Graph Grammar for Unity

A project for the UNIMI course “Artificial Intelligence for Video Games”

By Gianluca Iacchini

September 2023

Introduction

A grammar is a set of *production rules* used to rewrite a system into another. Each rule is of the form

LeftSymbol(s) $\rightarrow$ ***RightSymbol(s)***

Production rules specify a replacement of the left symbols (*left-hand side or LHS*) with the symbols on the right (*right-hand side or RHS).*

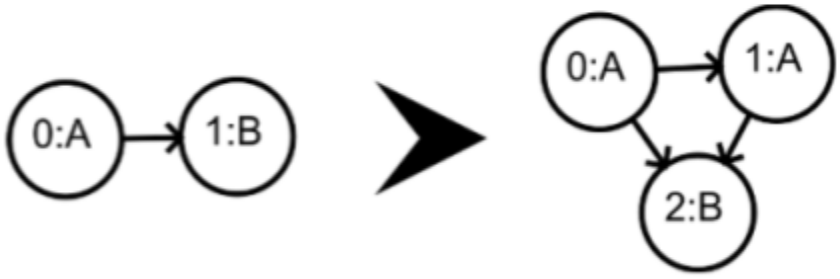
When talking about *Procedural Content Generation (PCG)*, Grammars are usually performed on strings and the production rules take the following form

$$A \rightarrow AB$$
$$B \rightarrow b$$

Grammars can be used with many different types of systems and are not restricted to strings only. Graph grammars are especially useful for video game PCG, as they can be utilized to generate many different aspects of the game such as spaces and missions.

Graph grammar production rules

Production rules for graph grammar usually look like the following



An example of a production rule for graph grammar.

Similarly to other grammars, production rules for graph grammars are composed by a left-hand side (the nodes and edges we want to replace) and a right-hand side (the nodes and edges we want to insert).

Each node of a production rule contains two pieces of information: a *symbol* and an *identifier.*

Identifiers

Symbols have the same purpose as those of other grammars, they are used to match the content in left-hand side of the rule with the given input and replace it with the content of the right-hand side

Identifiers are an additional piece of information that we need in order to apply the rule correctly. Let's see an example by looking at the previous image.

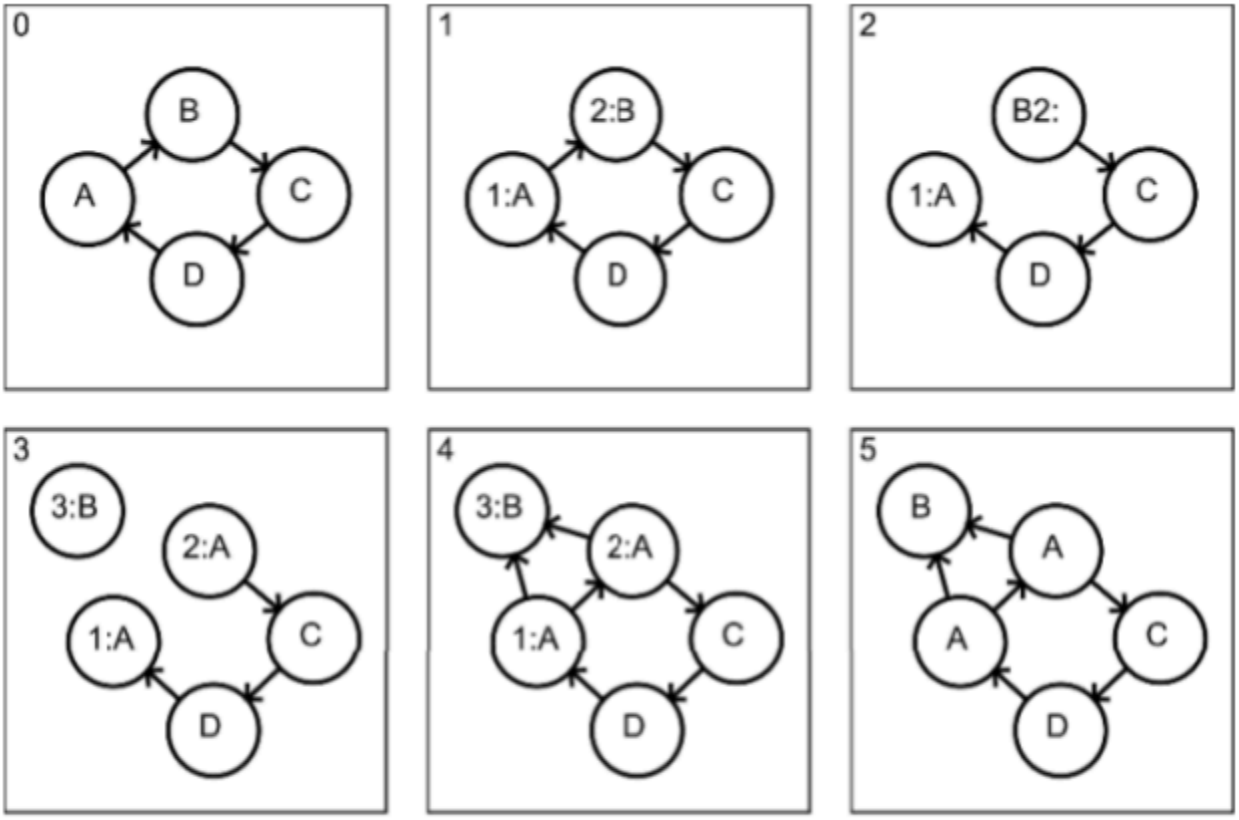
The production rule states that the node with *ID*:*0* on the left-hand has a symbol *A,* on the right-hand side the node with *ID:0* also has a symbol *A*, therefore this node does not need to be changed.
The node with *ID:1* however has a symbol *B* on the left-hand side and a symbol *A* on the right-hand side. It is still the same node (since the identifier is the same) but its content has changed from *B* to *A* .
Finally we have a node with *ID:2* on the right-hand side that we didn't have on the left-hand side, this simply means that a new node has to be created.

There are multiple ways to transform the graph from the left-hand side to the one on the right-hand side and without identifiers we wouldn't know which one to choose.
If we didn't have identifiers the quickest way to perform such a replacement would be to simply add a new node with symbol *A* and link it to the other two nodes, this is obviously very different from our production rule where we explicitly declared that we want the node with symbol *B* to change to *A* and to add a new node with symbol *B*.

Applying a rule

Given an initial graph *G* and a production rule we want to apply to it, we have to perform the following steps:

1. We find a subgraph of *G* that matches the left-hand side of the rule.
    a. We mark the subgraph of *G* by copying the identifiers of the nodes from the left-hand side to the matched nodes
2. We remove all the edges from the marked nodes in the subgraph
3. We transform the subgraph by transforming the marked nodes in their corresponding nodes on the right-hand side, adding the nodes on the right-hand side that are not present in the left-hand side and removing the marked nodes that have no corresponding node on the right-hand side.
4. We copy the edges as specified on the right-hand side
5. We remove all identifiers from the marked nodes in the subgraph.



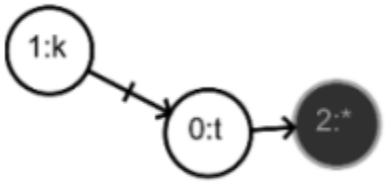
From Procedural Content Generation in Games (Shaker et Al.)

From the previous example we see that the nodes with symbols *A,B* of the graph have been matched with the left-hand side of the production rule even though they have two connections (respectively from node *D* and to node *C*) that are not present in the rule. This is because, unless specified, the production rule is matched to a subgraph that has at least the same edges as those present in the rules.

This shows why identifiers are very important. When we apply the rule, the node with symbol *B* in the initial graph is transformed into a node with symbol *A* but it **keeps its old connections**.

Edges

Just like nodes, edges can also have a symbol and be matched accordingly, different edges are usually represented by different arrows



The two edges shown here are different from each other. If this graph was used as a left-hand side rule then both nodes and edges must be matched accordingly to a subgraph in the given graph.

The * Symbol

In the previous image we saw that the node with *ID:*2 has a *"*"* as a symbol. This is a special symbol whose meaning is that it can be matched with any other symbol.

Using Graph Grammars in Unity.

Unity does not have a built-in system to create graph grammars, therefore the goal of this project was to create a tool that would allow it.

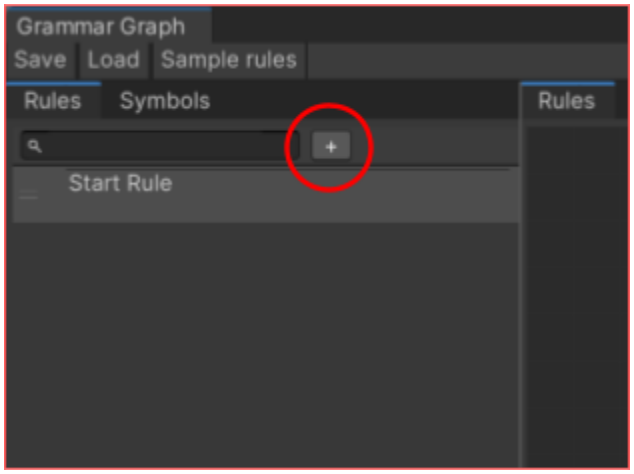
While string grammars can be built in the inspector (at least for simple cases) or made entirely through code, graph grammars are very tedious to implement without some kind of user interface; therefore a custom editor window was created in order to allow for an easier creation and testing of graph grammars.
In my project the tool can be found under *Windows* *Grammar Graph*

Graph Grammar Tool Description

1. Left-hand side of the selected rule
    - Nodes that will be matched in the subgraph for the selected production rule
2. Right-hand of the selected side
    - Nodes that will replace the matched nodes in the subgraph for the selected production rule
3. Production rule list / Symbol List
    - Shows all the production rules and allows to create new ones.
    - When the “Symbols” tab button is selected shows the Symbol list instead
4. Save / Load/ Test Rules
    - Allows to save the set of production rules and symbol list or to load one.
    - The “Sample rules” button tests ALL the rules created and generates a graph example
5. Switch between production rules interface or the graph generated using the “Sample rules” button

Creating a rule

If you wish to create a new rule simply click on the “+” button when the “Rules” tab is selected.



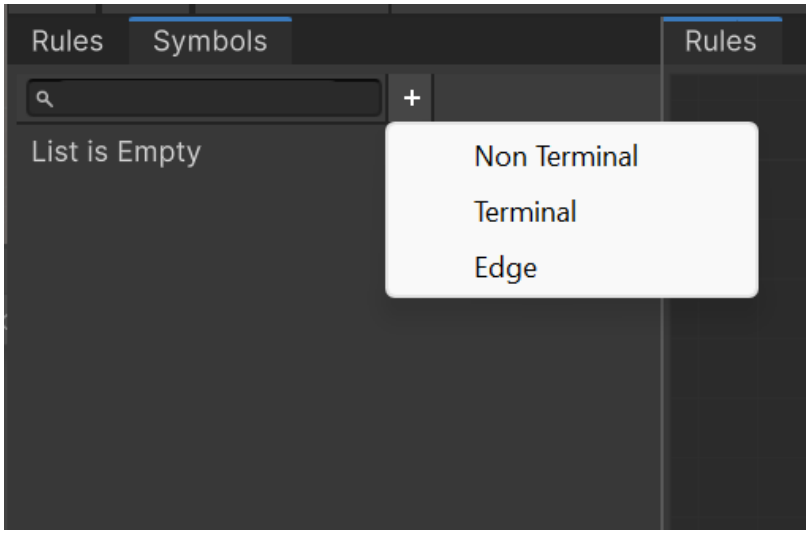
When opening the tool, a new grammar graph is created with a starting rule already present and ready to be modified.

Adding Symbols

Before we can add nodes to the left-hand side and right-hand side of our rule, we must first define a set of symbols to use.
To do that we must switch to the Symbols tab by clicking on the “Symbols” tab button.

Once we have switched to the Symbols tab, we can add new symbols by clicking on the “+” button. The available symbols type are Non-Terminal, Terminal and Edges

Terminal and Non-Terminal symbols are the same as those used for other grammars, where we have that a Non-Terminal symbol can be expanded into one or more terminal and Non-Terminal symbols by applying the production rules.
The *Edge* option allows the creation of new edge symbols. Edges do not have the distinction between terminal and non-terminal, each edge can be replaced with any other.



Editing the left-hand and right-hand side

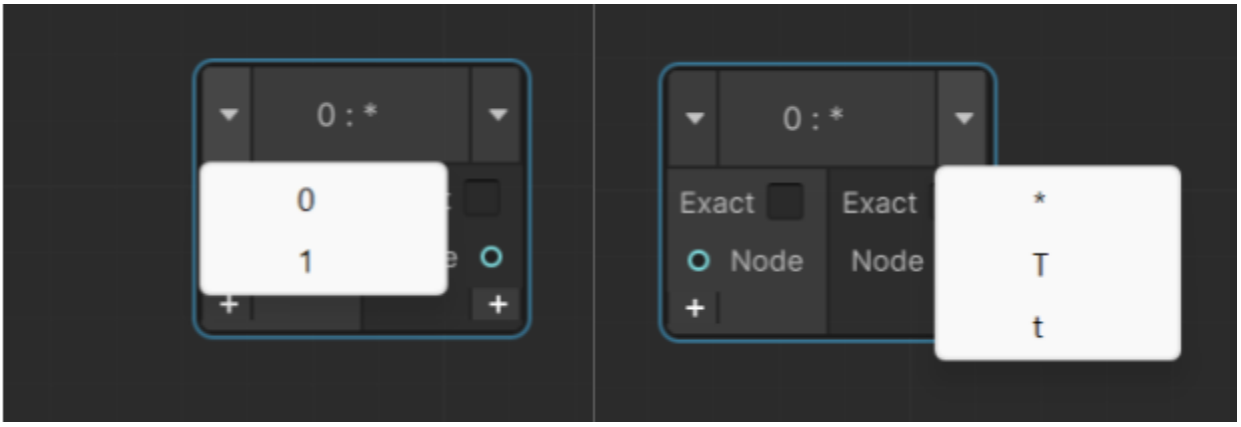
Once some symbols have been added, we can proceed to edit the left-hand side and right-hand side of a rule.
First click on the “Rules” tab button to switch back to the production rule list. Then click on the production rule you want to edit.

A node is already present on each side with a default *ID* of 0 and the default symbol “*”. To add a new node right click on an empty space and select the “Add node” option from the menu.

Editing IDs and Symbols

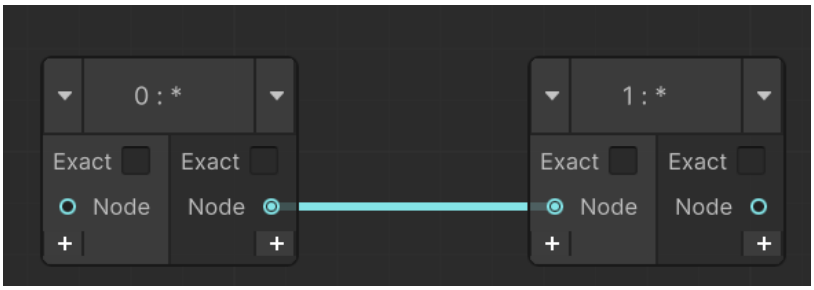
In order to change the ID of a node, simply click on the dropdown menu present on the top left of the node and choose the id from the list. Note that the number of IDs is the same as the number of nodes present on that side.

Similarly, to change the symbol of a node simply click on the dropdown menu on the top right of the node and choose the symbol from the list. These symbols are the one you added to the symbol list (excluding the edge symbols which will be explained shortly) with the addition of the wildstar symbol “*”.



Editing Edges

To link two nodes together simply click on the right port of a node and drag the line to the left port of the node you want to connect to.

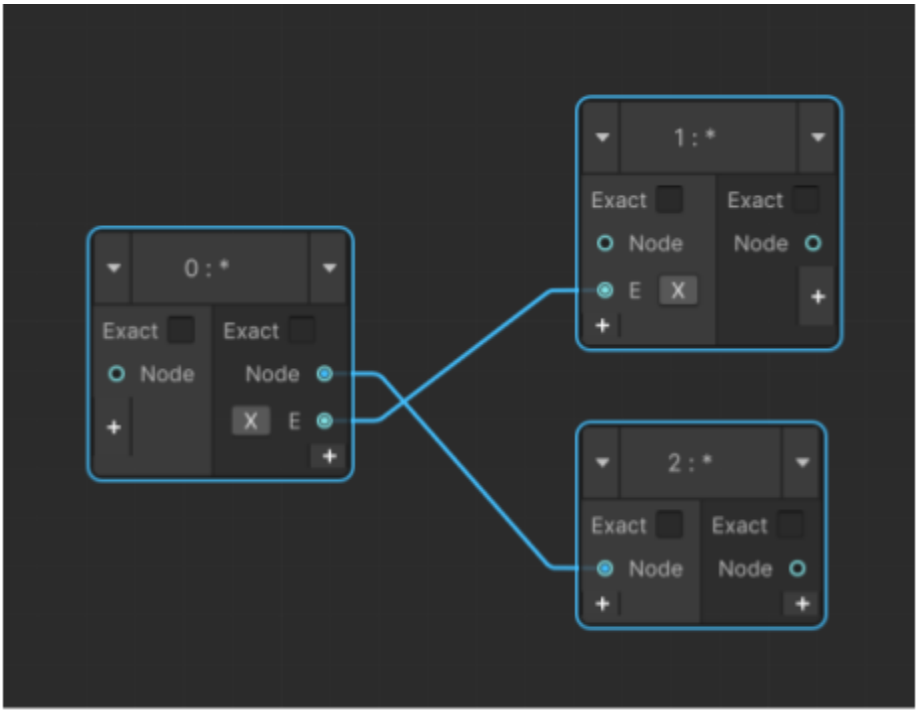


Note that the graph **is directed**, therefore the previous edge is a directed edge from node *ID:0* to node *ID:1*

Adding different Edges

To add different edges, simply click on the “+” button on the lower left or lower right side of the node and choose the edge from the list, just like symbols the edges from this list are populated with the one you created on the Symbols tab.
This will add a port whose side depends on which button you clicked, if you add a new edge on the left side, this means that this node can be reached by other nodes with one or more edges of the specified type.

Edges are not interchangeable, which means that you cannot link two ports whose symbols are different. Furthermore to connect two nodes you must add the correct ports on both nodes on the corresponding sides if you wish to connect them.

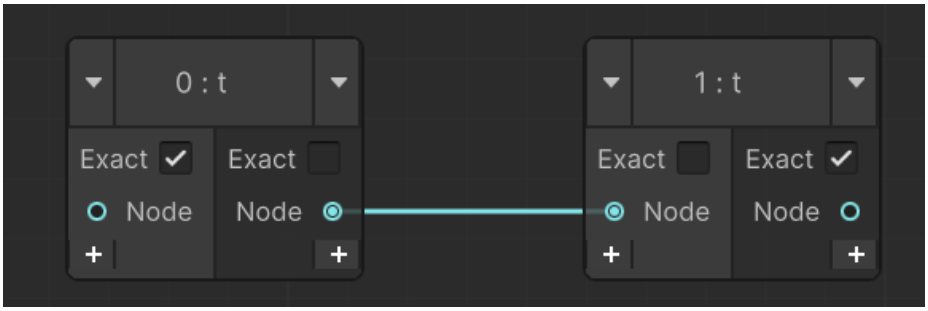


Ports with edge symbol “E” cannot be connected to the default “Node” port. Furthermore in order to connect node ID:0 and node ID:1 with an edge of type “E”, then both nodes must have an “E” port

The “Exact” checkbox

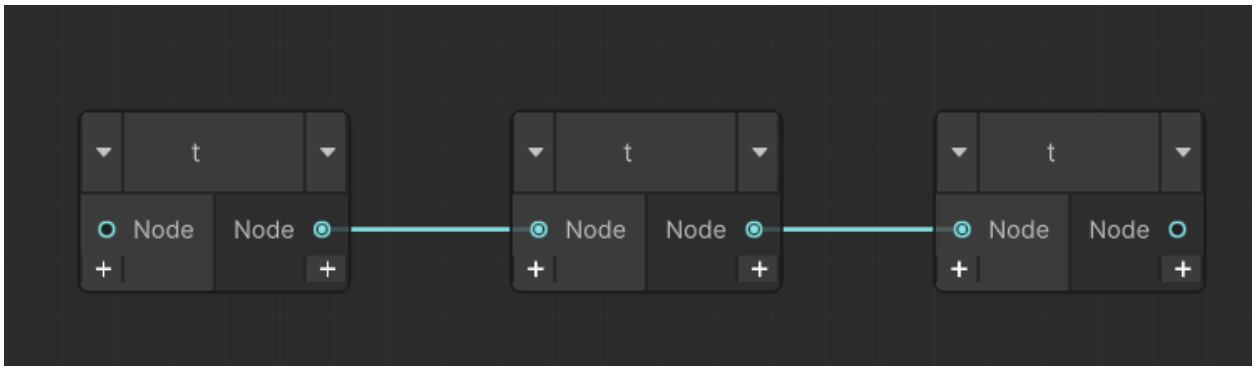
As said earlier, nodes on the left-hand side are matched to nodes of the input graph if the nodes on the graph have at least the same edges and symbols as those in the left-hand side of the production rules.
We also saw how nodes in the input graph can have additional connections that are not present in the left-hand side rules but still be selected as a match; this was the case with the first example.

However sometimes it is useful to restrict the connections exactly to those present in the left-hand side rule, for example

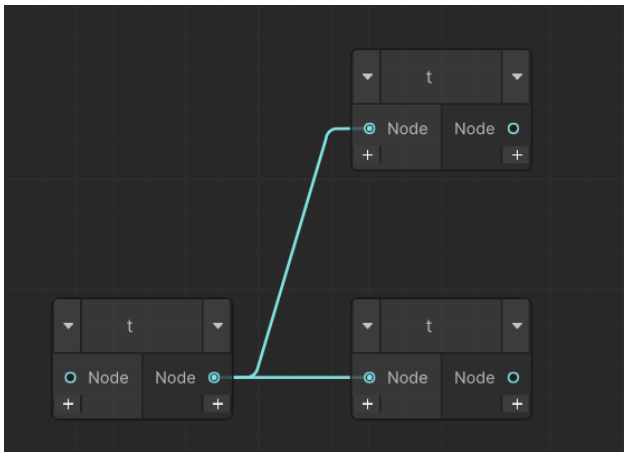


This means that to match the two nodes the input graph must have a subgraph with a node with symbol “t” **who has exactly 0 connections to it**, connected to **at least** one node with symbol “t” **who has exactly 0 connections from it**.

For example, the following graph would not match, because if we match the left node with node 0 then the node in the middle cannot be matched with node 1 (as it has more than zero output connection), similarly if we match the node on the right side with the node 1 then we can’t match the node in the middle with the node 0 (as it has more than zero input connection)



Let’s look at this other graph instead



Not only can this graph be matched to the previous rule, it actually has two match, one where the node 0 is matched to the node on the left and the node 1 to the node on the upper right, and another match where the node 0 is matched to the node on the left and the node 1 to the node on the lower right.

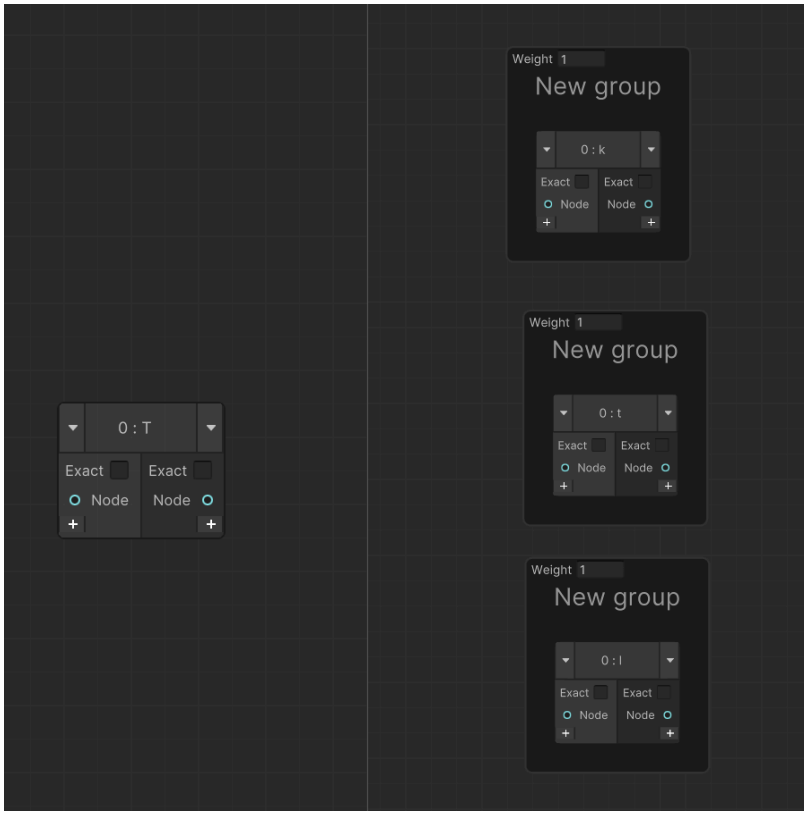
The reason why this is a match is because we only specified input connection for node 0 and output connection for node 1, but we haven’t restricted how many output connections can node 0 have or how many input connections can node 1 have.

Adding randomness

In order to produce interesting results, we can add a bit of randomness to the production rules. For example the following production rule

States that the node *ID:0* will have its symbol changed to either *k,t* or *l* chosen randomly (with equal chance).
We can achieve this in the tool by creating a new group by clicking on the background of the right-hand side of the selected rule and selecting “Add group”.
Each group can be thought of as its own right-hand side of the production rule, one group will be chosen randomly and the nodes that matched the left-hand side will be transformed into the nodes inside the chosen group, all the nodes present in the other groups will be ignored.

For example, the previous production rule can be expressed as



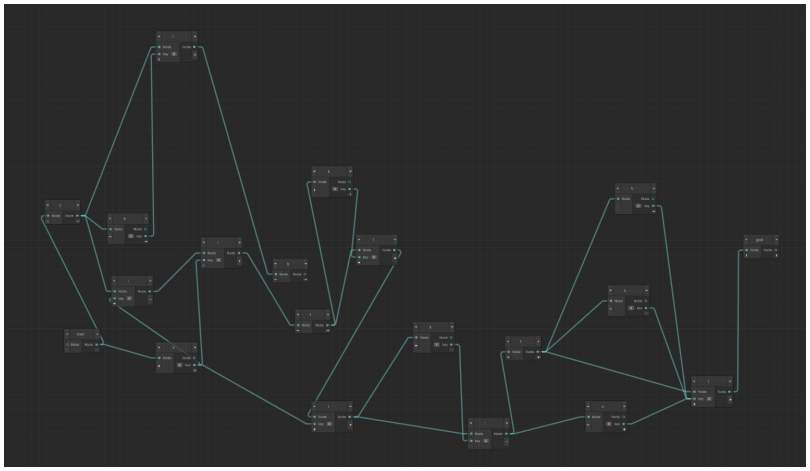
The textarea “Weight” inside each group indicates how likely it is for that group to be chosen, each group shown here has a weight of one, therefore each group has a chance of being selected of

$$\frac{group\ weight}{total\ weight} = \frac{1}{(1+1+1)} = 0.33$$

Testing the rules

Testing the production rules can be done simply by pressing the “Sample Rules” button on the top left of the window.
This will switch to the “Result” tab with a graph generated by applying **all** the production rules. The graph is generated as follows:

1. The initial graph *G* is generated as a copy of the left-hand side of the first rule from the rule list
2. The first production rule from the list is chosen as the active production rule *P*.
3. If there is a match for the active production rule in *G*
    - We apply the rule to *G,* transforming it: *G = P(G)*
    - We go back to step 3
4. If there is not a match
    - We select the next production rule in the list as the active production rule *P*
        - If we reached the end of the list we are done
        - Otherwise we go back to step 3



N.B: The sample graph is shown by separating the node to avoid overlaps, however the edges might still be crossing.

Saving the rules

The set of production rules and symbols can be saved by clicking on the “Save” button on the top left corner of the window. A pop up window will then appear where you can choose the file name and location for the file.
This will create a scriptable object of type *GGSaveDataSO* with an extension of “*.asset”*.

N.B: The specified location is only for the file that you can import and use by using the helper classes explained later. Most of its data is actually saved in the “Assets/GrammarGraph/GrammarGraphs” folder, therefore please DO NOT choose that specific folder as your save location as it could lead to unexpected behaviors.

Using the Graph Grammar

Once the graph grammar has been saved it is ready to be used.
The graph grammar file saved **is not** a graph, it is a file containing the set of production rules and symbols that can be used to generate one.
To create a graph an helper class was made called *GGBuilder*. This class has helper methods that take as input one or more production rules and an input graph (if no input graph is given then one is created by copying the left-hand side of the first rule, as explained earlier).

The *GGBuilder* class then tries to apply all the production rules it has received and outputs a generated graph.
The output graph is of type *GGGraph,* which contains nodes, edges and groups of types *GGNode, GGEdge, GGGroup*. Each class has its own helper methods to manage the data it contains.

LHS Matching

The most important and most difficult step so far was to implement the matching between the graph on the left-hand side of a specific rule and a subgraph on the input graph.
This is a subgraph isomorphism problem which I tackled using a modified version of the VF2 algorithm to also account for edges with different symbols.

Generating a Dungeon

As part of my project I implemented a dungeon generator. The production rules are mostly taken from *Procedural Content Generation in Games (Shaker et al.*) with some minor modifications.

Symbols

The list of symbols used for the dungeon are as follows:

- **Non-Terminal**
  - **S** - Start symbol used as a starting point for the generation of the graph
  - **T** - Represent a task that need to be done in order to progress to the next room
- **Terminal**
  - **start (or e in the book)** - Represents the starting point of the graph (i.e. entrance of a dungeon, first quest in a questline)
  - **goal** - Represents the ending point of the graph
  - **t** - Generic unspecified task (kill monsters, collect items, etc.)
  - **k** - States that need to be visited in order to access locks

    - **l** - Lock, a task that can only be completed by first collecting the corresponding key(s)
    - **b** - Treasure, a room where a treasure or other rewards can be found
- **Edges**
    - **node** - Directed edges identifying which nodes can be reached by other nodes
    - **key** - Edges that indicate for each key the corresponding lock(s)

About keys and locks

In the project a key - lock relationship is shown using the aforementioned *key* edge, in the book this is represented as an edge with a crossed line.

A key-lock relationship states that the key unlocks that lock, it does **NOT** mean that the lock is reachable by the key room (e.g. there is no actual path or corridor created between the key and the lock room, only the relation that the key is needed to unlock the lock)

Key and locks provide an unique relationship that helps to make the dungeon feel less like it is procedurally generated. This is because keys and locks can be represented as so much more than only their literal sense. When constructing a dungeon using PCG, keys and locks can assume many different form:

- An actual key that the player needs to collect to unlock an actual lock
- A boss can act as a key, when you defeat it a new area of the game opens up (lock)
    - Similarly a boss can act as lock, you can only defeat it by grabbing a specific weapon (key)
- An escort mission where an NPC (key) is required to proceed
- Blockages in the doors / roads (locks) that can only be cleared by grabbing a specific tool (key)

Additionally, locks and keys are not restricted to a 1:1 relationship, a key can unlock multiple doors and a door can be unlocked by multiple keys.
Time can also be used, for example a potion of fire can act as a key where a river of lava acts as a lock, but the player must be quick or the effect might expire too soon.

Production Rules

The production rules used to generate the dungeon will now be briefly explained, these are taken from *Procedural Content Generation in Games (Shaker et al.).*

start

Once generated the dungeon must have an entrance that eventually leads to a goal.
A key and a generic task are interposed between the entrance and the goal, their relation will be solved later

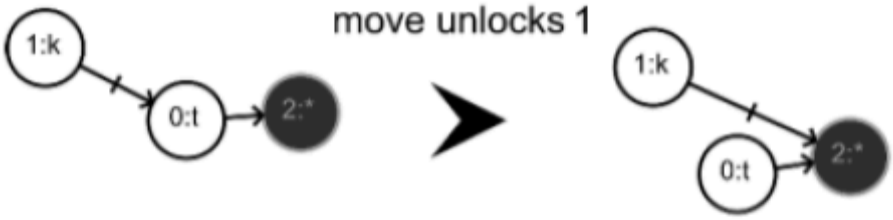
grow

We add more generic task to do before the goal . This rule can always be matched, no matter how many times it is applied; for this reason the algorithm has a (modifiable) maximum cap for the number of times that a single production rule can be applied before switching to the next

lock goal

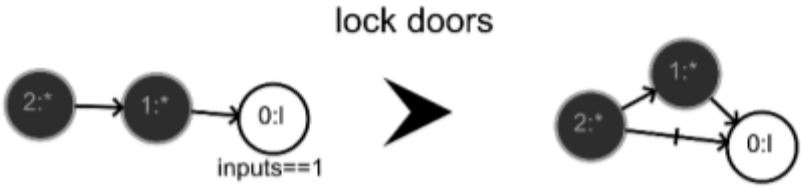
We lock the goal, which means that the player in order to reach the end must solve at least one key-lock relationship.
We know from the previous rule that the goal is always after a task and at least a key is present after the entrance

Tasks are randomly substituted with either a key, a generic terminal task or a lock

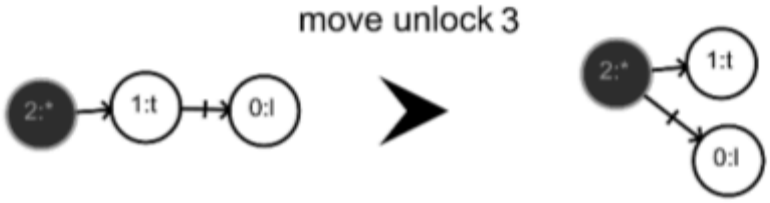


A key can only unlock locks. If a key is unlocking a generic task then move to the next one. We know that the last task before the goal is a lock, so we are sure that we will eventually land in one

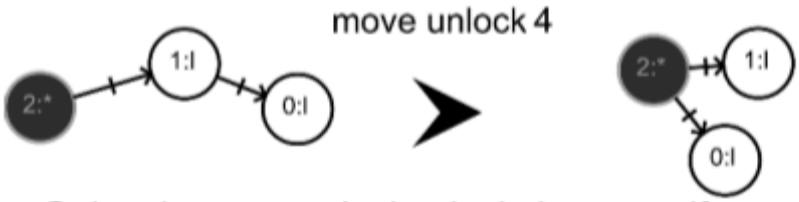
Same as before, a key can only unlock a lock, however we have found another key that is unlocking a lock, so we can have this key point to that lock too. This means that we must collect both keys to unlock that specific lock.



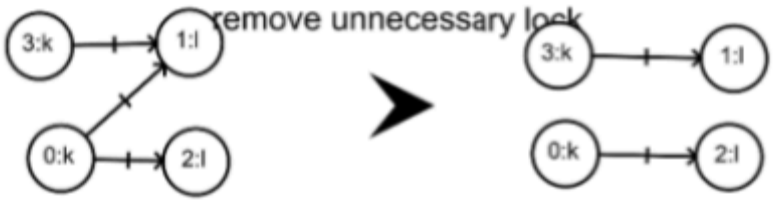
inputs==1 means that node 0:l has exactly the shown input. Therefore this lock cannot be unlocked has it does not have a key relationship. We create a new key relationship by looking in the previous states



Only a lock can point to a key, therefore we remove the key relationship from the generic terminal task and look for a look in the previous states

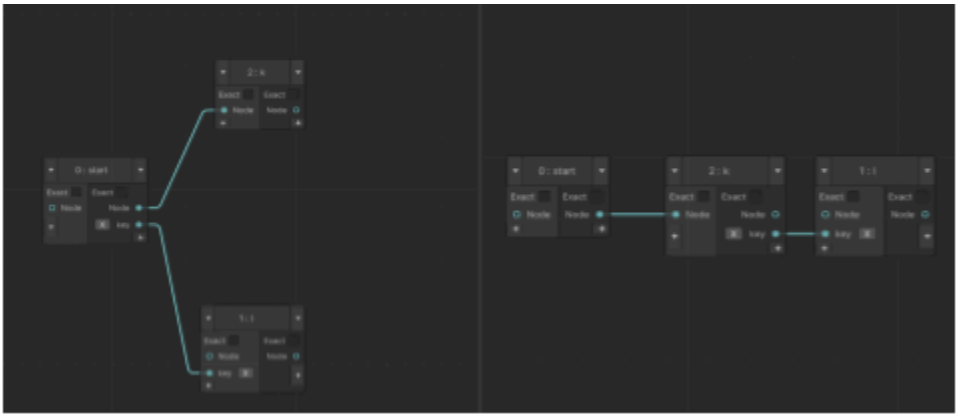


Only a key can unlock a lock, however if something is unlocking the node 1, then it can also unlock the node 0

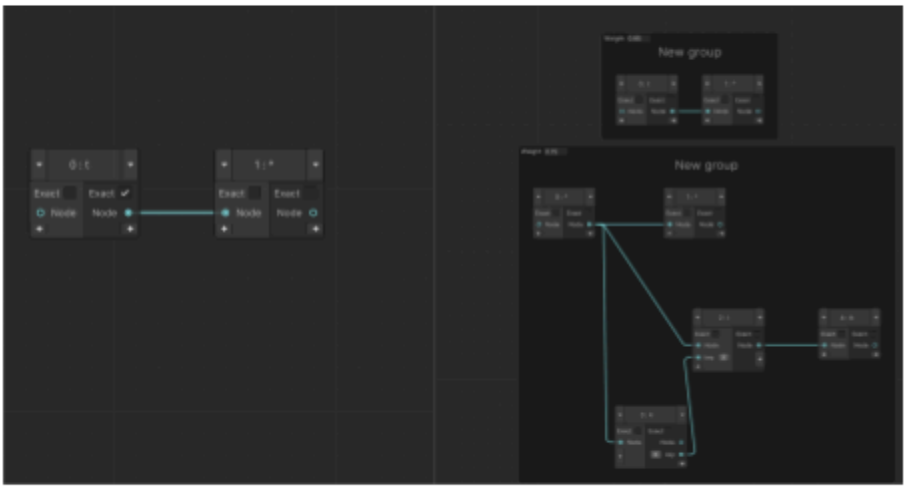


Removes superfluous key-lock relationships

On top of these rules, i have added two more



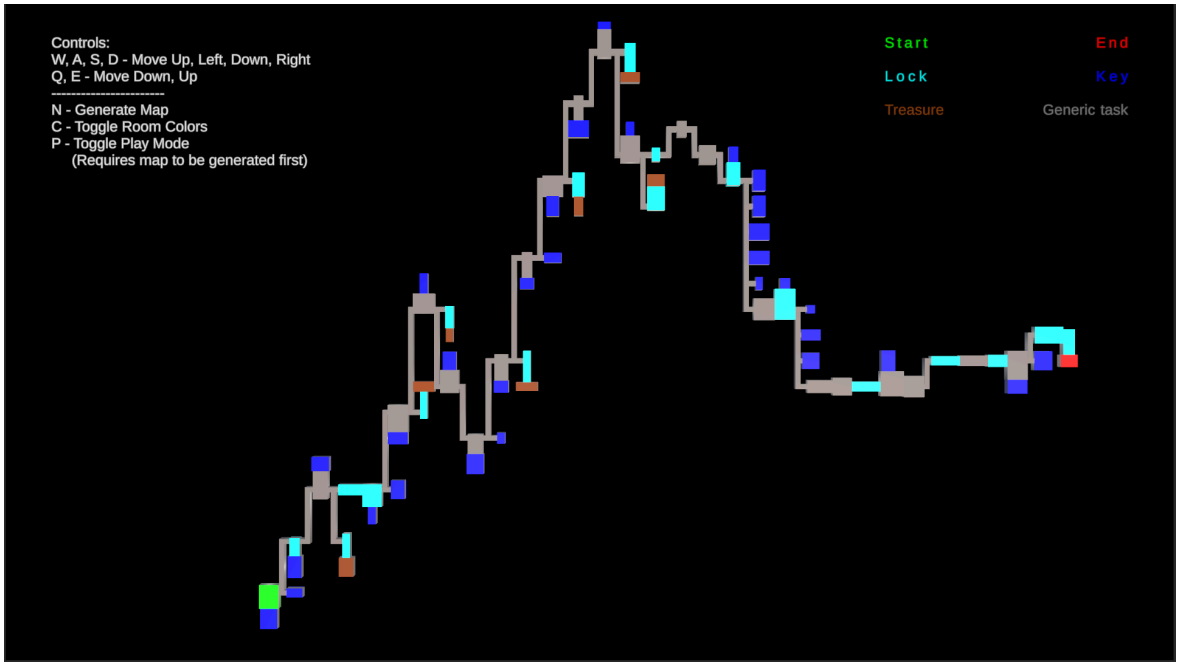
Remove Start Lock: Eventually a lock might end up being in a key relationship with the entrance (start) node. We know that because of the first rule the entrance always has a path to a key state, so we can use that as a key relationship.



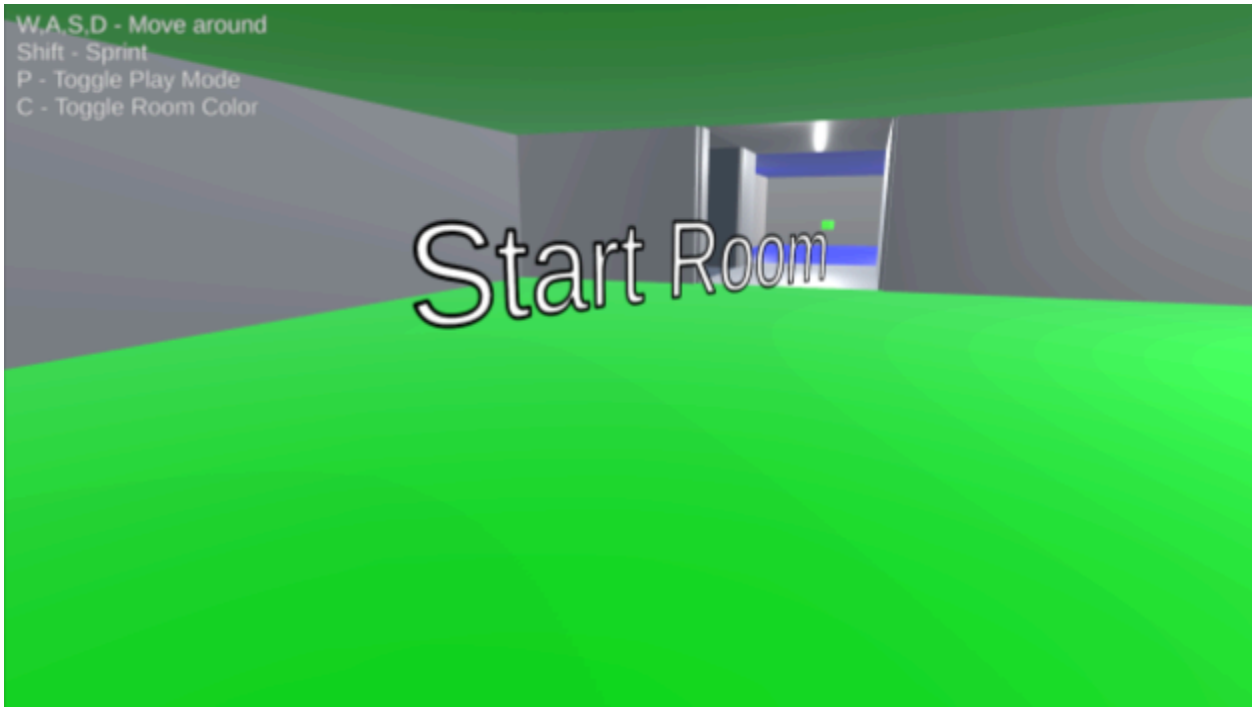
Add treasure: Whenever we find a generic task that points at anything, we have a 15% chance of adding two extra side room to that state which act as a key and lock for a treasure room.

The dungeon

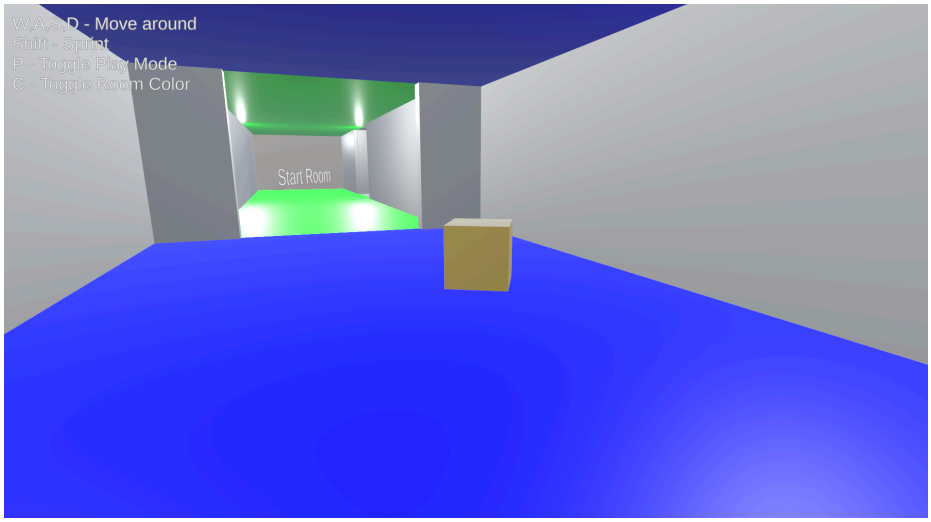
In the sample scene of the project a dungeon generator is ready to be used. This dungeon is generated using the production rules described in the previous section, and saved in the file called *“PCGBookGraph.asset”* in the “*Asset”* folder.



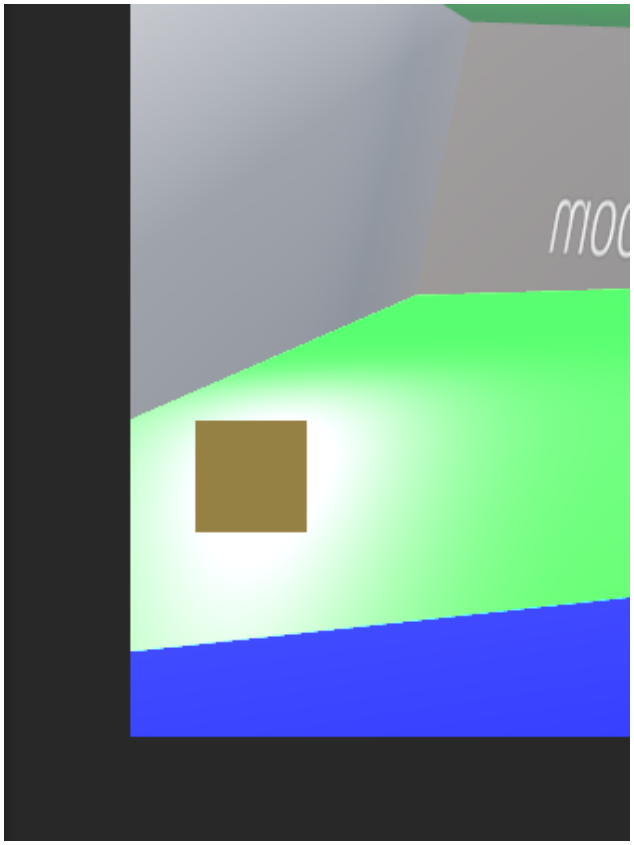
The dungeon can be viewed from above or played in first person.



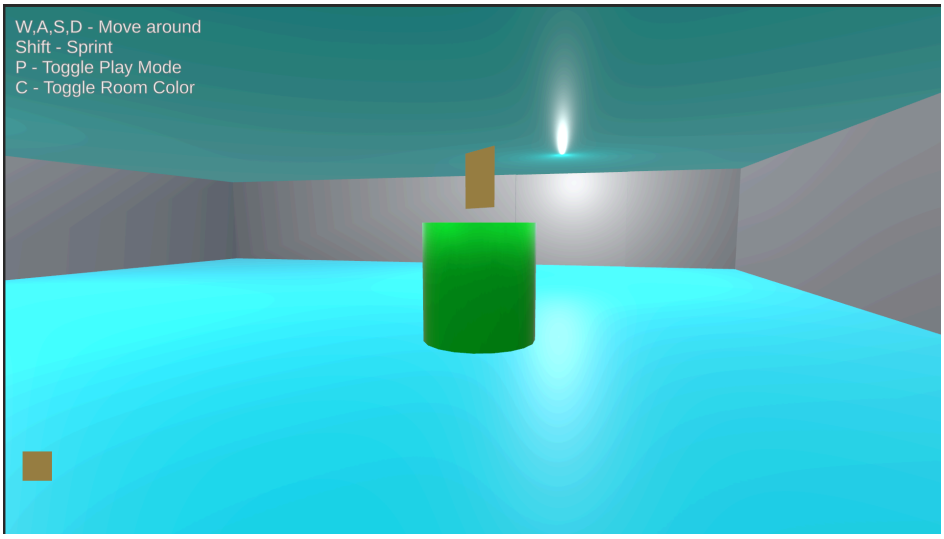
The keys are represented as cubes that can be grabbed simply by walking into them



Collected keys can be seen on the lower left side of the screen

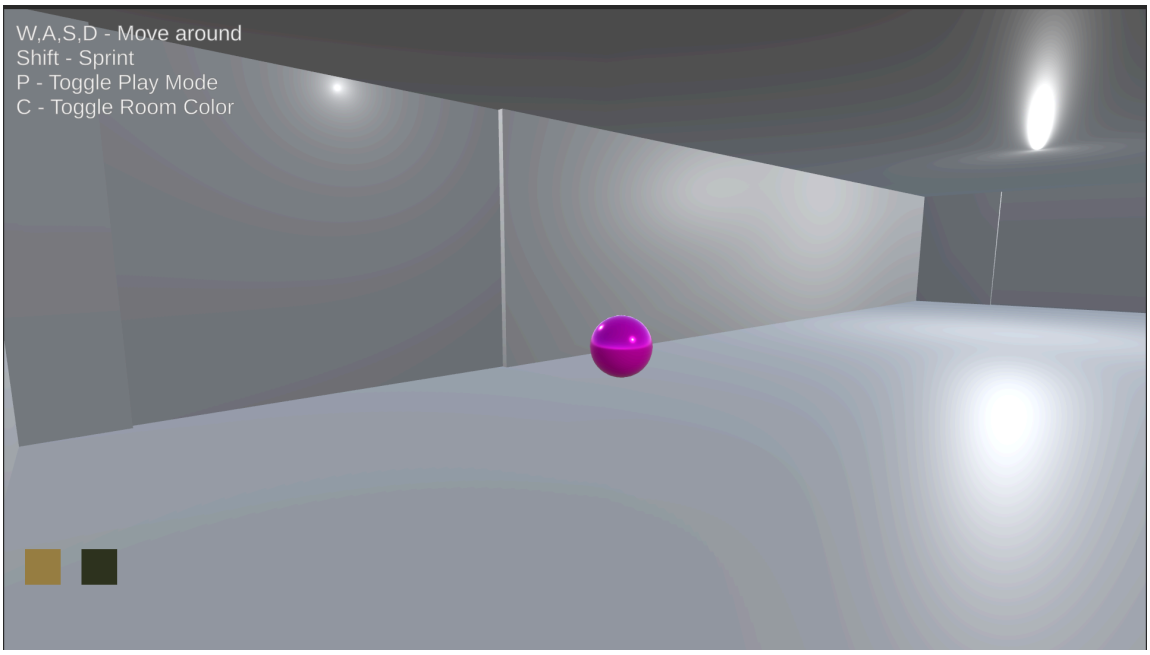


Locks are represented as cylinders, the keys needed are shown above their model



Running into them while having the right key/s will open all the closed doors present in the room.

Generic tasks are represented as purple spheres, running into them will open all the doors of that room, no keys are required for generic tasks.



Treasure room, entrance and exit are marked by floating texts with the corresponding name.

About the dungeon generator

The PCGGenerator is responsible for creating the actual 3D models of the dungeon and is tailored to work with the graph grammar from the book. Therefore, given a graph grammar with different rules may lead to unexpected results.
The assumptions that the generator make is that the symbols used for the grammar are as follows:

- **Non-Terminal**
  - S,T
- **Terminal**
  - start, goal, t, l, k, b
- **Edge**
  - key

The graph grammar for the book also generates graphs that are spanning trees, therefore the rooms and corridors are placed knowing that no loops are possible and that the only direction that the player can take is from the start to the goal.

In addition other small tricks have been used in order to make the dungeon feel less repetitive, such as squashing small corridors, randomly assigning room sizes and changing room position to avoid forming straight lines for too long, depending on your rules this might cause some issues such as overlapping corridors.

Final Remarks

This document briefly explained what graph grammars are and how to build one using the tool I made for Unity. It also shows how a graph grammar can be used to build a random dungeon generator.
Graph grammars, however, are not limited to small space generation such as the one shown in this document. They can also be used for generating terrain on a larger scale or even generate quests or missions.

Issues and other info

Meshes have been generated by manually inputting vertices instead of using prefabs such as cubes or planes because it allowed for an easier way (at the time) to compute intersection points used to add doors or corridors at the correct location. However this does cause some lighting issues and/or z-fighting.

Sometimes a room containing a key is inaccessible before its corresponding lock, which means that the player can no longer proceed. This is NOT due to a graph grammar issue but a mistake

that was made when assigning rooms to their tiles, it SHOULD be fixed but due to the nature of grammars it's hard to tell if it is actually fixed or all the rooms tested were "lucky".

When loading a graph grammar from disk the "Symbol List" is sometimes shown as empty. The production rules will still work but the Symbols must be readded manually in order to add new ones or modify old ones.
This is somewhat rare and usually happens when manually editing the graph grammar save files.

When playing in first person, the color of the keys are generated randomly due to their number, therefore some keys may have very similar color. This is not an issue per se, but if for some reason you are skipping keys make sure that the key in the inventory actually has the same color as the one above the lock.