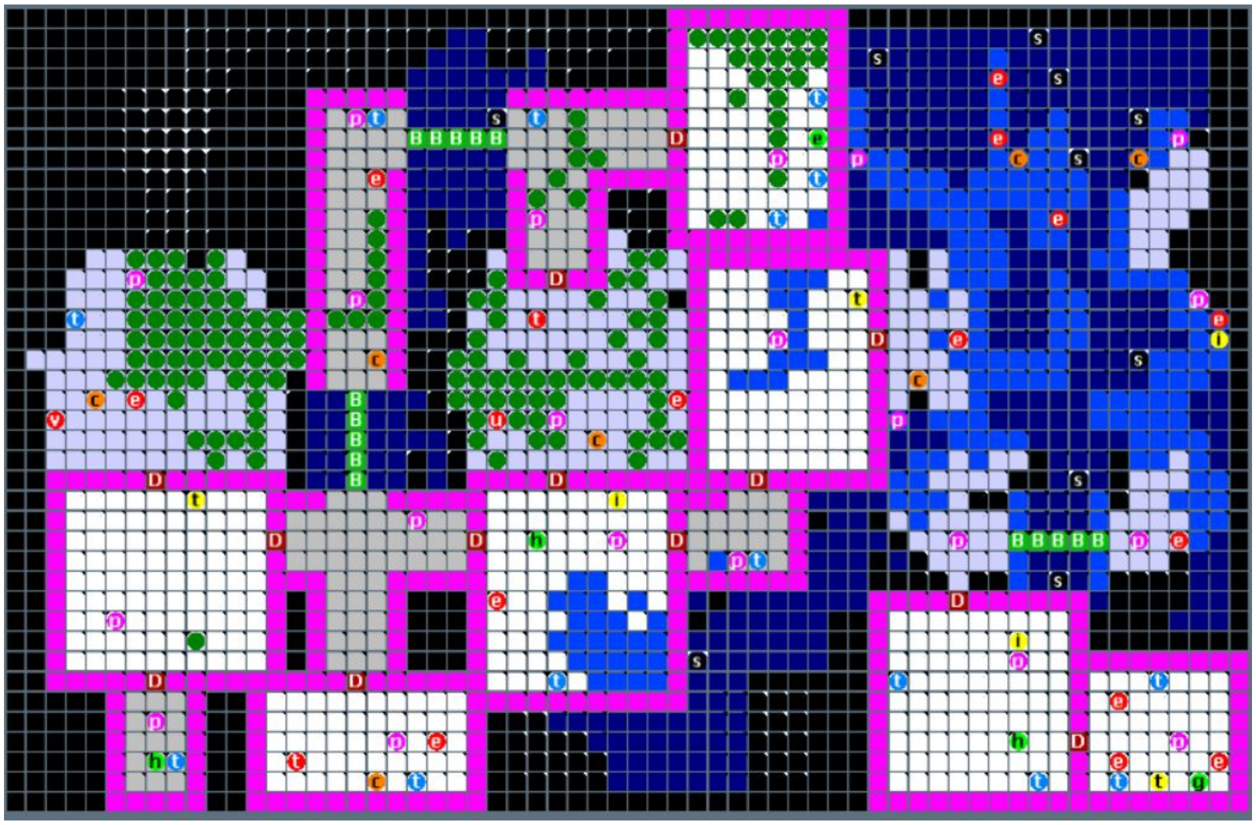


Figure 3: tile map of the same dungeon, showing rooms, deep and shallow water, a couple of bridges and many objects scattered around.

These two categories correspond closely to the elements of level design which I elsewhere call mission and space. Now one of the harder things to tackle with this approach is how to translate abstract mission structures into concrete spatial layouts. In the past I have tried several techniques: including automatic layout of graphs and folding graphs over voronoi diagrams. For this case (and a couple of others) I have settled on using models that are graphs, but graphs that already have some existing geographic structure. In this case, I started of from a graph that represents a grid of rooms (see figure 4).

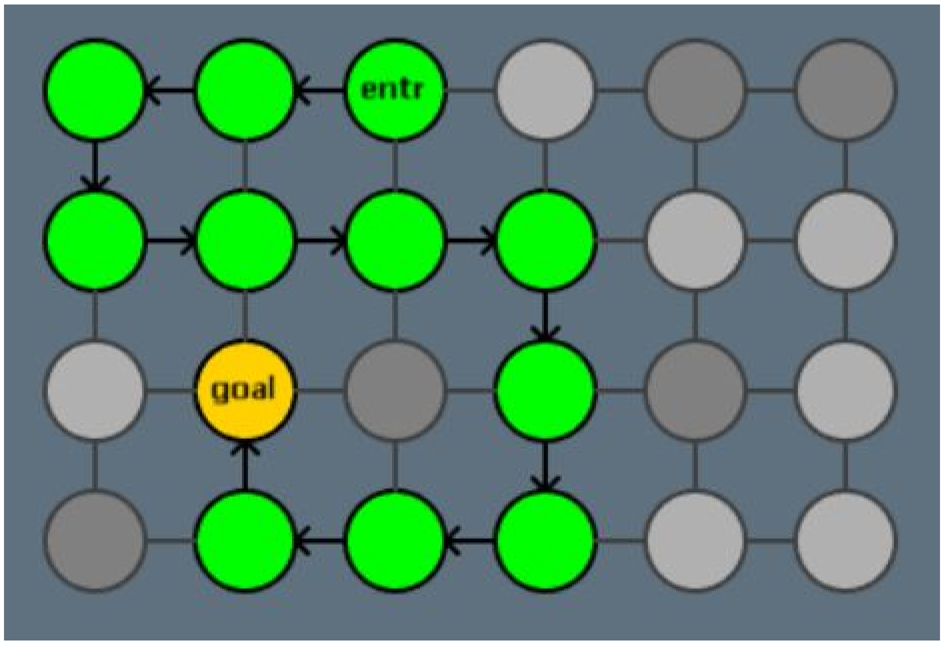
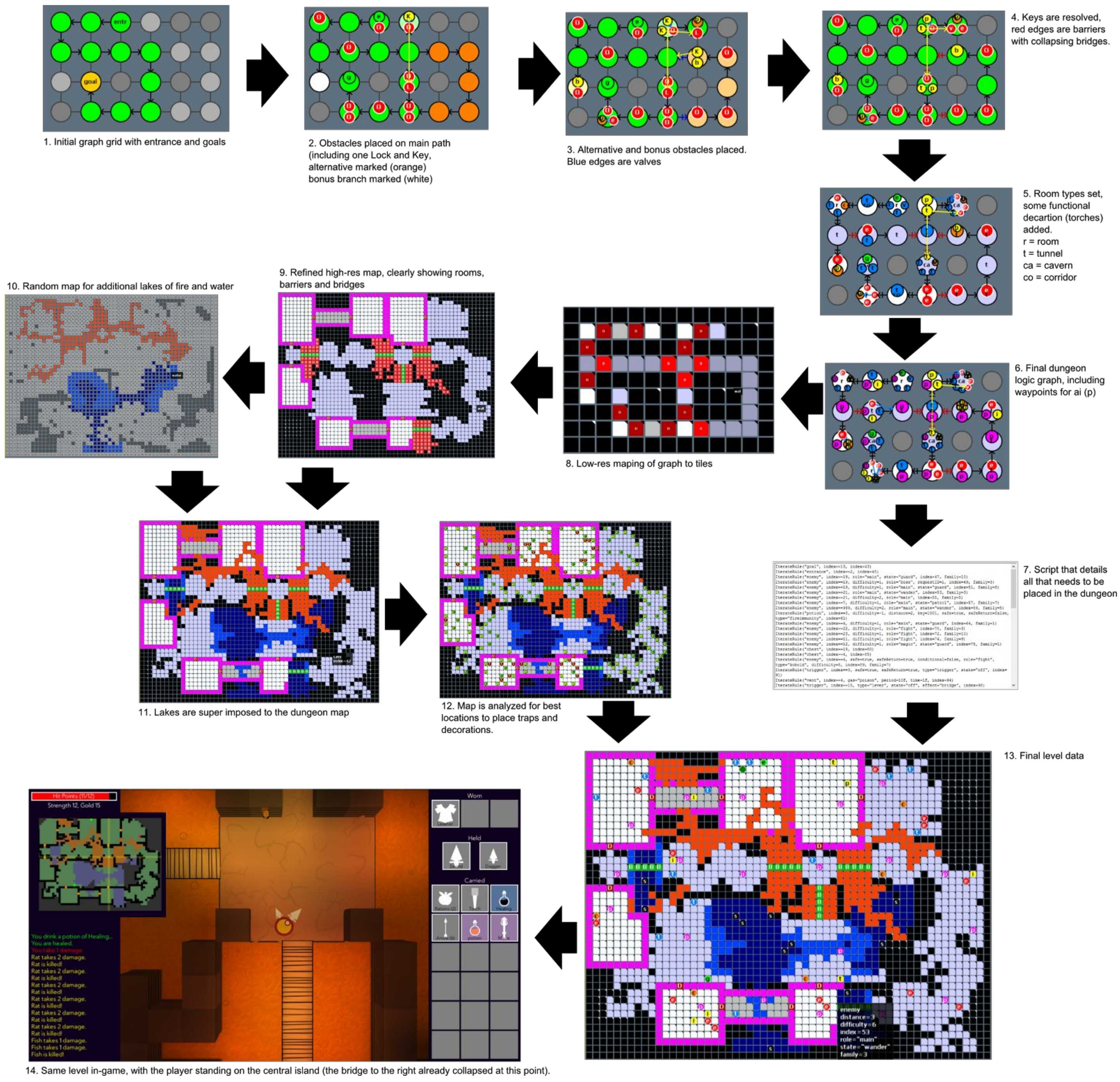


Figure 4 - A graph grid representing the start of new a level

The process of generating a level in Dungeon Run roughly follows the following steps (which are also depicted in figure 5):

- A graph grid is created.
- A random node in the grid is appointed as the entrance. From the entrance a random path is created. At the end of the path the exit is located.
- Possible alternative paths and extra branches are identified.
- On the path a number of obstacles are placed, including a couple possible lock and key mechanisms, where the key is placed at the end of a branching path. A progressive difficulty is added to each room on the main path. Basically the difficulty assigned to each room is 10 minus the number of obstacles you have passed to get there.
- Locks and keys might be placed on the main path.
- Alternative paths are assigned a number of obstacles based on the difficulty difference at the start and the end of the alternative path. The obstacles on each individual path can be themed: some might involve magic, lots of fighting, or traps. Locks and keys might control the two entrances of the alternative path.
- Extra branches get get bonus rewards and more obstacles assigned to them, that might also be themed in a similar way. A lock and key might be added to the start of the branch.
- Lock and key mechanics are determined. What game elements are used as lock and key depends on what the dungeon level is allowed to do (can it use lava for example), but also on structural characteristics required by the lock and key mechanisms themselves (explained in detail below).
- The remaining tasks are specified: obstacles become enemies, traps, and so on.

- Rooms are decorated and assigned a type (room, corridor, tunnel, or cavern) based on what is in them. Extra loot is scattered in the dungeon.
- The whole structure gets translated into a tile map.
- Several tilemap operations create the correct outline for rooms, corridors, caverns, and so on. Each room, and corridor maintains a reference to the original room node in the graph structure.
- In a separate step, lakes of water, pools of lava and chasm are generated.
- The terrain is superimposed on the dungeon and resolved to make sure the dungeon structure is more or less maintained.
- Objects, enemies, traps, and decorations are placed in the dungeon.
- Now a lot of 'magic' happens in each individual step. There is far too much detail to discuss everything in detail. In the rest of this article I will cherry-pick a couple of the more interesting examples.


1. Initial graph grid with entrance and goals
entr
goal
2. Obstacles placed on main path (including one Lock and Key, alternative marked (orange) bonus branch marked (white)
3. Alternative and bonus obstacles placed. Blue edges are valves
4. Keys are resolved, red edges are barriers with collapsing bridges.
5. Room types set, some functional decartion (torches) added.
r = room
t = tunnel
ca = cavern
co = corridor
6. Final dungeon logic graph, including waypoints for ai (p)
7. Script that details all that needs to be placed in the dungeon
8. Low-res maping of graph to tiles
9. Refined high-res map, clearly showing rooms, barriers and bridges
10. Random map for additional lakes of fire and water
11. Lakes are super imposed to the dungeon map
12. Map is analyzed for best locations to place traps and decorations.
13. Final level data
enemy
distance=3
difficulty=6
index=53
role="main"
state="wander"
family=3
Hit Points (11/12)
Strength 12, Gold 15
Worn
Held
Carried
14. Same level in-game, with the player standing on the central island (the bridge to the right already collapsed at this point).

```
IterateRule("goal", index==13, index=43)
IterateRule("entrance", index==2, index=45)
IterateRule("enemy", index==19, role="main", state="guard", index=47, family=10)
IterateRule("enemy", index==19, difficulty=1, role="boss", requestID=1, index=49, family=3)
IterateRule("enemy", index==19, difficulty=1, role="main", state="guard", index=51, family=8)
IterateRule("enemy", index==21, role="main", state="wander", index=53, family=3)
IterateRule("enemy", index==21, difficulty=2, role="main", index=55, family=5)
IterateRule("enemy", index==7, difficulty=1, role="main", state="patrol", index=57, family=7)
IterateRule("enemy", index==999, difficulty=2, role="main", state="wander", index=59, family=5)
IterateRule("potion", index==3, difficulty=-1, distance=2, key=1001, safe=true, safeReturn=false,
type="fireimmunity", index=61)
IterateRule("enemy", index==4, difficulty=1, role="main", state="guard", index=64, family=1)
IterateRule("enemy", index==22, difficulty=1, role="fight", index=70, family=3)
IterateRule("enemy", index==23, difficulty=1, role="fight", index=72, family=10)
IterateRule("enemy", index==11, difficulty=1, role="fight", index=74, family=9)
IterateRule("enemy", index==12, difficulty=1, role="magic", state="guard", index=78, family=1)
IterateRule("chest", index==19, index=80)
IterateRule("chest", index==4, index=85)
IterateRule("enemy", index==4, safe=true, safeReturn=true, conditional=false, role="fight",
type="kobold", difficulty=5, index=89, family=7)
IterateRule("trigger", index==3, safe=true, safeReturn=true, type="trigger", state="off", index=
91)
IterateRule("vent", index==4, gas="poison", period=10f, time=1f, index=94)
IterateRule("trigger", index==15, type="lever", state="off", effect="bridge", index=98)
```

```
You drink a potion of Healing..
You are healed.
You take 1 damage.
Rat takes 2 damage.
Rat is killed!
Rat takes 2 damage.
Rat is killed!
Rat takes 2 damage.
Rat is killed!
Rat takes 2 damage.
Rat is killed!
Rat takes 2 damage.
Rat is killed!
Rat takes 2 damage.
Rat is killed!
Fish takes 1 damage.
Fish takes 1 damage.
Fish is killed!
```

Figure 5 - The entire process

Generate structure first, specify later

In my experience, any content generation procedure should start with creating abstract shape and structure and refining these into specific level designs. An important reason for this is that when you starts with generating specific constructions, you often end up trying to control the output too much. You might have been better off hand designing levels. As an example of focussing on structure before focussing on details: in Dungeon Run the entire dungeon consists

of rooms and obstacles. Their exact nature is not defined at this point. Later on, an obstacle might turn into an enemy, a trap, a locked door, and so on. And rooms with many stuff end up being actual rooms, while rooms that have little in them might be turned into corridors. The advantage of leaving the nature of the obstacle undetermined initially, is that I can assign structural roles to obstacles first, and later design what specific type of obstacle best matches the role assigned to it. For example, an obstacle might be assigned the role that it can do damage, or that it tests the player's dexterity, or both. Based on the placement of the obstacle (is it on the main path, and alternative route, or a branch leading to a reward) different attributes can be assigned to it. In general, I found it far more effective to let the structure of the dungeon dictate all these things before I try to determine what specific obstacle is best used.
Now you might think that this means that is generally better to start with abstract models such as graphs first, but this not necessarily true. As I have found out designing the right way to deal with the alternative paths in the dungeon, sometimes it is better to let the spatial affordances of the dungeon dictate the design.

This is a lesson I had to learn the hard way: when creating alternative paths I first was tempted to try to generate more or less balanced paths from the start. It simply seemed the natural way to go. However, that proved to be very tricky. Easy enough within the logic of a graph, but very hard to map to space correctly. At one point I switch back to using a graph that was locked to a grid to generate paths in a more geographic fashion. This creates to problem that it was very difficult to control the length of alternative paths. An alternative might cut the main path short be several nodes, making it the far better route. However, at one point I realized that this is not necessarily a bad thing, in fact, it proved to be an opportunity. It simply means that more and more difficult obstacles need to be placed within that branch. In my experience it turns out that letting arbitrary and random spatial constraints dictate design solutions to create an interesting a balanced game is a far easier way to generate variety than trying to control the design space from the start. It is somewhat akin to drawing a dungeon by hand on an empty piece of paper, or one randomly folded and cut. The latter technique inspires much more creativity from most designers.

The parts of the generation process that are responsible to create an interesting experience are basically solving the problems the earlier and more random parts of the generator is generating: The alternative route is cutting a long main path short, how to balance that? This branch goes nowhere, what can I do to make it interesting or rewarding for the player, nonetheless? What are the choke points in this level and how can I best used them?

Game Design: Working with the right components
Now I would be lying if I would not admit that procedural content generation is about 50% clever algorithms and 50% happy accidents. Sometimes Dungeon Run produces very interesting curve balls, but frequently this is the result of a lucky combination of game features. There is probably no way around it, and we have to accept that when generating levels for a game like DUngeon Run, nothing can beat a well constructed level designed by an experienced human designer. However, what we can do is to design the game's component in such way that the chance for

happy accidents is maximized. As a result the feel of the dungeon might approach the apparent cleverness of hand-designed levels.

For Dungeon Run, and many other roguelikes, the way happy accidents are encouraged it to make each individual game element have many possible interactions with other elements in the game. For example, a simple element like the torch in Dungeon Run has many possible interactions: it provides light, which allows you to detect traps more easily, but also makes using stealth impossible. Rats fear torches, but swimming douses torches, making sure that you don't want to go for a swim with many rats nearby (and as it happens the chances for water to appear increases when rats are in a level). Gasses might explode or burn away when ignited by a burning torch, creating potential deathtraps, as well as viable strategies to overcome obstacles. The more interactions you can design into your game elements the better. Especially if those reactions are both good and bad. That way you avoid emerging dominant strategies (at least to a certain point). The torch is a good example of good and bad effects, but so are many potions. Levitation for example, allows you to cross hazardous terrain safely and avoid traps. But it also prevents you from picking up submerged items, and makes your character more difficult to control. In addition you have to be very careful when a potion is nearly finished as you might fall into a gap or even be killed by landing on lava. All possible combinations in Dungeon Run are too many to catalogue here, but in general all main features of the game: weapons, enemies, traps, terrain, gasses, light, darkness, and magic effects, are setup to maximize interactions between them.

Meta Design: Planning the Dungeon
So far, I have been going on about the generation of individual levels of the dungeon. However, that is not where the actual generation process in *Dungeon Run* starts. It starts with a general plan for the entire dungeon. At its simplest that plan is a succession of dungeon levels with the Amulet of Yendor placed at a specific level. But the dungeon plan dictates much more: it creates pacing, by setting up bosses and special rewards every so often, it creates puzzles that span multiple dungeon levels, and it allows the foreshadowing of some of these events.

The placement of rewards plays an important role in this design. A reward is found on about 1 in 3 levels. Rewards might include vaults of treasure, special magical weapons, or simply a special item required to progress through the dungeon. All rewards are guarded by bosses, although some reward keys that give access to the reward instead of the rewards themselves. Bosses play an important role in how levels are themed. Crucially, each boss casts a shadow to levels above it. For example, one of the first bosses you might encounter in the dungeon is the giant rat. If it is located at level 2, you are sure you can find many rats at that level, too. And on the level 1 as well. Foreshadowing is created by using specific minions for a boss you'll likely encounter soon, but also by providing particular sort of equipment to deal with the boss or its minions. For example, there is a specific hint that is often generated when players find themselves on a level with many rats for the first time telling that rats fear fire, at the same time, more torches are placed on that level to find. When having to deal with many skeletons, that game is likely to generate a stash of Scrolls of Last Rites, which prevents skeletons from rising again after you defeated them.

Figure 6 gives a simplified example of a dungeon plan. It simply determines what sort of enemies, terrain, bosses, special rewards should be encountered on a particular level. Some of these items are translated directly into generation instructions for the level generator: this level must include a boss, two bonus items, and a hint, for example, or this level likely contains lakes of lava, but no rooms and corridors. In addition, the level generator doesn't really specify what specific creatures are placed in the dungeon, rather it specifies their role: it is a heavy hitting enemy that can patrol, or a magic creature that is guarding this chest. It is only when the level is actually instantiated the game determines what specific creatures allowed by the dungeon plan best fulfill these particular roles.

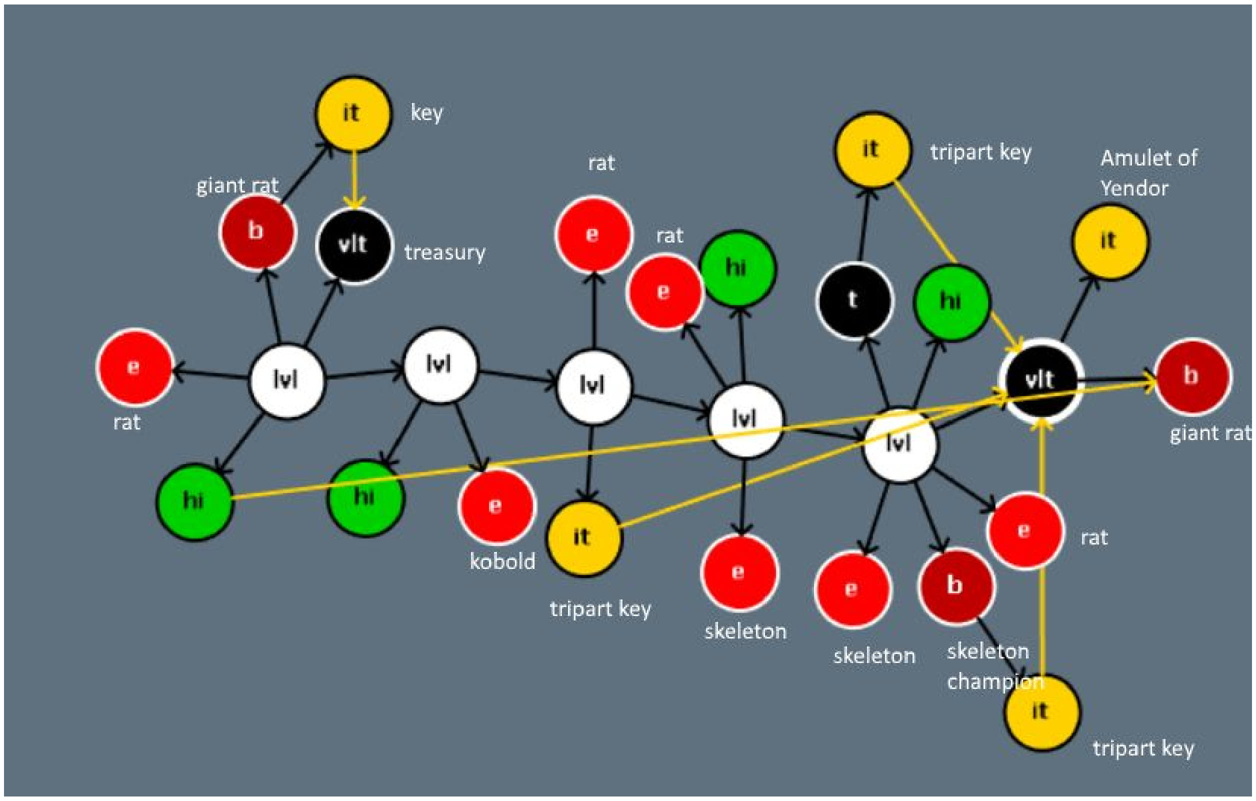


Figure 6: A limited sample of a dungeon generated with five floors and two vaults. Legend: e=enemy, b=boss, lvl=level, vlt=vault, hi=hint, it=item, t=test. Yellow connections indicate links of association.

This way coherence can be created in the dungeon, up to the point that quests and storylines give the dungeon flavor and purpose. Right now the generator is capable of foreshadowing bosses using enemy types and explicit hints. It can generate lock and keys spread over multiple floors, create alternative routes back up into vaults, and reward special quest items such as the hookshot or a magic bow that can exploited as key mechanisms after they have been found.

Beyond simple locks and keys
Locks and keys are the staple of dungeon design in *Zelda* games. As *Dungeon Run* borrows gameplay from that series, locks and keys are more prevalent *Dungeon Run* than in most roguelikes. In the Zelda games, the design of lock and key mechanisms is critical for the gameplay. Locks and keys are not just actually locked doors and associated keys, although they sometimes are just that. Rather, most locks and keys come in many different shapes and sizes, as too obvious locks and keys are very boring very quickly.

For Dungeon Run this means that the keys being generated should follow similar logic. They need to be quite frequent, but varied as much as possible, but preferably without creating deadlocks or other design flaws that might end up trapping the player in an unbeatable level. In order to make that happen the generation process needs to be aware of the different attributes locks and keys might have, and how these relate to the structure of the dungeon. Below is a discussion of some of these qualities, followed by a discussion how these qualities were leveraged to create more interesting quests in *Dungeon Run*.

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. I refer to this type of lock as a conditional locks. 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 which might be crossed at the expense of hit points (a certain risk). A magic 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. In general I find the latter two types of locks to be more interesting as they allow for more varied gameplay. Instead of simply hitting an arbitrary barrier, the player might try to cross, fail and retreat, come up with different strategies, and so on. Keys for dangerous or uncertain locks might be combined the reduce risk even further calling for careful risk assessments on the part of the player. However, conditional locks do have their uses, especially within protecting quest objectives and preventing the player from entering certain locations before they are ready. However, in my experience, it is best to use conditional locks sparingly as they can make a game feel more rigid and bore more easily than dangerous or uncertain locks do.
Note that a lock can be conditional and dangerous, or dangerous and uncertain. For example when crossing a dangerous area requires you to have a certain key item to even try to make it through. The lock is both conditional and dangerous.

Locks are Permanent, Reversible, or Temporary. 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). Permanent locks are the safest to use, as once they are opened nothing can go wrong. Temporary doors typically produces more gameplay at a certain risk, they can turn into valves (see below). Reversible locks can create problems depending on the type of key you are using for them (see below). In Dungeon Run I tried to create permanent locks for the main path down into the dungeon, but bonus objectives and alternative routes can make more frequent use of other types.

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 typically are conditional locks, but they tend to create more interesting conditional locks. Also they have different effects on the structure of a dungeon. For Dungeon Run 2 I used them to great effect to seal of one or two ends of an alternative route. For example, by using a valve as the access point to the start of an alternative route you force the player to commit to that route even though that route turns out to be more challenging. The use of valves and or asymmetrical locks also creates a possible solution for the problem of too short alternative routes (as discussed above). If that route can only be used to travel back, the problem is solved. In fact you see many of such asymmetrical locks creating permanent shortcuts to a central hub in Zelda levels. Asymmetry can also be a problem with locks on the main path in Dungeon Run as the player actually might get behind them accidently. So for example when a key need to be found to unlock a door on the main path, Dungeon Run generates a second key just behind the lock for the player to use should have found another way down.

> ***Special case: the unlocked valve.*** *I also think of valves as locks, although many valves don't need a key. For example the player jumping down a chasm in Dungeon is using a valve into the next level (and a risky one at that!). Other unlocked valves in the game include collapsing bridges over lakes of lava. Although they are not locked per se, they share many similar properties of locks actually operated by a key.*

Locks and Keys can be Safe or Unsafe. A safe lock is guaranteed to have a solution, while a unsafe lock is not. To a certain extend a dangerous or a uncertain lock is always safe. It is important for a game like Dungeon Run to have only safe, conditional locks on its main path. It is fine to use unsafe, conditional locks on optional paths, although if that lock is also temporary or a valve, you must make sure that the path back is safely locked (if locked at all). Whether a lock is safe or unsafe depends on the particulars of the lock and its key. Sometimes weird combinations of game elements might make a lock unsafe. For example in Dungeon Run is is possible for a key to be carried by an enemy. If that enemy is then killed by throwing it into a chasm, the key is lost, and the lock remains locked forever. So essentially that lock and key combination was unsafe. It can help to mark the key as 'must-be-safe' when using it on a mission critical path, so at least the game can respond when it is about to be lost.

Keys can be Single-Purpose or Multi-Purpose. Single-purpose keys can only be used to open a lock, and for nothing else, while multi-purpose keys can also be used in different ways. Even if a key can open multiple locks it can be considered to be multi-purpose. In general it is better to make use of multi-purpose 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 multi-purpose key. In fact, many of the best keys in Zelda also double as a weapon.

Keys are Particular or Non-Particular. Particular keys are the only thing that unlocks a particular lock, whereas several non-particular 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 non-particular keys. Often because non-particular keys tend to be multi-purpose as well (although they do not need to be). Particular keys function as do real-life keys, while the small keys of Zelda Dungeon are a good example of single-purpose, non-particular keys. One effect of using non-particular keys is that they player for whatever reason might have one when the encounter the lock, especially when you have not placed a non-particular key for each lock you want it to open, this can make conditional locks feel less rigid.

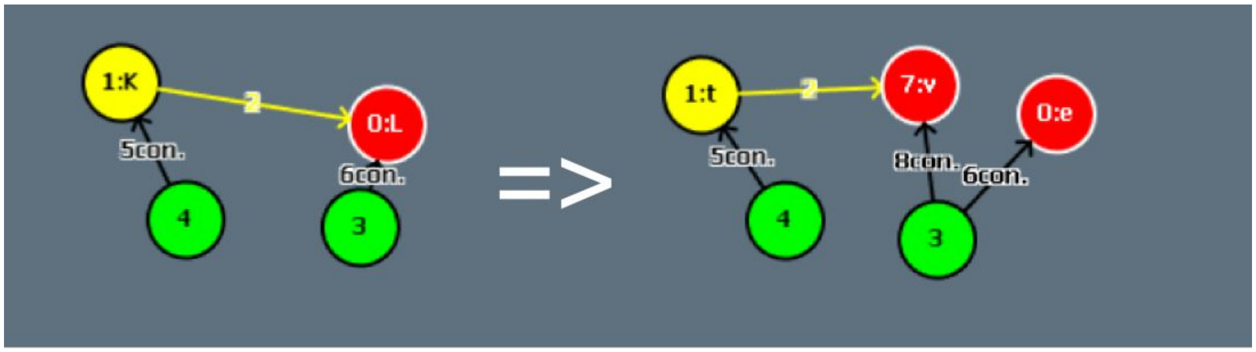
Keys might be Consumed or Persistent. Keys that are destroyed somehow in the process of unlocking a door are consumable, while keys that are not are persistent. Keys that are consumed tend to be less safe than persistent keys, especially if those keys are also multi-purpose and can be consumed to achieve other goals. Small keys in Zelda are consumed, as is a potion of fire-immunity that is placed as a key to pass a certain fire barrier. Like non-particular 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, are make sure the lock is permanent and not reversible (as often is the case with levers).

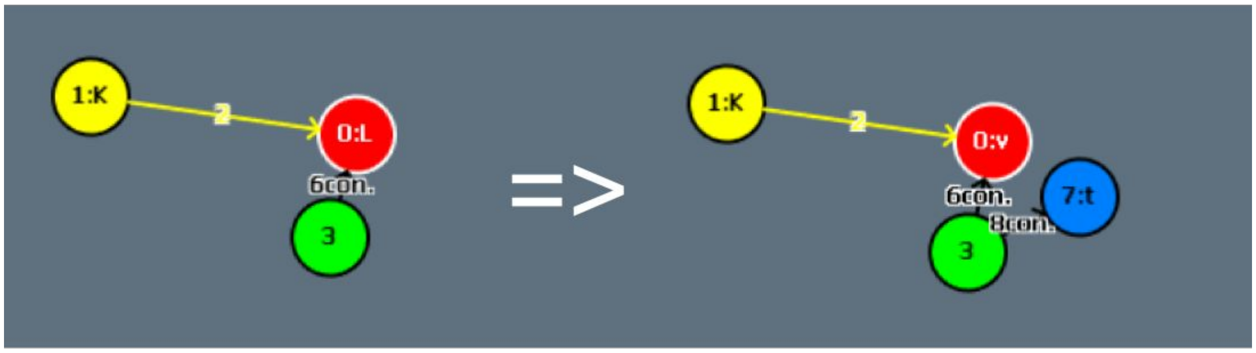
Discussion: Combining and Transforming Locks and Keys
Locks and keys tend to be strung together. For example, a door might be opened by a lever that is placed behind a field of fire, for which the player can find a potion of fire immunity elsewhere. When stringing together keys in this way you transfer characteristics from one lock and key mechanism to the next. In the case of the example above the door now is unsafe, because the player might use the potion for something else and never be able to reach the lever in the first place.

As should have become clear from the descriptions above, each type of lock and key has its function within the dungeon. Using a wide variety of types of locks and keys will add greater variety to your game. At the same time it is critical for the main route to use only safe, and symmetric keys, (and not too many conditional locks). In order to do so it is important to understand what combinations of characteristics make a lock and key combination unsafe, and what can be done to make sure the game stays fair. To get back to the example above, you might also want to make sure that the player can go back safely. The field of fire in this case is a temporary lock opened by a consumable key. While the player needs to cross it twice. The player might not get back in time. For this reason it is best to allow the player to return through in a valve or better still, an asymmetrical, permanent lock. That way the player can always return to the lever to make sure it is moved to the right position.

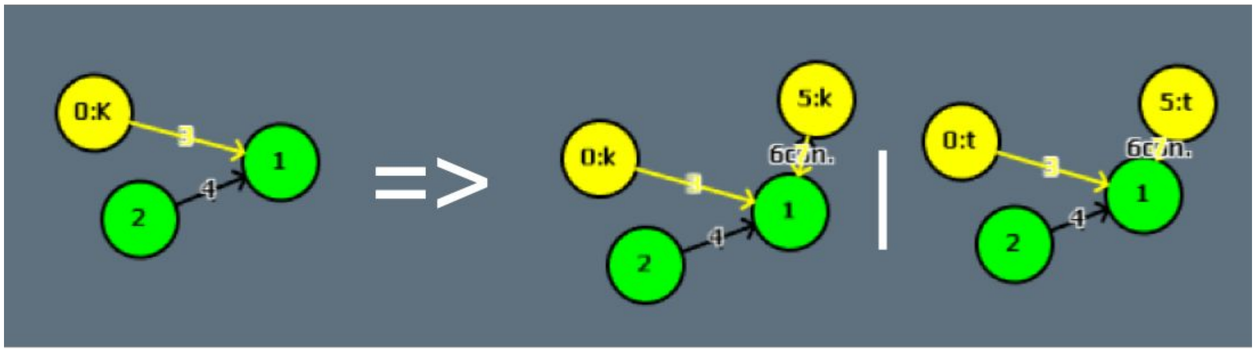
Dungeon Run makes heavy use of all these different characteristics of locks and keys during the dungeon generation. Before specifying what specific type of lock or key is used, the game generated the characteristics that are best suited to the circumstances. For example, they game uses valves and asymmetrical locks to one or two ends of an alternative route, while all locks and keys that are being generated on the main path must be safe and symmetrical.
The way the generator enforces this, is not by restricting the types of lock and keys usable on the main path, but by being aware of all these characteristics and fixing problems as they arise. For example, when a door unlocked by a remote trigger is used on the main path, the generate is aware of the fact it just used an asymmetrical lock where it should not have used one. As a result it will either make the lock symmetrical by adding an extra key on the other side of the door, or by creating alternative valve (or asymmetrical lock) to allow the player a safe return.
Figure 7 below shows a number a graph grammar rules used by the generator in order to create lock and keys. Note that in general these work in three steps. First the lock is generated, next a suitable key is added, and finally that key might be protected somehow (possibly by another lock and key mechanism).



A Key to a Lock can be a trigger that controls a poison vent near a big group of enemies



A Key to a Lock can be a Key that controls an explosive vent near a burning torch.
(In this case the key can be a controling lever or a potion of fire immunity)



A Key to a locked door that must be symmetrical can be a unique key for that door with an extra copy placed behind the door, or a two levers that operate the door.

Figure 7: some graph grammar rules to create locks and keys

Graphs can be hard to read. So here are some natural language translations of the rules that handle locks and keys:

- *A Lock can a flame venting turret999*
- *(When rats are viable enemies) A Lock can be a fight against many rats with its Key being a torch and a book explaining rats fear fire.*

- *(When rats are viable enemies) A Lock can be a fight against many poisonous rats with is Key being a potion of poison immunity. Optionally provide a hint to the nature of the potion.*
- *(When kobolds are viable enemies) A Lock can be a fight against many kobolds with is key being a lever that activates poison-vents nearby.*
- *A Lock can be a periodic poison vent.*
- *A Lock can be a periodic explosive gas vent near a burning torch.*
- *A Lock can be an hard to avoid trap.*
- *A Lock can be a secret door.*
- *A (conditional) Lock can be locked door.*
- *(When lava is allowed) A Lock can be a pool of lava.*
- *(When chasms are allowed) A Lock can be a chasm.*
- *(When water is allowed) A Valve can be a collapsing bridge across deep water (remove Key)*
- *A Key to a locked door (that can be asymmetrical) can be a unique key or a lever.*
- *A Key to a locked door (that cannot be asymmetrical) can be a unique key or a lever with a duplicate on the other hand.*
- *A Key to a secret door can be a scroll of magic mapping or a map of the dungeon floor.*
- *A Key to a secret door can be a unlit torch that must be lit to open the door. With a hint revealing the secret.*
- *A Key to a pool of lava can be a potion of fire immunity or a potion of levitation.*
- *If a potion is the key to a pool of lava (that must be symmetrical), add a lever to raise a bridge on the pool's far end.*
- *A lever can be protected by an enemy that guards that trigger.*
- *A potion that needs to be protected can be carried by an enemy or stowed in a trapped chest.*

Final Words

The devil is in the details, as they say, and although this article is already quite lengthy it can only scratch the surface of all the cleverness that I tried to pour into *Dungeon Run*. The generator is a huge and complex thing that I built over the past couple of weeks, but really was years in the making (see figure 8). Although I try to approach content generation in a structured manner, the collection of techniques I used is no silver bullet that is easily adapted to other projects. However, I do hope that the quality of the levels generated by the game do show the fruitfulness of the approach I took, and that this writing helps you understand the fundamental ideas behind its design.

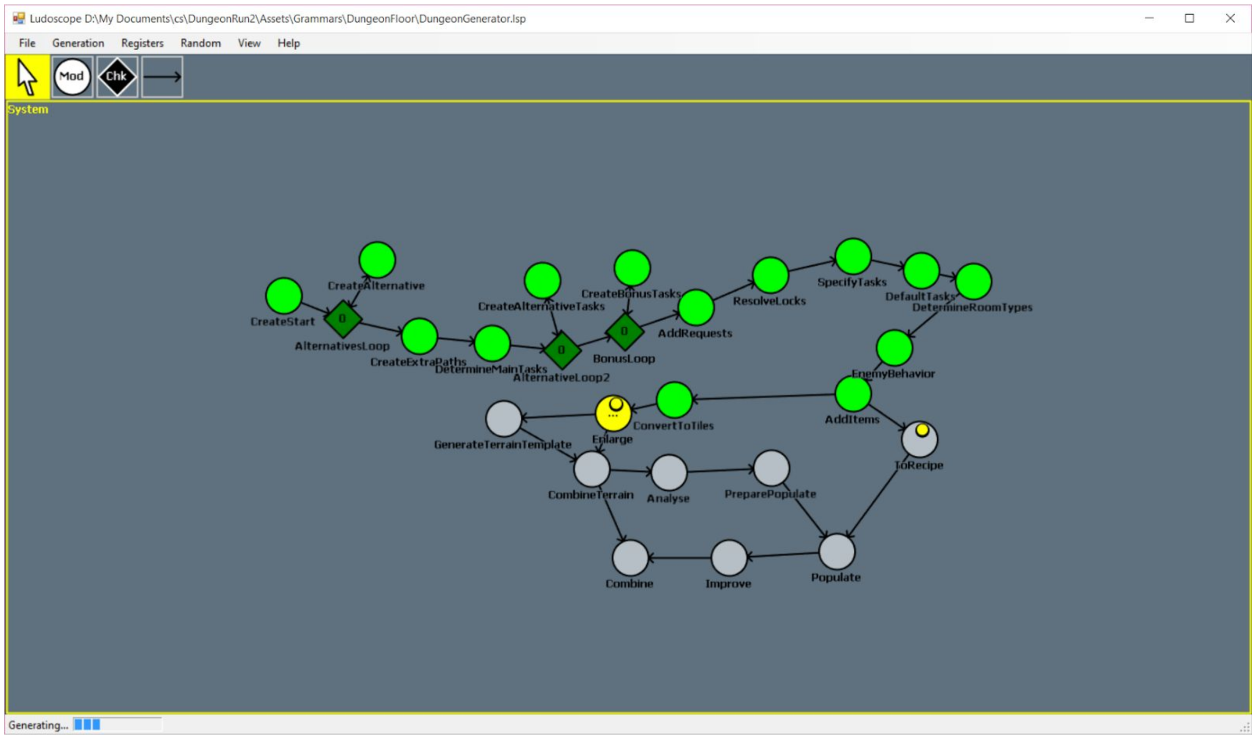


Figure 8: All the steps to generate a dungeon level as shown in Ludoscope while generating a new level.