Cozy Caves

A D&D Dungeon Generator

Room Generation Module

Written By
Abdulrahman Asfari

Reviewed By
Gavin Lim
Gideon Wilkins

Technical Outline - Rooms

1. Brief
Our project methodology places a large focus on creating small robust systems. In order for this module to align with that goal, a high level of encapsulation should be implemented, so that the most relevant parts of the module are exposed to other modules. With JavaScript I cannot completely hide other aspects of the module, however using the builder pattern, I define a clear interface for other modules to interact with.

Encapsulation is vital to creating abstraction in this module, as well as preventing control over parts of an object that should not be changed outside of the module. Encapsulation will largely take form using the JS private access modifier. As there is no support for the protected modifier, abstracted objects should generally avoid using inheritance unless necessary.

2. Module Inputs
When the dungeon generation module attempts to access this module, there should be a high level of customizability when requesting a room in order to meet the generation algorithm's exact needs. Because of this requirement, there may be additional parameters added to the room generation inputs as development progresses, and for this reason I will be using the builder pattern. This not only makes using the module more maintainable due to not having to rewrite the constructor call, but also allows the caller to omit options they are not concerned with.

Builder Seed - The same seed given should always generate the exact same room.
Reset on Build - Resets builder parameters when a room is built.
Size - Expected room size.
Leniency - How far off from the room size the generated room can be.
Overgrowth Allowal - Whether leniency can generate a room larger than size.
Non-Rect Allowal - Whether a non-rectangular shape is allowed.
Item Population Allowal - Whether a room should be populated with items.
Tileset - A bundle with sprites to render and wall sprite decision logic.
Layout Blacklist - Layouts to not consider when generating a room.

3. Data Structures
A more traditional approach would represent tiles in a 2D array. My data structure for rooms will instead use a custom data structure. The goal is for it to provide any functionality a 2D array would, as well as linear iteration and additional utilities that can be implemented as needs arise.

Room
This will be the object returned to the caller of this module. It should contain any information other modules may need, and will likely be an object organising the other data structures in an accessible manner, with utility methods to manipulate the data. At its core, it should have a collection of tiles and a position value.

Tile

A basic data structure containing:

- Tile Type
- Tile Source
- Position
- Offset
- Rotation
- Depth

Tile Source

Contains image source and dimensions.

Tileset

Each tileset maps each tile type to a tile source getter callback. In the future it may also contain metadata about the tileset that will affect room generation logic such as whether the room walls are inset or not, or if blending floors are used.

4. Generation Pipeline

Inputs - Process inputs from the room builder.
Choose Shape - Decide on layout from a pool of layout.
Generate Room - Scale/build the room using tiles. If the layout is invalid, try another one.
Populate Room - Send room to prop & item module, if the option is selected.

5. Hallways

Hallways will not be considered for this first iteration of room generation. This is because generating hallways will require radically different parameters that are highly dependent on how the dungeon generation module decides to approach hallways. It is likely that it will be its own module that uses some utility classes from this module.