Procedural Dungeon Generation Using Graph & Gameplay Grammars

2022-2023

Graduation Work

Sasha Vigneron

Kevin Hoefman

TABLE OF CONTENTS

ABSTRACT

Graph grammars can be used as a way to procedurally generate levels. This can provide a lot of control, whilst speeding up the development process by quite a lot. Next to graph grammars, there are also gameplay grammars, which makes it possible to generate the level based on desired gameplay elements. In this paper we will have a look at how we could combine the use of gameplay grammars and graph grammars to generate 3-dimensional levels. More specifically, we will use the gameplay grammars to generate the level-layout, whilst using the graph grammars to create the individual room-layouts.

INTRODUCTION

In order to be able to automatically generate levels, we first need to have a good understanding of how procedural generation works. Obviously, there are a lot of different ways one can handle proceduralism. We first have a look at what graph grammars really are, and how they can be used to generate levels. We then expand on this concept and have a look at how we could introduce these grammars into the game development sector.

We will be using a concept called gameplay grammars to generate our levels. Using gameplay grammars, instead of the gameplay being influenced by the rules that define the geometry and content of the level, it is the other way around. This makes it possible for the level designers to quickly adapt their graphs, creating entirely different levels, using the same gameplay mechanics. The first part of this case study will have a look at how we could develop a tool that can create, visualize and edit these graphs, that can be read during the generation process of our level. Once we have this system set up, something else we have to focus on is the visual aspect of the games. Besides gameplay, they also play a huge role in making an immersive and believable experience. To do this, we will use Unity's prefab system. During the generation of an individual room, we will procedurally place fitting prefabs inside of that room. This hugely simplifies the generation logic, since we can use 2-dimensional placement here, whilst keeping the third dimension inside of the prefab itself. Once all rooms have completed their generation process, the next step will be to link them to each other, creating a dungeon-like environment.

Once we have accomplished this, we will make it slightly more complicated, testing out a system where the generation of the level design and the visuals of the level are separated. Since the logic is mostly already there to handle the level design generation, we can focus on the visual aspect of the game here. The reason we're splitting this from the level design itself is to be able to give the artists a lot more freedom concerning the environment, in contrast to the use of prefabs. To procedurally generate these worlds, we will experiment to see if it is possible to use graph grammars, in combination with gameplay grammars, to try and achieve this.

On top of this, we will also have a look at whether or not we can add height alterations to give the environment a bit more 3-dimensional characteristics. We will try and add hallways to our rooms that can be used to achieve these height alterations. Whilst having a look at what levels of complexity this introduces when generating our levels. And how we could work around these possible issues.

RELATED WORK

Procedural generation in games has existed for quite some time now. During this time the generation process has improved significantly. Despite that, grammar-based generation is a field that is has not been explored a lot yet. Most related work focuses on subjects other than grammar-based generation. [17] mainly focuses on the use of gameplay grammars as a way of generating levels. According to [17] a gameplay grammar is 'a result of designer-expressed constraints, generates graphs of player actions, as well as their associated content. This action graph can then be used to determine a game level layout.'

The objective of our research is to use graph grammars in combination with gameplay grammars to generate a 3-dimensional dungeon-like level. In [3] they have a look at different methods of generating dungeons, including gameplay grammars, and compare those results. They concluded that 'gameplay-based control is now being successfully investigated, e.g., using missions, stories, player actions or even player models to control PCG (Procedurally Generated Content). This, in turn, provides a significant basis for games to automatically adapt to their players'. When it comes to Quality Assurance, [11] looked at different ways of measuring the quality of grammars for procedural level generation. In this paper they broadly explain how they generate their levels using graph grammars, possible obstacles whilst doing this and how they expand on the core concepts in grammar-based generation.

CASE STUDY

1. GRAMMARS

1.1. WHAT ARE GRAPH GRAMMARS?

Since we want to have a look at using graph grammars as a method for procedurally generating levels, it's a good idea to start looking at what graph grammars actually represent. The first thing to mention is the fact that a graph grammar is a derivative from a formal grammar (often written short as 'grammar'). A grammar is a set of rules and structures to define how to write/perform statements/actions that are valid for the system in question. Usually this is in a language-driven context, where the rules would define syntax and morphology semantics.

In the image down below, you can see how we convert a similar system from a formal grammar to a graph grammar.
The use of graph grammars introduces another level of complication to the system. This way we also have to define our rules in more detail. For example, if we have a look at the first example of the graph grammar representation, what happens if something was attached to S? Do we assign these to A now? Or do we disregard them completely?
To avoid confusion here, a well-formed graph grammar should explain what nodes and edges are replaced.

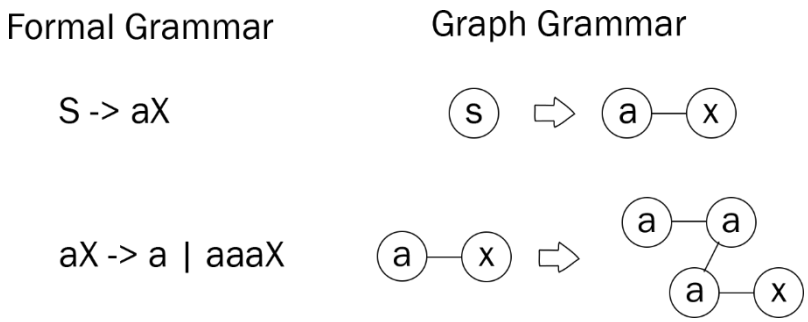

Figure 1: From Formal Grammar To Graph Grammar (Mark Gritter, 2019)

1.2. GAMEPLAY GRAMMARS

Now that we know the core concept of graph grammars, we can expand our knowledge and learn about gameplay grammars. In the paper "Designing Procedurally Generated Levels" by Roland van der Linden, Ricardo Lopes, and Rafael Bidarra this concept is explained. Gameplay grammars are quite similar to graph grammars, a gameplay grammar is also a set of rules, but they define the possible actions and interactions that can take place within a game. They specify what actions the player can perform and how the game state will change as a result of them. As a result of this, instead of the gameplay being influenced by the rules that define the geometry and content of the level, it is the other way around. This makes much more sense for the game designers. One of the downsides of this approach, however, is the fact that there's no universal set of actions. This means that for each game, designers will have to specify their own.

Another thing this paper mentioned, is the use of compound- and sub-actions. If you have a look at the example demonstrated below, you could have an action that says "Acquire Key". This would exist of the sub-actions "Kill an enemy" and then "Loot key from body". In this example, we're already using a compound action as well since the sub-action is another action with a sub-action. On top of this, we could also specify different options for completing an action. (specified at design & generation time, not at game time) This can increase variability and flexibility.

Action 1: **Acquire key**
Sub-actions: *Kill an enemy → Loot key from body*

Action 2: **Enter locked chamber**
Sub-actions: (***Acquire key*** *→ Unlock related door → M through doorway) || (Climb on roof → Enter chimney)*

Action 1: **Acquire key**
Sub-actions:
Option 1: if *Difficulty* == 25 and *Length* > 25
1.1: *Kill an enemy → Loot key from body*
1.2: *Distract enemy → Steal key*
Option 2: else
Look under doormat → Pickup key

Figure 2: Gameplay Grammars (Roland van der Linden et. al, 2013)

1.3. PROCEDURAL LEVEL DESIGN USING GRAPH GRAMMARS IN GAMES

Let's have a closer look at some games that already used graph and/or gameplay grammars to generate procedural level designs. Whilst I couldn't immediately find any game that already implemented the use of gameplay grammars, I did stumble upon the following paper: "Graph Grammars for Super Mario Bros' Levels" by Santiago Londono and Olana Missura. In this paper, they have a section where they propose a graph-based representation of Super Mario Bros Levels. They start off by separating the different gameplay elements the game has, namely platforms and item clusters. Where the item clusters actually represent sub-graphs. These could contain coins, power-ups, enemies…
On top of these two elements, they also introduce a new constraint, reachability. According to the algorithm they used, once all platforms and item clusters have been constructed, the algorithm evaluates what elements are reachable, and a "reach" node will be inserted in the graph.

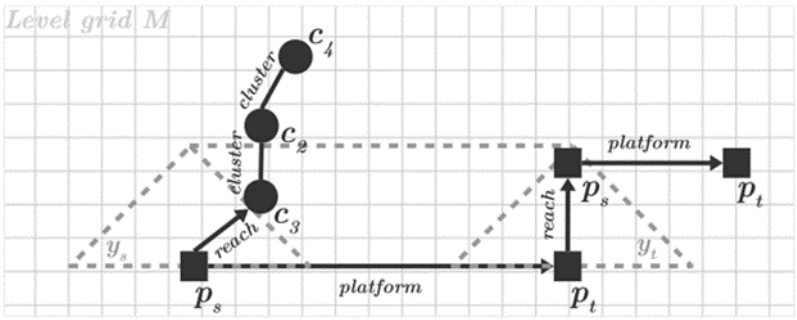

Figure 3: Reachability area of a platform (Santiago Londono et. al, 2015)

2. GETTING STARTED

2.1. WHAT TOOLS CAN WE USE?

So we know quite a bit about graph grammars and gameplay grammars by now. But the next step is still a big question mark: "How are we going to implement this?"
To properly discuss this, we need to have a look at the available options first.
The first 3 options that come to mind are the following:
- Unity
- Unreal Engine
- Raw C++ Project

Under the last category, we still have a few different options. For example, we could start from scratch. But we could also start from a Win32 engine Kevin (my supervisor) gave us. On the other hand, we also have access to the Overlord Engine (Engine used in Graphics Programming @ DAE) or the Elite Framework. (Framework used in Gameplay Programming @ DAE)

Before we can have a look at what engine/framework we'll be using, we need to think about what we are actually going to do. One thing is for certain: we need to find a way to easily create, edit and visualize the gameplay grammar graphs. So somehow, we need to create a tool that can handle these things. For this, there are multiple options available as well, here's a list of what I believe are the simplest options:
- Unity Editor Tool
- Unreal Engine Tool
- WPF Program

Obviously, we could start things off a lot simpler by using a text file of some sort that includes all the data. But I do believe that this will get confusing and messy once we start getting a bigger scope, so I want to make sure that everything is easy-to-use from the beginning.
For the scope of this project, I decided to stick with Unity for 2 main reasons. One of which is the fact that Unity has a very easy-to-set-up prefab system, which we'll be using extensively throughout this application. The other main reason why I have decided to go with Unity is the fact that there is much more documentation on how to create tools within the editor itself.

2.2. DEVELOPING THE TOOL

2.2.1. WHAT DOES IT NEED?

In order to properly understand what features our tool needs to have, we need to think ahead about the structure and approach of our project. One of the most important features of our tool is the fact that it's content needs to be able to be read from our generator. To ensure this, we will use Unity's scriptable objects to define gameplay actions, rooms etc. This way we can easily connect different actions to different rooms if needed. Whilst keeping room for both expandability and customizability.
To keep things clear and simple for the end-user, I decided to split the gameplay grammars from the graph grammars.
Whilst the gameplay grammars will decide the sequence of gameplay elements the players will encounter, graph grammars will be responsible for creating the layout of the rooms individually.

2.2.2. TOOL FRAMEWORK

Since we would like to use a system like Unity's prefab system for an earlier version of our procedural generator, I started looking into how I could create a tool that handles everything surrounding the graph systems.
I stumbled upon this GitHub repository that integrates a visual graph editor inside Unity:
https://github.com/Seneral/Node_Editor_Framework

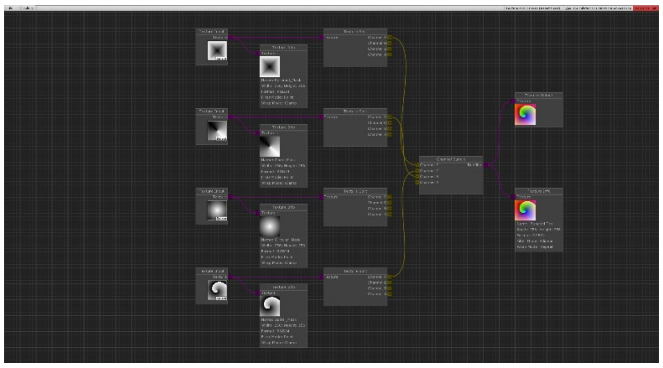

Figure 4: Node Editor Framework (Seneral, 2020)

Integrating this into the Unity project wasn't too difficult. However, now the system needs to be adapted to support our different graph systems. Before we get started on this, we need to think about this process a little more in-depth. This way we avoid major changes of the core graphs during development of the generator itself.

2.2.3. GAMEPLAY GRAMMARS

In terms of gameplay grammars, we do not need a lot of different node types, especially since the approach I chose uses scriptable objects. To give more control over the dungeon layout for the game designer, I stepped away from the previously known gameplay grammars, and chose to adapt it a little bit. In this application, we will convert the gameplay grammar system to a gameplay flowchart.
The most obvious and important node type we need is the Action node itself. This action node will contain a reference to a scriptable object that contains the data of that specific action. To keep things simple, this action will have a list of references to the different room types that contain this action. This way, we guarantee that the player will be able to or even forced to perform this gameplay action, whilst keeping it customizable enough with entirely different room types. Here's an example of a 3-room dungeon:

Figure 5: Action Nodes Inside A Gameplay Flowchart

This in itself already gives us the opportunity to fully create our flowchart, however, besides the use of different room types for each action, this is a quite linear way of working. To increase the feeling of interacting with a different environment each time, we will introduce a new node type: the "Option" node. This option will take in a Boolean, and depending on its value, continue down a different path. Using this system, we can make the level more dynamic. For example, we can introduce a difficulty parameter that will follow a more difficult flow than others. In the examples used for this application, instead of using a difficulty parameter, we are using random bools to simulate this effect. Here's an example of a system like this:

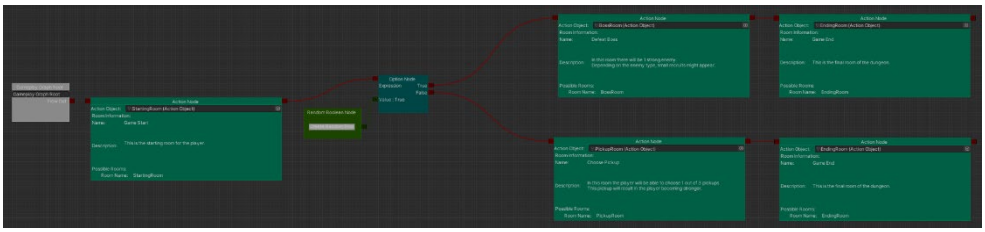

Figure 6: Increasing Randomness Using Option Nodes Inside A Flowchart

2.2.4. GRAPH GRAMMARS

When it comes to our graph grammar system, we need to be able to specify a scenario that gets adjusted to a desired result, based on a specific rule. Because of this, we can simply create one "rule" node, that defines the specific rule, possibly in a desired direction. This rule will then connect the starting scenario to its end result. To keep things simple and intuitive, we will represent a scenario with tile-types. In the figure down below, you can see an example of where we place walls around the entire room.

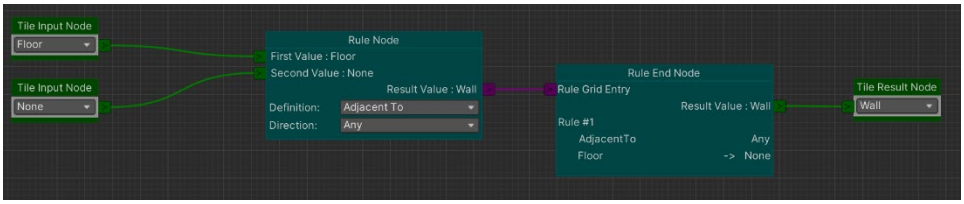

Figure 7: Graph Grammar Rule Definition

As you can see, we also have a Rule End Node. This way the generator only needs to fetch the Rule End Nodes, since these are connected to all information needed. Because of this, we are also able to expand our ruleset by using "AND" cases, which increases customizability and control by a lot. The following figure represents a combination of two rules to place doors in the room.

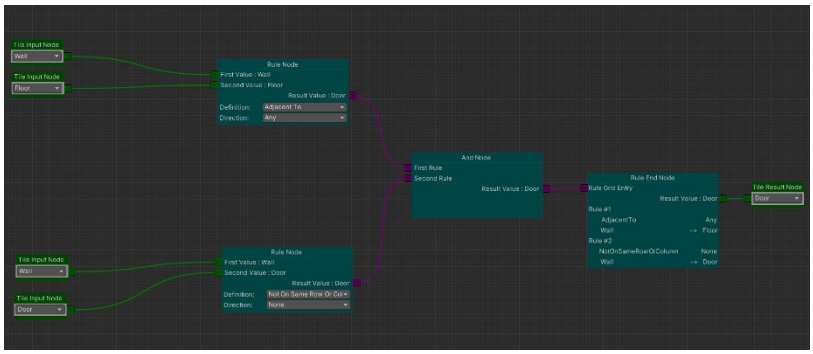

Figure 8: Increasing Rule Complexity For Graph Grammars

2.3. CREATING THE GENERATOR

2.3.1. GENERATING ROOMS

Now that we have our tool to create, visualize and edit our graphs, it's time to get started on the actual generation of the dungeon. The workflow we are using makes it possible for us to generate our rooms individually, and then link them to the dungeon itself later on. As specified previously, we will represent rooms as scriptable objects inside Unity. This raises the following question: "What data does this need to include?"

2.3.1.1. ROOM DATA

The first major thing that a room needs is the room size, to increase customizability we provide a minimum and a maximum vector, the generator will then select a random value that fits these criteria. On top of this, we need to have a reference to the props we would like to place in the scene. The downside of this approach is that you end up having lots of references to Unity prefabs inside of each room object. An optimization I applied here is to use an asset manager, that has all of these references, and is easily accessible from within our generator script.

The next important data we need is of course our grammar graph for this room. However, having one graph to represent an entire room might get a bit messy, and is not quite modular. To improve this, based on [11], I decided to create modules as a separate data object, which contains a reference to its grammar graph, as well as a sort order index to determine in what order the modules are executed. This way, we can reuse these modules for different rooms. For example, in my demo application, most of my rooms shared the module that handles wall and/or door placement.

2.3.1.2. GENERATING ROOM LAYOUT

2.3.1.2.1. GENERATION PROCESS

We already know that we are using modules that contain rules to generate our room layout. So the general process for generating our room layouts is quite simple. For every module, we want to loop over all the rules this module contains. And for each of these rules, we will go over all the tiles in our room layout to see what tiles can apply this rule. After deciding what tiles we can apply our rules to, we should still be able to specify a minimum and/or maximum amount of times this rule can be applied. To introduce this possibility, we will add a list of prop information to our room data objects. All this really needs for the moment, is a minimum and maximum amount of times you want this prop to appear inside this room.

Now that this is possible, once we have collected all of our tiles where we can apply the current rule, we will check if we haven't already placed too many of these props. And if not, we will select a random tile from the list where we will place the prop.
When it comes to actually placing these props, we will make use of Unity's prefab system. This makes it possible to give vertical offsets to our objects inside of the prefabs themselves. This way, we do not have to think about vertical placement during the generation process.

2.3.1.2.2. EXAMPLE

Let's have a closer look at this process using an example. In this example, we will be generating some sort of treasure room.
We know that we start with a randomly chosen rectangle and that the floors are already placed. The next thing we want to think about is the generation of walls. We will generate a wall everywhere a floor tile is adjacent to an empty tile.

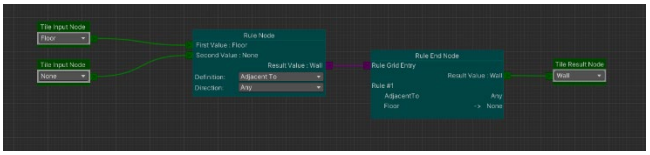

Figure 8: Wall Placement Ruleset

The next thing we want to add are doors. The rule for the doors is also pretty straightforward. We want to place a door where there is a floor object next to a wall. But we also want to make sure that the door is not standing right next to another door.

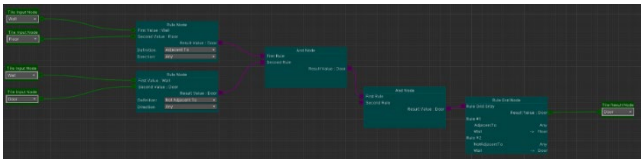

Figure 9: Door Placement Ruleset

To create a bit of lighting we will place some torches as well. For this example, we will place the torches anywhere on a wall, but not next to another torch, and not next to a door. Next to this, we will also introduce some obstacles to our room to make it look a little bit more interesting.

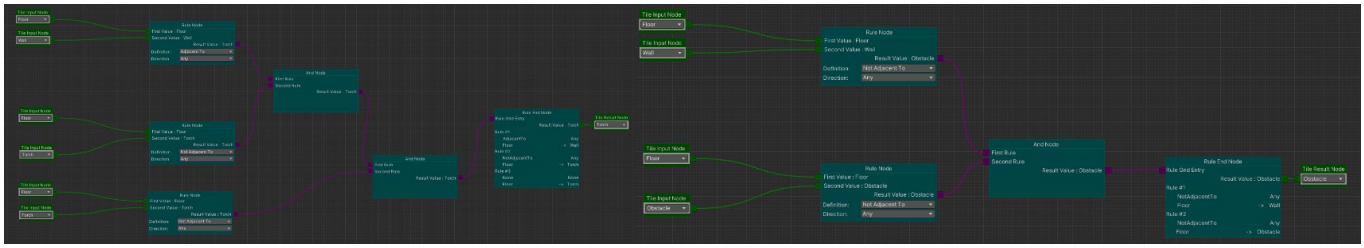

Figure 10 & 11: Generation Of Torches And Obstacle

Now that we have an appealing version of our room, it's time to place our gameplay elements. In this case, I chose to place the treasure chests anywhere in the scene, but not next to an obstacle, and not adjacent to any wall.

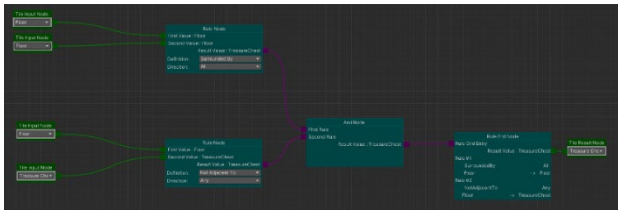

Figure 12: Treasure Chest Placement Ruleset

As you can see, this procedure is really easy to set up, whilst generating a fairly neat result. And because we are using these as modules, we can reuse any of the modules in any other room we want.

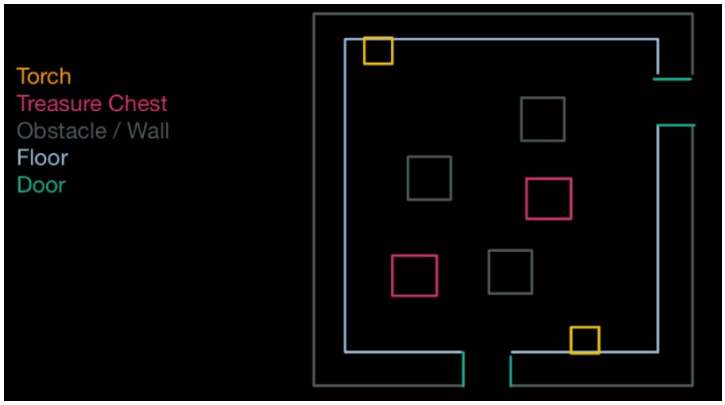

Figure 13: Example Result Room Layout

2.3.2. LINKING ROOMS

2.3.2.1. CONNECTING DOORS

Now that we are properly generating our rooms, the next step is to link them together. In our application, we'll be using doors to connect the different rooms. But how do we actually do this?
The first step is to find a door that is not yet connected for both rooms. Once we have access to these doors, the next step would be to check if they are on opposite sides of the rooms, so that they can be 'welded' together. If this is the case, all we need to do here is to transform the second room to the correct position, so that it lines up perfectly with the first room. To get this transformation vector, we can simply take the difference vector between the first and the second door, and translate the second room with this vector.

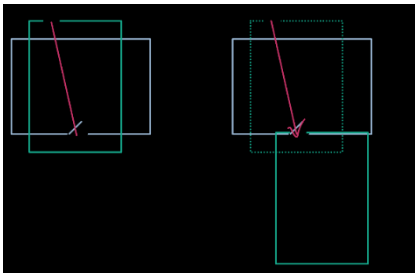

Figure 14: Room Translation For Connection

However, when the doors are not on opposite sides, this will cause the rooms to overlap. We already have control over the number of doors that will be placed, so we can guarantee that there will always be at least 1 door that will be able to connect. But they could be placed on the wrong sides. To solve this, we simply rotate that entire room 90° and re-evaluate if the doors are in the

correct position. This way, we will always have a possible solution within 4 tries. Once we have found our solution, we simply perform the exact same translation as mentioned above, and the rooms should be connected.

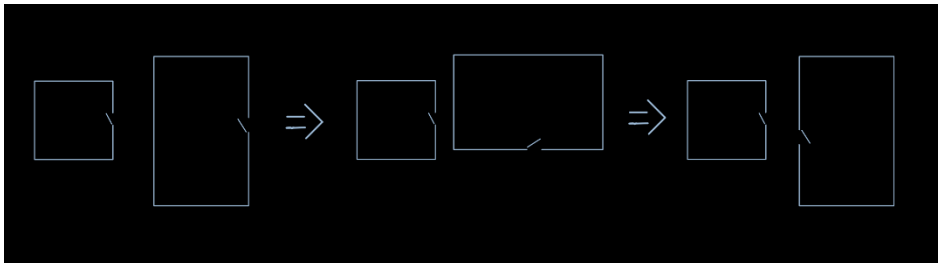

Figure 15: Finding Door Connections

2.3.2.2. AVOIDING OVERLAPPING ROOMS

Now that our rooms are correctly connecting with each other, we still have an underlying issue we have to solve.
When we already have a few rooms that are connected, there is a chance that the room we're trying to connect with the current door is too large, resulting in an overlapping area between 2 rooms. We could try to avoid this by re-generating that room with some extra criteria to make sure that it fits perfectly within the current grid. However, this is an extremely inefficient way of working. Whilst on top of this, there is still a chance that you're so close to another room that there won't be any room that can possibly fit this space.

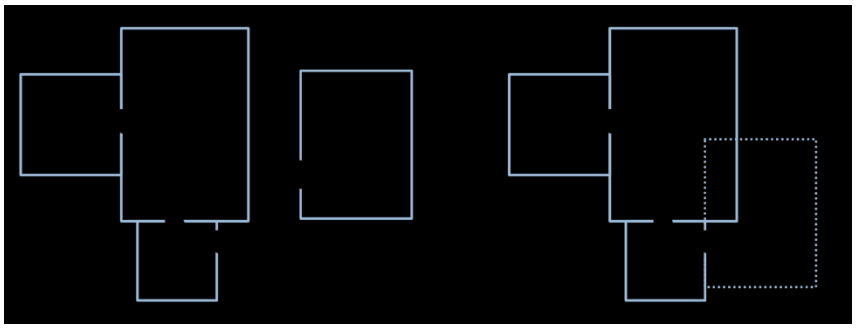

Figure 16: Overlapping Rooms

Instead of re-generating this room, we could add some kind of corridor that will make sure this room is able to fit.
Obviously, adding this extra hallway makes the generation process a little more complex. For example, one of the possible issues of adding these hallways is the fact that you might end up generating really long hallways, which we definitely want to avoid. Another issue we have to take into account when doing this is that hallways might also collide with other rooms, or even other hallways.

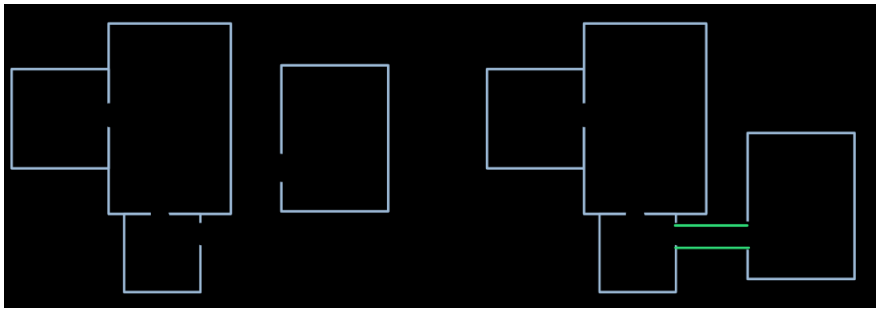

Figure 17: Hallway Generation

To try and prevent these issues from happening we'll have to introduce another level of complexity, namely height alterations. When generating the hallway, we can check if it is overlapping with anything else in our scene, and if so, we know that we should

either go upwards or downwards to avoid collision with the other object. Besides the fact that we are avoiding overlapping areas in our level, this also gives the level a lot more variety, and is visually much more appealing than before.

2.3.2.3. HANDLE EMPTY DOORS

Up until now, we have mainly been focusing on following our main gameplay flowchart to generate our dungeon. However, this way we are creating a very long path of rooms that are connected to each other, but this doesn't quite have the option to generate multiple side paths or something similar to that.
In our room data objects, we can already specify how many doors we want that room to have, but currently, nothing is happening to the doors that are not used in our main path. To handle these, when the main path has finished generating, we will go over every door that is not connected yet, and generate a room to link this door to. This leaves us with one more question to solve, what room type will we generate here?
To give even more control to the game designer in question, we add a list of possible side rooms to the properties of a room data object. The generator will then select a random room out of this list, and repeat this process until there are no more doors that are not connected.

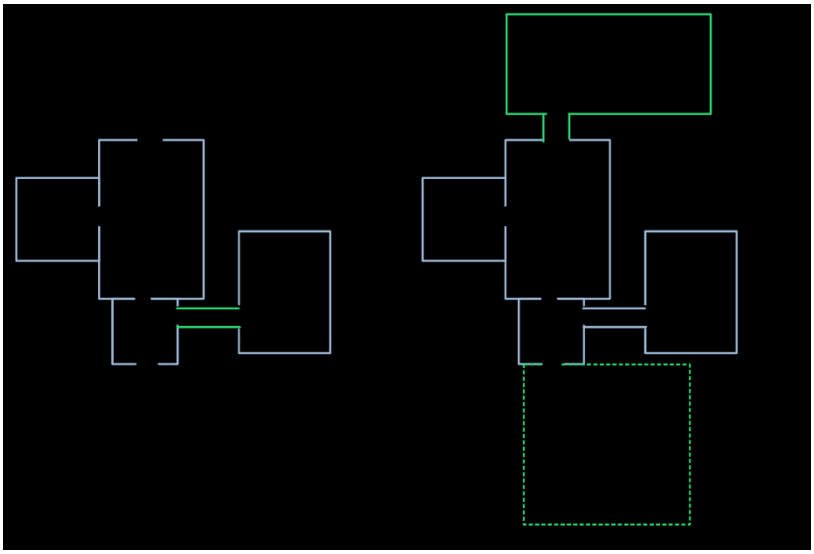

Figure 18: Adding Rooms To Non-Connected Doors

EXPERIMENTS & RESULTS

Using the generator I was able to make a small demo-application. The game in this showcase is a 3-dimensional roguelike dungeon crawler. In this example you have to defeat enemies and bosses to get your way to treasure rooms and/or pickup rooms that make you stronger. As a unique gameplay feature there are also challenge-rooms, where you could start a challenge that potentially will make your character stronger. To make use of our generation system, this example also introduces the perma-death mechanic. This way, when you die, you have to start over every time. But as you can see in the figures below, because of the huge amount of variation our generator provides, chances of getting dungeons that are similar to each other is very slim.

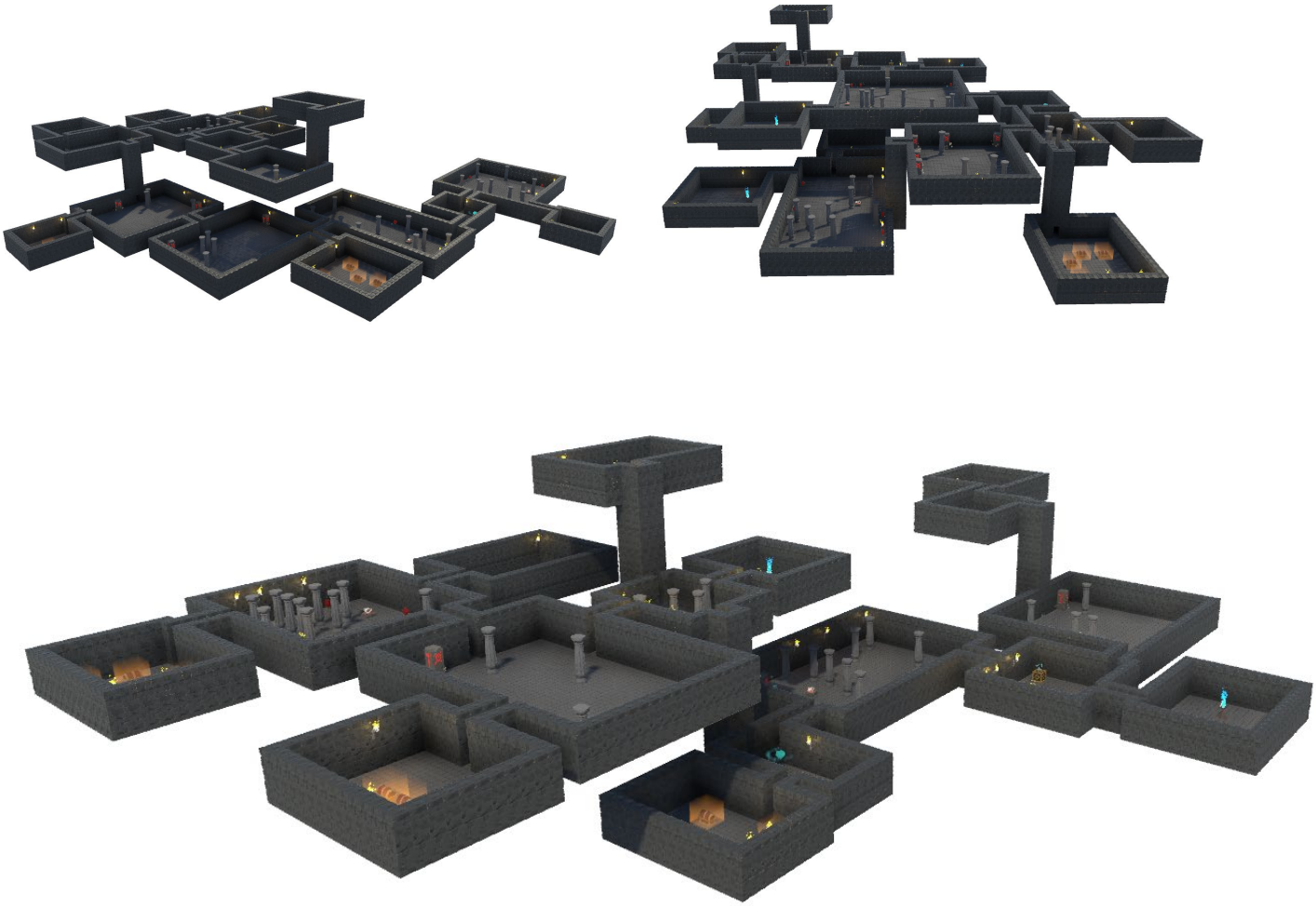

Figure 19-21: Demo Application Results

When we look at these results in a bit more detail, you can see that we are making use of the graph grammar modules extensively. Not only do we use the same modules to generate the walls and doors in every room. But we also have some rooms that place pillars all over the room, which is more concentrated in some rooms, and less in others. The placement of the torches are also shared over most rooms. This shows our modularity really well, which increases our production speed by a lot.

CONCLUSION & FUTURE WORK

The combination of gameplay grammars with graph grammars is definitely an interesting way of working. It massively increases production speed, whilst still giving the designers a lot of control over their levels. When it comes to the gameplay elements, this is a really good system to use. In some cases, the generator might even produce a result that the game designers did not think of in the first place, possibly giving them new ideas. However, despite this, the levels still do not feel 'natural' compared to hand-made levels, more specifically on a visual level.

When it comes to quality assurance, it is no secret that procedural generation tends to struggle with this. Nevertheless, using the graph grammars for the individual room generation, we can already remove some corner cases. For example, when you would have 2 doors placed next to each other, you can put this inside of the ruleset to avoid these cases. Obviously, there will always be corner cases, whether it is because of something the game designer overlooked, or the developer of the generator in question. This is something that is currently planned for the future.

Something I would still want to take a look at is the game saving process. As of right now, the generator will generate a completely random level each time. But if you would like to save your game in its current state, you would have to save the entire generated level with it. To optimize this, a common solution is the use of pseudo-randomness. In short, the generator will act based on a given seed. This means you only have to save that seed in order to be able to re-create that exact world. This is something I would like to implement in my generator as well.

BIBLIOGRAPHY

[1] de Kegel, B., & Haahr, M. (2020). Procedural puzzle generation: A survey. In *IEEE Transactions on Games* (Vol. 12, Issue 1, pp. 21–40). Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/TG.2019.2917792

[2] Millington, I. (n.d.). *AI for Games THIRD EDITION*.

[3] van der Linden, R., Lopes, R., & Bidarra, R. (2014). Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, *6*(1), 78–89. https://doi.org/10.1109/TCIAIG.2013.2290371

[4] *2019 IEEE Conference on Games (CoG)*. IEEE.

[5] Londoño, S., & Missura, O. (n.d.). *Graph Grammars for Super Mario Bros * Levels*.

[6] Nwankwo, G., Mohammed, S., & Fiaidhi, J. (2017). Procedural Content Generation for Dynamic Level Design and Difficulty in a 2D Game Using UNITY. *International Journal of Multimedia and Ubiquitous Engineering*, *12*(9), 41–52. https://doi.org/10.14257/ijmue.2017.12.9.04

[7] Short Tarn Adams, T. X. (n.d.). *Procedural Storytelling in Game Design*.

[8] Viana, B. M. F., & dos Santos, S. R. (2019). A Survey of Procedural Dungeon Generation. *Brazilian Symposium on Games and Digital Entertainment, SBGAMES*, *2019-October*, 29–38. https://doi.org/10.1109/SBGames.2019.00015

[9] *Procedural Generation in Game Design*. (n.d.).

[10] Totten, C. W. (n.d.). *An Architectural Approach to Level Design*.

[11] van Rozen, R., & Heijn, Q. (2018, August 7). Measuring quality of grammars for procedural level generation. *ACM International Conference Proceeding Series*. https://doi.org/10.1145/3235765.3235821

[12] Ashlock, D., Annual IEEE Computer Conference, IEEE Conference on Computational Intelligence and Games 2013.08.11-13 Niagara Falls, & CIG 2013.08.11-13 Niagara Falls. (n.d.). *2013 IEEE Conference on Computational Intelligence and Games (CIG) 11-13 Aug. 2013, Niagara Falls*.

[13] ACM Digital Library. (2011). *Second International Workshop on Procedural Content Generation in Games : co-located with the 6th Internal Conference on the Foundations of Digital Games : June 28, 2011, Bordeaux, France.* ACM.

[14] Gellel, A., & Sweetser, P. (2020, September 15). A Hybrid Approach to Procedural Generation of Roguelike Video Game Levels. *ACM International Conference Proceeding Series*. https://doi.org/10.1145/3402942.3402945

[15] Valls-Vargas, J., Zhu, J., & Ontanon, S. (2017). Graph grammar-based controllable generation of puzzles for a learning game about parallel programming. *ACM International Conference Proceeding Series*, *Part F130151*. https://doi.org/10.1145/3102071.3102079

[16] Canossa, A., & Smith, G. (n.d.). *Towards a Procedural Evaluation Technique: Metrics for Level Design*. http://www.pouetpu-

[17] van der Linden, R., Lopes, R., & Bidarra, R. (2013). *Designing Procedurally Generated Levels*. www.aaai.org

[18] Compton, K., & Mateas, M. (2006). *Procedural Level Design for Platform Games*. www.aaai.org

[19] van der Linden, R., & Bidarra, R. (2013). *Designing procedurally generated levels, Virtual Coaching and Storytelling technology for post-traumatic stress disorder treatment (VESP) View project*. https://www.researchgate.net/publication/288320122

[20] Christopher W. Totten. (2017). *Level Design, Processes and Experiences.*

[21] Christopher W. Totten. (2019). *An Architectural Approach to Level Design, Second Edition*.