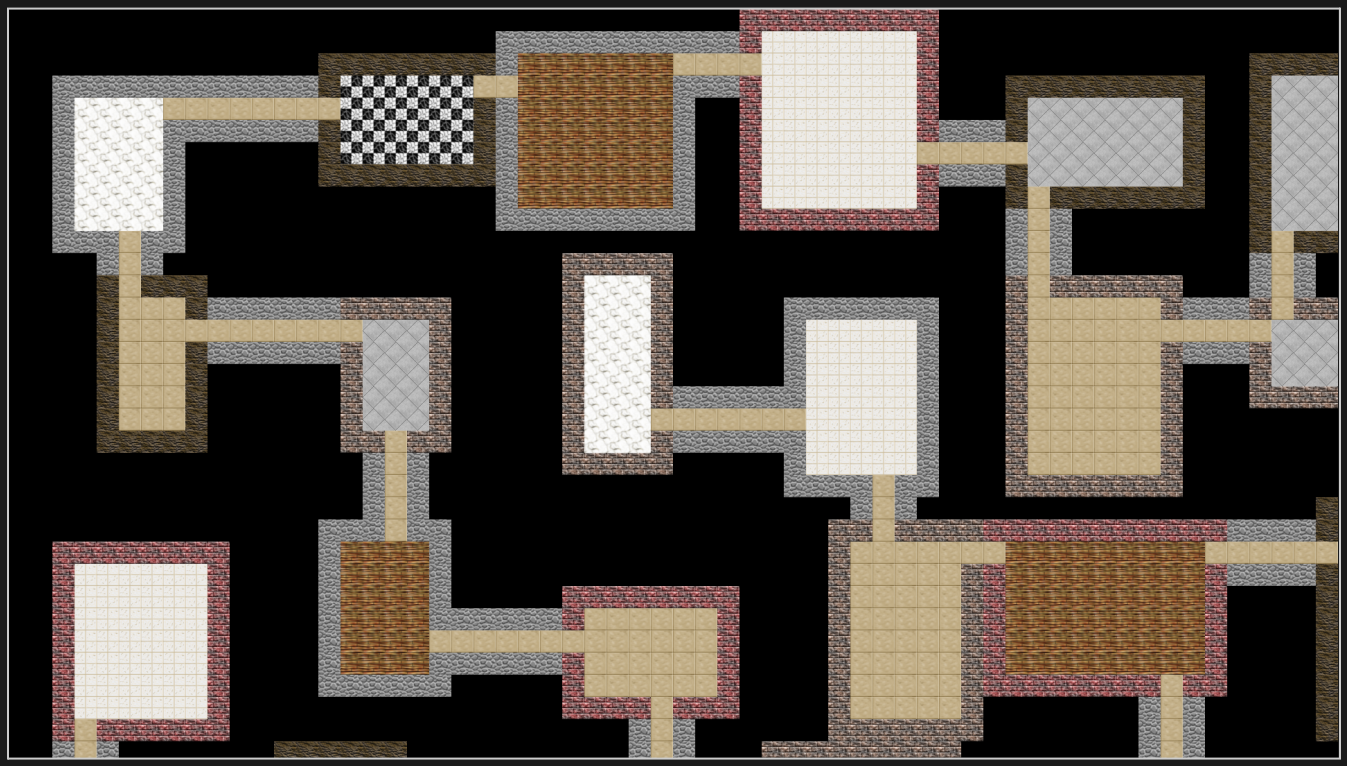
Dungeon_gen Technical Notes

This document will cover how the Dungeon_gen code works, why this method was chosen over others, and several other technical topics.



What does Dungeon_gen do?

Dungeon_gen can generate a dungeon with over a million interconnected rooms in approximately one second using one core/thread on a three gigahertz system. All of the rooms in the dungeon generated are connected together using a modified depth-first search algorithm.

Theory and Process

If you were to start placing rooms at random into a limited area you would eventually have rooms colliding with one another. You would then have two choices, you could either 1) try to move the room you just created or 2) try placing it again. This method becomes increasingly inefficient as more rooms are placed since each room will have to be checked against all of the other rooms that have already been placed. So when you place your first (1st) room you will have zero (0) other rooms to check against, when you place your second (2nd) room you will have one (1) other room to check against, when you place your third (3rd) room you will have two (2) other rooms to check against, and so on so that when you place your one thousandth (1000th) room you will have nine-hundred and ninety-nine (999) other rooms to check it against!

Therefore, such a method would be too inefficient for the intended use of this project.

What about binary space partitioning (BSP) trees?

BSP trees have a number of issues. The tree structure, even if implemented in an array, is prone to cache misses. If you implement it as an actual tree using pointers then traversal will slow you down.

Even if you make the most efficient BSP tree implementation (putting the tree structure in an array and using a loop—rather than function calls—and an array based stack) you will still have cache misses because the tree is not built in an end to end order in the array.

Now that you have placed the rooms you still have to figure out how to connect them. Since the rooms are not arranged in a straight line nor in equally-sized boxes it is not possible to use methods such as a recursive backtracking algorithm to connect them.

This method therefore is not optimal for usage in this project given the issues that would arise from it.

Lessons learned from previous projects

When I made a similar project seven years ago it was only able to generate forty-thousand rooms in a single second. That project worked by placing rooms offset within blocks from left to right and row by row.

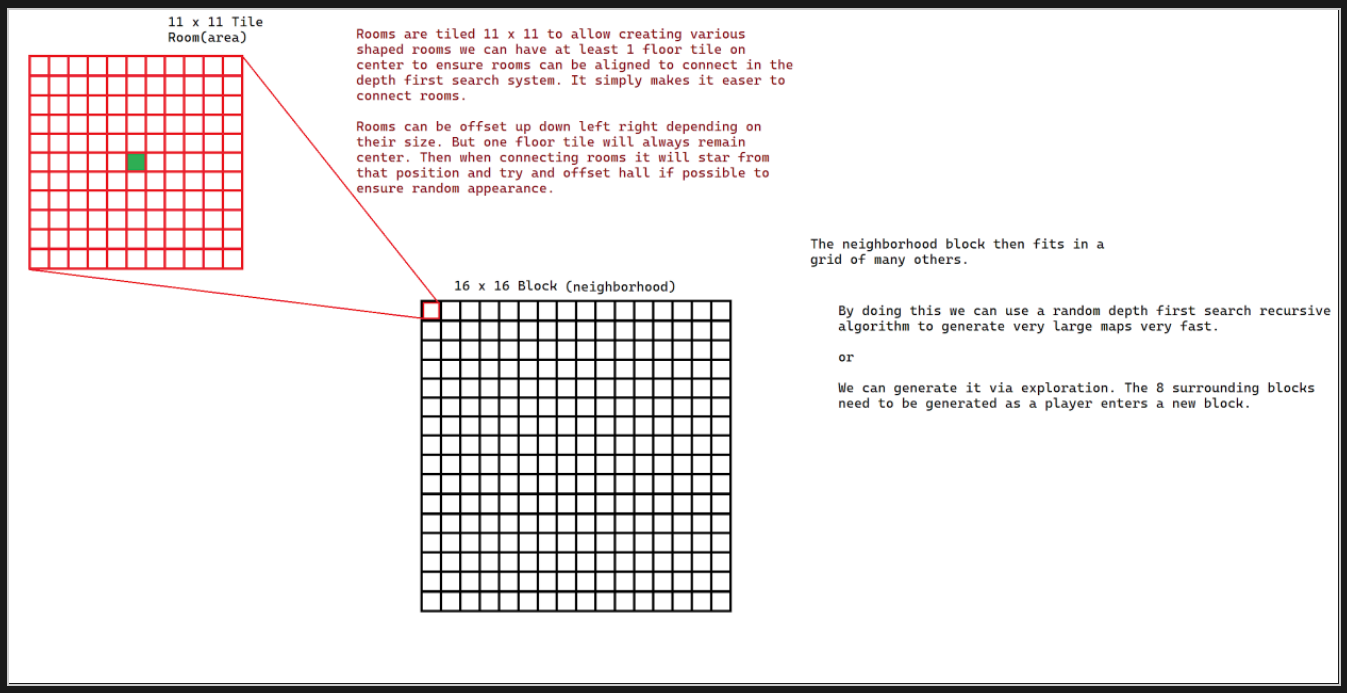
The recursive backtracking algorithm was able to work with some modifications, however there were rooms still disconnected at the end that had to be detected and connected which used up time. Performance was able to be improved by reducing the number of rooms which the path system was used on and then those were placed in blocks onto a larger map.

So what method is used?

Dungeon_gen takes into account all of the lessons learned from previous projects.

The rooms are generated in room.cpp:

- Rooms fit into an 11 x 11 tile area
    - This is done using a weighted position system
- Top[] contains the values to generate the top left room coordinates
- Bot[] contains the values to generate the bottom right room coordinate
- (tx,ty),(bx,by) are randomly selected from Top[] and Bot[]
- If the distance between bx and tx is less than 4 tiles it randomly decides which dimensions to increase while ensuring tile position (5,5) is inside the room
- Then the neighbors are determined for the purpose of creating the block-level depth-first search algorithm.



Blocks

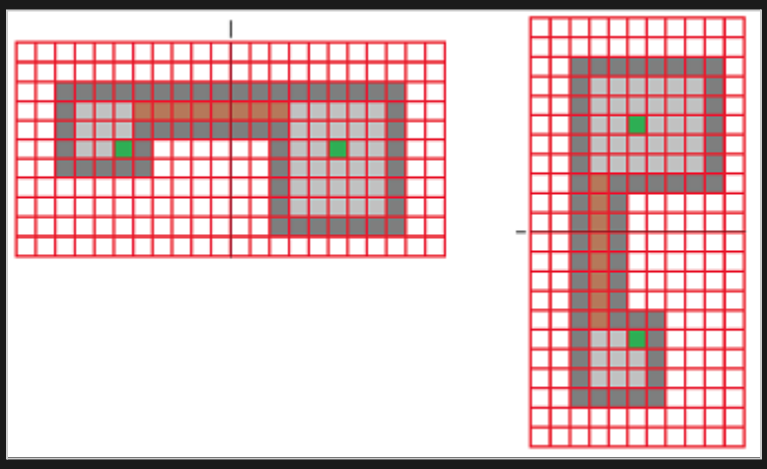
block.cpp creates the block-level sections of the map. This is a 16 by 16 room (or 176 x 176 tile) block.

init() allocates the space for tiles.

gen_rooms() generates the rooms and copies them into the tile location as they are created.

check_neighbors() returns what neighbors are **not visited** using N_stack.

make_halls() is responsible for filling the tiles of the halls once the rooms are selected to connect. This is done using that (5,5) block in each room as a start point. That room is guaranteed to be a floor tile in every room. This means you could just connect all the rooms from that position. Instead the halls are offset in some direction in most cases. If the hall is horizontal then the offset is up or down and if the hall is vertical then the offset is left or right.



The first thing to do is determine if the hall is vertical or horizontal. Once that is done we enforce the order of left to right and top to bottom so that we only need one vertical and one horizontal manner of filling hall tiles.

We calculate the start position which is the wall closest to the other room. Then fill the space between the two.

We draw on each side of it the wall.

Nota bene:

```
if(tiles[roomoffset+((ky-1)*176)+startx+x]==0){tiles[roomoffset+((ky-1)*176)+startx+x]=1;}
tiles[roomoffset+(ky*176)+startx+x]=35;
if(tiles[roomoffset+((ky+1)*176)+startx+x]==0){tiles[roomoffset+((ky+1)*176)+startx+x]=1;}
```

One anonymous user claimed that this code was not efficient due to the fact that the math is repeated. However, the repetition is insignificant to the code's performance. The average run-time remained 1.191 seconds both with and without the repetition meaning that the difference was not even 0.001 seconds.

```
sss = roomoffset+startx+x;
wall_1=((ky-1)*176)+sss;
wall_2=((ky+1)*176)+sss;
if(tiles[wall_1]==0){tiles[wall_1]=1;}
tiles[(ky*176)+sss]=35;
if(tiles[wall_2]==0){tiles[wall_2]=1;}
```

(Without Repetition)

Gen_halls() is where the recursive backtrack function is run that selects the rooms to connect then passes that to the **make_halls()**.

Block::Build() is called by **dungeon::gen_blocks()** to build the block.

Dungeon Generator

The dungeon generator uses blocks similar to how blocks uses rooms.
dungeon::gen_blocks() does effectively what **dungeon::gen_blocks()** did with rooms but with blocks;

dungeon::make_halls() is very similar to **block::make_halls()**

The primary difference is **block::make_halls()** only has the (5,5) position for each room. A block has 16 rooms on each side and their 5,5 position for each room falls on {5,16,27,38,49,60,71,82,93,104,115,126,137,148,159,170}. So it allows connecting to a random room.

Dungeon::gen_halls() uses the same depth-first search recursive backtrack algorithm as **block::gen_halls()** does.

All the print stuff is from when I first tested it printing it out before I dropped it into a graphical system.

I left some stuff in that I commented out to give a sense of some changes I made with this. Such as in rooms the position selection. That was because the room size was too random and you ended up with far too many small rooms.

Performance

The reason this method is fast is because of a number of factors. I divided the work. This is a common method used in sorting functions. Secondly array access is for the most part very linear. Maybe 1 cache miss for every 176 or so. That is caused by swapping to another row in the array. The recursive back track system will get far slowed the larger it gets. We eliminate data touches such as comparing room positions when placing them entirely. We eliminate trying to find connection points between rooms and blocks. We have guaranteed starting points to work with.

When it comes to speed there are several key factors

How well you fit in the cache size.

How well you avoid cache misses.

How often you need to touch data or access it.

Examples

> Those room's comparisons each time you have to compare a room against another that adds up.
>
> The recursive backtracking would be much worse the larger it gets. If we did not use the blocks and just tried to connect all million rooms that way it would take far longer
>
> Trying to find connection points

Avoiding excessive abstraction, every function call adds up. There is a limit to what a compiler can fix.

There is a limit to what compilers can do Optimization wise. It can not turn your crappy code into great code at best it can turn it into slightly less crappy code. 99.9% of optimization can not be done by a compiler.

You can see that in sorting algorithms and this here.
This program without all the optimizations built in by me that with -O2 turned on would take years to complete. I ran a test on smaller versions while I was building this over the years.

If you look at my youtube channel you can see some of the earlier results from 7 years ago.
https://www.youtube.com/@GRHmedia/videos

You will notice one similar map at 7000 rooms in 7 seconds and another video I talk about the collision rate once you hit 30% of the area filled you will get a thousand collisions before you get a placement and so on.

So it is impossible for the compiler to make up for that. If you don't plan your project out to that extent, trying to rewrite it and add all the optimization in isn't going to work out as well. In fact I just scratched most of those projects and started a new one each time. I can safely say the current system has ZERO of my original code in it. It has at most the art and the basic game shell from the last version.

How much does the compiler improve it? 3.186 seconds down to 1.191 seconds average.
However, that is nothing compared to the time I reduced the workload. A mere 2 seconds compared days, weeks, months and years of time.

Of course this discussion doesn't go into all the various methods I used like space partitioning which was used for the one below.

Premature optimization:

1,048,576 rooms are generated in a grid of 64 by 64 blocks each block having 256 rooms. All rooms in the map are connected in some way to one another. The space that contains these rooms is 11264x11264 tiles in size. Each room is between 5x5 and 11x11 in size. On average the rooms are larger due to weighting. In short three quarters of the block space available is filled with rooms.

If you were to drop rooms in at random into an area once you reach 30% of the area being filled you will have more collisions than not. It will take a very long time to place rooms in that manner. If you elect to move the rooms you will need to compare them to the other rooms. You could use a relaxing function such as can be done with Varoni diagrams or other methods. They are computationally expensive but won't take you as much time as dropping the rooms in.

Those methods are fine for doing a few rooms to even hundreds of rooms. The relaxation function can be fairly fast. But you are then left with rooms that are far less easy to connect.

Even with -O2 enabled the base of this project, just dropping rooms in, would have taken years if it ever completed. It is only with the optimization I created to modify the workflow that it was possible to reduce the time to such a great extent.

Each major change effectively forced a rewrite of the entire method used. Many projects are incompatible with making modifications to simply improve performance. Take a sorting function like bubble sort; it is slow in comparison to quicksort or radix sort. If you want to improve it you can not simply make some small changes to get the performance you will with the other sorts. You are forced to rewrite it entirely.

Premature optimization is poorly named. It would be better to say ineffective optimization. You can certainly spend time on ineffectively optimizing code. You can not be premature when it comes to optimization. You have to think about optimization from the start to get the best optimization.

I hear a lot of programmers spout the term premature optimization using it as an excuse to wait until the end to consider how optimization will impact their projects and thinking they can simply run a profiler and fix bottlenecks. They are fools.