# Overview

For my project, I decided to create a procedurally generated dungeon using L-Systems. My project begins with an axiom (SXE) and will iterate on that axiom by applying a set of rules to it. My system will probabilistically replace characters in the axiom using the given rules every iteration until an entirely new string is generated, with some rules holding more sway than others.

Once the final string is complete, there are a set of drawing rules that decide how each character in the string is interpreted. As the string is built, my program will look at these rules and the current string to create a set of vertices that will represent the path the dungeon will follow. Importantly, my program will also go through a series of checks to make sure that there are no overlaps in the dungeon path.

Once the path is created, I create a list of connections represented by a dictionary structure, where every key is a unique vertex in the path and every value is a list of vertices that connect to the key’s vertex. Once this list of connections is made, I spawn a room at each unique vertex based on the number and type of connections held by that vertex.

## Iteration Rules

X → F

X → FF

X → F[>F]F

X → F[<F]F

X → \*U\*F

X → \*D\*F

F → FX

F → >FX

F → <FX

F → [>F]F

F → [<F]F

S → <S

S → <<S

S → <<<S

S → S

E → <E

E → <<E

E → <<<E

E → E

Drawing Rules

X = Terminating character

F = Straight line

< = 90-degree rotation counterclockwise

> = 90-degree rotation clockwise

[ = Start a branch

] = End a branch

U = Angle up

D = Angle down

S = Start

E = End

\* = Straight line

# System Implementation

As mentioned, I made use of L-Systems to generate my procedural dungeon. To help with this, I expanded upon code given in class.

Prior to my changes, the code was mostly geared towards plant generation, and the only visual aspect the implementation was a way to see the path created after all iterations were ran. Additionally, the code only worked to generate a path in the x and y directions

For my project, I expanded on the provided code to include overlap detection, pathing in all three dimensions, and a visual display of rooms in the dungeon. I also adjusted/expanded the provided iteration and drawing rules to create a path that was more similar to a dungeon.

# Design

The algorithms I set out to implement have all been thoroughly tested and appear to work as designed. There were many edge cases I had to deal with, particularly in regard to the overlap detection, so it is possible there is something I missed somewhere. Otherwise, the major algorithms related to the L-System are as follows.

## Algorithms

### Selecting Rules

I first pull out a single byte from the current iteration of the L-System string and figure out if it has any rules associated with it. If it does not, I immediately try to apply the rule and add vertices to the dungeon path. Otherwise, I will select a single rule associated with that byte at random and try to apply my drawing rules to add vertices to the dungeon’s path. If the vertices can be added without causing an overlap, then they are added to the list of vertices. This list of vertices is cleared every iteration so that a new path can be created for each iteration of the L-System string.

In this part of the algorithm, I also make sure an exit can spawn, as well as handle the case where a valid rule cannot be found. If this is the case, I simply replace the rule with my terminating character.

### Adding Vertices

As vertices are added to my dungeon, I keep track of the current state of the path. This includes position, angle, and branch status of the most recent vertex in the path. For this algorithm, I copy all of this data into temporary variables in order to have a “correct” state prepared in case the vertices I’m trying to add cannot be added.

I start with a byte string representing the potential rule to be added to my L-System string. I then move through each byte of this string and apply my drawing rules, adding vertices to my temporary path and updating the temporary current state as needed.

Once I have a set of potential vertices from my drawing rules, I check to see if adding them to my dungeon path will result in any overlaps or invalid room placements. If so, the only thing I’ve changed is my temporary state, while the actual state of the dungeon path remains unchanged. If the vertices can be added to the path, I update the current state of my dungeon path using the temporary state and add the vertices to my path.

## Class Diagram

Graphical user interface, text, application

Description automatically generated with medium confidence

# Clever Implementations

I was particularly proud of the way I generated and displayed the rooms of the dungeon, even if it may not be the most optimal solution. I first created a dictionary of unique vertices and their connected vertices. Once this list was built, I used the number, direction, and type of each connection from each vertex and created a unique room string.

Specifically, I started with an array of four bytes, with each index representing a connection in a different direction. If the value of a byte in the array equals 1, there is a connection in that direction. If it equals 0, there is no connection in that direction. This array then becomes the start of my unique room string representing the number and direction of connections from the current vertex (or room) in the dungeon. For example, a string of 1011 represents a connection forward, right, and left (The first value is a potential connection forward from the current vertex, then to the right, and then back, and then left).

The next part of the room string represents the type of room. This could be the starting room, the ending room, or an upward/downward path moving forward, left, back, or right.

The final part of the string further elaborates on the room type, being either a hallway or more of a room.

Finally, I created a series of room prefabs that represented the possible values of this room string. Once the room string is complete, I loop through all of the available rooms to find the correct room for the current vertex.

This implementation, although perhaps not the most optimal solution, was one that was relatively intuitive for me and allowed me to have a pretty large degree of control over the types and styles of rooms I wanted in my dungeon. It also freed me from needing to worry about rotating my rooms on the fly in order to properly represent the correct orientation of connections to and from a particular room.

Example room strings include “0110 (1)” (A hallway style room with connections to the right and back), “1001 DownRight” (A downward path moving from the right towards the forward direction), and “1000 Start” (A starting room that exits in the forward direction).

# Difficulties

The most difficult part of this project for me was by far the overlap detection.

In the end, I needed to find a way to temporarily implement my drawing rules so I could determine if they would result in any overlaps or invalid room placements. Once I figured this out, I started by trying to handle overlaps in just the x-z plane. When I figured out the proper checks to make sure overlaps didn’t happen here, I moved on to adding paths that moved up and down. These paths came with a whole host of other issues and edge cases, such as diagonal paths that crossed each other, and straight branching paths that came from the same vertex as an upward/downward path.

One particularly annoying edge case had me needing to create 16 additional room prefabs to handle a relatively rare bug where an upward or downward path would have nowhere to go besides to the left or right. Prior to discovering this bug, I wanted to avoid creating too many prefabs for upward and downward paths because it would have meant creating a lot of different combinations of branching paths. As such, I tried to create rules for these rooms that meant that there could only be a straight line before and after an upward or downward path. This way, I wouldn’t have to worry about creating and checking for multiple paths leading out of a single upward or downward slope.

However, if in the process of iterating through my L-System string one of these straight paths suddenly became invalid, my program would simply remove that byte entirely from the string because there was nowhere for it to go. This then resulted in a situation where my dungeon was looking to place a room type that didn’t exist, hence why I ended up needing to add additional room types. Fortunately, my room selection code was robust enough that only needed to add the additional prefabs; I didn’t need to adjust the code at all.

# Reflection

If I had to do the project over again, I don’t know that I’d necessarily change anything with my implementation. There are probably some things here and there that I could rework or simplify, but for the most part, I’m pretty happy with where it ended up and can say I’m relatively proud of the end result.

That being said, I wasn’t entirely happy with my rule sets. Ideally, I would like the number of iterations to be pretty closely related to the size of the dungeon, with more iterations creating larger dungeons. For the most part, this is true of my project as is, but it is still relatively inconsistent. For example, I currently have the number of iterations set at 6. This usually makes some pretty interesting and sprawling dungeons, but there are occasionally long stretches of straight paths. Additionally, the dungeon is also occasionally a bit smaller and simpler than I would like.

If I had a bit more time, I’d probably want to make my rooms nicer, most likely with either assets from the Unity asset store or with assets I create myself. I also wouldn’t mind adding gameplay elements, but at the moment I’m unsure what exact form those would take.

Finally, I also wouldn’t mind adding an AI NPC that finds their way through dungeon, with the dungeon rebuilding in a new configuration every time they reached the end.