

# Procedural Dungeon Generation - JTippets

## Generic Maze Generating Algorithm

My previous rant got me thinking a little bit. I may have jumped the gun a bit. The original Accidental article dealt with generating mazes within the context of the tile-map grid, rather than at a higher level of abstraction that would make it easier to deal with the maze as a graph of connected nodes. Therefore, I began work on a more abstract representation of a maze, and I find that it can be pretty useful for things like this.

The representation I chose was to store information regarding the connection of edges between cell vertices, as well as to be able to reference or store information regarding the edges of a given cell. So I required operations to be performed on either cells or nodes(vertices). Get/Set/Clear operations can be done on an edge basis (ie, connect an edge from this corner in the north direction) or on a cell basis (ie, set the north edge of this cell). Such a representation facilitates multiple methods of maze generation, and allows easy analysis of each cell to determine its edges and select a pre-made room pattern corresponding to the edge pattern.

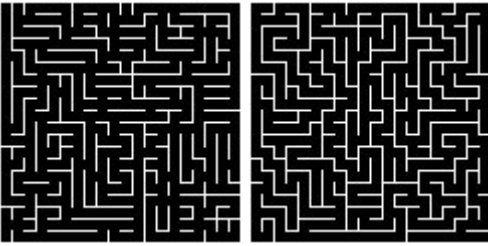
I also wanted to specify certain additional cell-based functionality useful for the depth-first random generation algorithm; specifically, the ability to flag a cell as visited.

I represent each node(vertex) as a single unsigned char, and store connectivity information in individual bits. Additionally, I represent each cell as a single uchar, and encode visited as a single bit. Has the advantage of compacting a maze into  $w*h + (w+1)*(h+1)$  bytes, where  $w,h$  are the dimensions of the maze in cells. So a 200x200 maze is represented in 80401 bytes. That's about as compact as I could make it without getting extremely weird with bit packing, and I will rarely if ever have occasion to use it for large mazes anyway since I am only building it as a basis for the level templating described in my previous post. And a templated level built on a maze of 200x200 cells would be huge, bewilderingly complex, and completely unnecessary.

I provide two methods of maze generation. The first is the wall-growth mechanism detailed in the original Accidental article. All node positions in the maze are compiled into a list then shuffled into random order. The list is then iterated, and at each location I check to see if the node is blocked, ie connected to any other node by an edge. A blocking condition halts wall building. If the node is not blocked, a random direction and wall length are determined, and the algorithm attempts to build a wall in that direction and up to that length. Again, blocked nodes halt wall building. The result is a maze comprised of many looping paths, wherein one can travel from one point to another by multiple paths.

The second generation method is a randomized recursive depth-first search of the maze. Starting at a random starting location, mark the cell as visited, then visit each of its 4 neighbors in turn, random order. For each neighbor cell that has not been visited, knock down the wall between the two cells, then recurse with that neighbor cell as the current cell. This algorithm generates a 'perfect' maze, wherein there is exactly 1 path between any two randomly selected points. Creates a maze with lots of long, branching, twisting passages and dead ends.

These 2 images show the 2 algorithms in action, first the wall-growth technique, then the depth-first technique.



The maze header:

```
#ifndef MAZE_H
#define MAZE_H
#include <vector>

struct CMazeNode
{
    unsigned char edges;

    void setEdge(unsigned char e){edges |= e;};
    void clearEdge(unsigned char e){edges &= ~e;};
    void clear(){edges=0;};
    void set(){edges=15;};
    bool getEdge(unsigned char e){return ((edges & e) != 0);};
    bool isBlocked(){return (edges != 0);};
};

struct SNodeListElement
{
    unsigned int x, y;

    SNodeListElement(unsigned int X, unsigned int Y) : x(X), y(Y) {};
};

enum EEdgeMasks
{
    E_North=0x01,
    E_West=0x02,
    E_East=0x04,
    E_South=0x08,
    E_Visited=0x10
};

class CMaze
{
public:
    CMaze(){nodes=0;cells=0;};
    ~CMaze(){destroy();};

    void init(unsigned int cw, unsigned int ch);
    void destroy();

    unsigned int getCellWidth(){return cell_width;};
    unsigned int getCellHeight(){return cell_height;};
    unsigned int getNodeWidth(){return node_width;};
    unsigned int getNodeHeight(){return node_height;};

    // Node-based functions
    void setEdgeNode(unsigned int x, unsigned int y, unsigned char cm_edge);
    void clearEdgeNode(unsigned int x, unsigned int y, unsigned char
cm_edge);
    bool isBlockedNode(unsigned int x, unsigned int y);
```

```

bool getEdgeNode(unsigned int x, unsigned int y, unsigned char cm_edge);

// Cell-based functions
void setEdgeCell(unsigned int x, unsigned int y, unsigned char cm_edge);
void clearEdgeCell(unsigned int x, unsigned int y, unsigned char
cm_edge);
bool getEdgeCell(unsigned x, unsigned y, unsigned char cm_edge);
unsigned char getEdgePattern(unsigned int x, unsigned int y);

// Spanning-tree based generation base functionality
// Used by algorithms that generate based on some sort of
// recursive or tree-based randomized search algorithm
void setVisited(unsigned int x, unsigned int y);
bool getVisited(unsigned int x, unsigned int y);
void clearVisited(unsigned int x, unsigned int y);
void setBackDirection(unsigned int x, unsigned int y, unsigned char
dir);
unsigned char getBackDirection(unsigned int x, unsigned int y);
void clearBackDirection(unsigned int x, unsigned int y);

// General functions
void setAllEdges();
void clearAllEdges();
void setBorderEdges();

void clearAllCells(); // Clear back and visited for all cells

void clear(){clearAllEdges(); clearAllCells(); setBorderEdges();};

// Generation functions
void generateWallGrowthMaze(unsigned int min_wall, unsigned int
max_wall);
void generateDepthFirstMaze();

private:
CMazeNode *nodes;
unsigned char *cells;
unsigned int cell_width, cell_height; // dimensions in cells
unsigned int node_width, node_height; // dims in nodes

void buildNodeList(std::vector<SNodeListElement> &list);
void shuffleNodeList(std::vector<SNodeListElement> &list);
unsigned char randomDir();
void buildMazeWall(unsigned int x, unsigned int y, unsigned char cm_dir,
unsigned int len);
void recurseDepthFirst(unsigned int x, unsigned int y);
};

#endif

```

### And the source:

```

#include "maze.h"
#include <vxnlbr/vxnlbr.h>

#include <vector>
#include <algorithm>

// Helper functions

void CMaze::buildNodeList(std::vector<SNodeListElement> &list)
{

```

```

// Populate a vector with (x,y) pairs for every node in the maze
if(!nodes) return; // Not initialized

unsigned int x,y;
list.clear();
list.reserve(node_width*node_height);

for(x=0; x<node_width; ++x)
{
    for(y=0; y<node_height; ++y)
    {
        list.push_back(SNodeListElement(x,y));
    }
}

shuffleNodeList(list);
return;
}

void CMaze::shuffleNodeList(std::vector<SNodeListElement> &list)
{
    // Use our global random generator for predictable level regeneration
    /*unsigned int size=list.size();
    if(size==0) return;

    unsigned int c;
    for(c=0; c<size; ++c)
    {
        unsigned int rn;
        do
        {
            rn=g_randRange(0,size-1);
        }
        while(rn==c);

        SNodeListElement tmp=list[c];
        list[c]=list[rn];
        list[rn]=tmp;
    }*/

    // algorithm defines a shuffle algorithm that operates on
    // iterators
    // std::random_shuffle(list.begin(), list.end());

    // Or, we can pass our global g_randTarget as a functor
    std::random_shuffle(list.begin(), list.end(), g_randTarget);
}

void CMaze::init(unsigned int cw, unsigned int ch)
{
    destroy();
    if(cw==0 || ch==0) return;

    cell_width=cw; cell_height=ch;
    node_width=cw+1; node_height=ch+1;

    nodes=new CMazeNode[node_width * node_height];
    cells=new unsigned char[cell_width*cell_height];
    clearAllEdges();
    clearAllCells();
}

```

```

void CMaze::destroy()
{
    if(nodes)
    {
        delete[] nodes;
        nodes=0;
    }
    if(cells)
    {
        delete[] cells;
        cells=0;
    }
}

// Node-based functions

// setEdgeNode-- Connect an edge starting at x,y and extending one edge length
in the
// direction given by cm_edge.
void CMaze::setEdgeNode(unsigned int x, unsigned int y, unsigned char cm_edge)
{
    if(x>=node_width || y>=node_height) return;
    if(!nodes) return;

    nodes[y*node_width+x].setEdge(cm_edge);

    // Now, set other node
    unsigned char e;
    unsigned int nx, ny;
    if(cm_edge==E_North)
    {
        e=E_South;
        nx=x;
        ny=y-1;
    }
    else if(cm_edge==E_South)
    {
        e=E_North;
        nx=x;
        ny=y+1;
    }
    else if(cm_edge==E_East)
    {
        e=E_West;
        nx=x+1;
        ny=y;
    }
    else if(cm_edge==E_West)
    {
        e=E_East;
        nx=x-1;
        ny=y;
    }
    else return;

    if(nx>=node_width || ny>=node_height) return;

    nodes[ny*node_width+nx].setEdge(e);
}

// clearEdgeNode -- Remove an edge starting at x,y in direction cm_edge

```

```

void CMaze::clearEdgeNode(unsigned int x, unsigned int y, unsigned char cm_edge)
{
    if(x>=node_width || y>=node_height) return;
    if(!nodes) return;

    nodes[y*node_width+x].clearEdge(cm_edge);

    // Now, clear other node
    unsigned char e;
    unsigned int nx, ny;
    if(cm_edge==E_North)
    {
        e=E_South;
        nx=x;
        ny=y-1;
    }
    else if(cm_edge==E_South)
    {
        e=E_North;
        nx=x;
        ny=y+1;
    }
    else if(cm_edge==E_East)
    {
        e=E_West;
        nx=x+1;
        ny=y;
    }
    else if(cm_edge==E_West)
    {
        e=E_East;
        nx=x-1;
        ny=y;
    }
    else return;

    if(nx>=node_width || ny>=node_height) return;

    nodes[ny*node_width+nx].clearEdge(e);
}

// isBlockedNode -- Query a node to see if any edges connect to it;
// this constitutes a 'blocking' condition for the wall-growth algorithm
bool CMaze::isBlockedNode(unsigned int x, unsigned int y)
{
    if(x>=node_width || y>=node_height) return false;
    if(!nodes) return false;

    return (nodes[y*node_width+x].isBlocked());
}

// getEdgeNode -- Determine if a node is connected to an edge extending in the
given direction
bool CMaze::getEdgeNode(unsigned int x, unsigned int y, unsigned char cm_edge)
{
    if(x>=node_width || y>=node_height) return false;
    if(!nodes) return false;

    return (nodes[y*node_width+x].getEdge(cm_edge));
}

```

```

// Cell-based functions

// setEdgeCell -- Set the edge of the given cell in the given direction
void CMaze::setEdgeCell(unsigned int x, unsigned int y, unsigned char cm_edge)
{
    if(x>=cell_width || y>=cell_height || !nodes) return;

    unsigned int nx, ny;
    unsigned char e;

    // Calculate one endpoint of edge
    if(cm_edge==E_North)
    {
        nx=x;
        ny=y;
        e=E_East;
    }
    else if(cm_edge==E_South)
    {
        nx=x;
        ny=y+1;
        e=E_East;
    }
    else if(cm_edge==E_East)
    {
        nx=x+1;
        ny=y;
        e=E_South;
    }
    else if(cm_edge==E_West)
    {
        nx=x;
        ny=y;
        e=E_South;
    }
    else return;

    setEdgeNode(nx, ny, e);
}

// clearEdgeCell -- Remove the given edge of the given cell
void CMaze::clearEdgeCell(unsigned int x, unsigned int y, unsigned char cm_edge)
{
    if(x>=cell_width || y>=cell_height || !nodes) return;

    unsigned int nx, ny;
    unsigned char e;

    // Calculate one endpoint of edge
    if(cm_edge==E_North)
    {
        nx=x;
        ny=y;
        e=E_East;
    }
    else if(cm_edge==E_South)
    {
        nx=x;
        ny=y+1;
        e=E_East;
    }
    else if(cm_edge==E_East)
    {

```

```

        nx=x+1;
        ny=y;
        e=E_South;
    }
    else if(cm_edge==E_West)
    {
        nx=x;
        ny=y;
        e=E_South;
    }
    else return;

    clearEdgeNode(nx, ny, e);
}

```

```

// getEdgeCell -- Query the given edge of the given cell
bool CMaze::getEdgeCell(unsigned int x, unsigned int y, unsigned char cm_edge)
{
    if(x>=cell_width || y>=cell_height || !nodes) return false;

    unsigned int nx, ny;
    unsigned char e;

    // Calculate one endpoint of edge
    if(cm_edge==E_North)
    {
        nx=x;
        ny=y;
        e=E_East;
    }
    else if(cm_edge==E_South)
    {
        nx=x;
        ny=y+1;
        e=E_East;
    }
    else if(cm_edge==E_East)
    {
        nx=x+1;
        ny=y;
        e=E_South;
    }
    else if(cm_edge==E_West)
    {
        nx=x;
        ny=y;
        e=E_South;
    }
    else return false;

    return getEdgeNode(nx, ny, e);
}

```

```

// getEdgePattern -- Compile an unsigned char value specifying the edge pattern
of
// the given cell. Classifies a cell based on it's open and closed edges into
one of
// sixteen categories.
unsigned char CMaze::getEdgePattern(unsigned int x, unsigned int y)
{
    unsigned char pattern=0;

```

```

    if(getEdgeCell(x,y,E_North)) pattern |= E_North;
    if(getEdgeCell(x,y,E_South)) pattern |= E_South;
    if(getEdgeCell(x,y,E_East)) pattern |= E_East;
    if(getEdgeCell(x,y,E_West)) pattern |= E_West;

    return pattern;
}

// General functions

// setAllEdges -- Connect all edges between nodes in the entire maze
void CMaze::setAllEdges()
{
    if(!nodes) return;

    unsigned int c;
    for(c=0; c<node_width*node_height; ++c)
    {
        nodes[c].set();
    }
}

// clearAllEdges -- Clear all edges between nodes in the entire maze
void CMaze::clearAllEdges()
{
    if(!nodes) return;

    unsigned int c;
    for(c=0; c<node_width*node_height; ++c)
    {
        nodes[c].clear();
    }
}

// setBorderEdges -- Set the edges on the outside border of the maze
void CMaze::setBorderEdges()
{
    if(!nodes) return;

    int c;

    // Set top and bottom edge
    for(c=0; c<cell_width; ++c)
    {
        setEdgeCell(c,0,E_North);
        setEdgeCell(c,cell_height-1,E_South);
    }

    // Set left and right edge
    for(c=0; c<cell_height; ++c)
    {
        setEdgeCell(0,c,E_West);
        setEdgeCell(cell_width-1,c,E_East);
    }
}

// setVisited -- Mark the given cell has 'visited'. Used by the depth-first
generation algo
void CMaze::setVisited(unsigned int x, unsigned int y)
{
    if(x>=cell_width || y>=cell_height || !cells) return;
}

```

```

    cells[y*cell_width+x] |= E_Visited;
}

// getVisited -- Check to see if the given cell has been visited already
bool CMaze::getVisited(unsigned int x, unsigned int y)
{
    if(x>=cell_width || y>=cell_height || !cells) return false;

    return (cells[y*cell_width+x] & E_Visited);
}

// clearVisited -- Clear the visited status of a cell
void CMaze::clearVisited(unsigned int x, unsigned int y)
{
    if(x>=cell_width || y>=cell_height || !cells) return;

    cells[y*cell_width+x] &= ~E_Visited;
}

void CMaze::setBackDirection(unsigned int x, unsigned int y, unsigned char dir)
{
    if(x>=cell_width || y>=cell_height || !cells) return;

    cells[y*cell_width+x] |= dir;
}

unsigned char CMaze::getBackDirection(unsigned int x, unsigned int y)
{
    if(x>=cell_width || y>=cell_height || !cells) return 0;

    return cells[y*cell_width+x] & 0x0F;
}

void CMaze::clearBackDirection(unsigned int x, unsigned int y)
{
    if(x>=cell_width || y>=cell_height || !cells) return;

    cells[y*cell_width+x] &= ~0x0F;
}

// clearAllCells -- Clear the visited and other status flags of all cells in the
// maze
void CMaze::clearAllCells()
{
    if(!cells) return;

    for(unsigned int c=0; c<cell_width*cell_height; ++c)
    {
        cells[c]=0;
    }
}

// buildMazeWall -- The workhorse of the wall-growth algorithm. Given a starting
// location, a
// direction, and a length, attempt to build a wall. Blocked nodes terminate
// wall building
void CMaze::buildMazeWall(unsigned int x, unsigned int y, unsigned char dir,
unsigned int len)
{
    unsigned int wx=x, wy=y;
    unsigned int ox,oy;

```

```

    if(isBlockedNode(x,y)) return;
    unsigned int c;
    for(c=0; c<len; ++c)
    {
        ox=wx;
        oy=wy;
        if(dir==1) wy-=1;
        if(dir==2) wx-=1;
        if(dir==4) wx+=1;
        if(dir==8) wy+=1;

        if(isBlockedNode(wx,wy))
        {
            setEdgeNode(ox,oy,dir);
            return;
        }
        else
        {
            setEdgeNode(ox,oy,dir);
        }
    }
}

// randomDir -- Select one of the for random directions
unsigned char CMaze::randomDir()
{
    unsigned int roll=g_randRange(1,4);

    switch(roll)
    {
        case 1: return E_North;
        case 2: return E_West;
        case 3: return E_East;
        case 4: return E_South;
    };
}

// generateWallGrowthMaze -- Construct a maze using the wall-growth method.
// The method works by accumulating all node coordinates into a list, shuffling
// the list, then iterating
// through it. At each location, attempt to draw a wall of randomly determined
// length and direction.
// Blocked nodes prevent wall building. This algorithm results in a maze that
// has lots of loops and
// multiple paths between any two given points. The passed parameters min_wall
// and max_wall
// determine the range of lengths of walls to be drawn. Walls may in reality
// turn out shorter
// due to hitting a wall. Longer wall lengths result in longer straight
// passages.
void CMaze::generateWallGrowthMaze(unsigned int min_wall, unsigned int max_wall)
{
    if(!nodes) return;

    std::vector<SNodeListElement> nlist;
    buildNodeList(nlist);

    unsigned int size=nlist.size();
    if(size==0) return;

    unsigned int c;

```

```

for(c=0; c<size; ++c)
{
    unsigned char d=randomDir();
    unsigned int len=g_randRange(min_wall, max_wall);
    unsigned int x=nlist[c].x;
    unsigned int y=nlist[c].y;
    buildMazeWall(x,y,d,len);
}
}

// recurseDepthFirst -- The workhorse of the depth-first tree-based generation
algorithm
// The function marks the current cell as visited, then shuffles the cell's 4
adjacent neighbors into
// random order and calls itself for each cell in the list that has not already
been visited. When
// a cell's neighbor is visited, the edge between the two cells is knocked down,
creating a passage.

void CMaze::recurseDepthFirst(unsigned int x, unsigned int y)
{
    // First, mark cell as visited
    setVisited(x,y);

    // Now, check each neighbor. If it hasn't been visited, knock down the
// wall and recurse on that cell
    unsigned int cx,cy;

    unsigned char dirs[4]={E_North, E_South, E_East, E_West};
    std::random_shuffle(dirs, dirs+4, g_randTarget);

    for(int c=0; c<4; ++c)
    {
        unsigned char d=dirs[c];
        switch(d)
        {
            case E_North: cx=x; cy=y-1; break;
            case E_South: cx=x; cy=y+1; break;
            case E_East: cx=x+1; cy=y; break;
            case E_West: cx=x-1; cy=y; break;
        }

        if(cx<cell_width && cy<cell_height)
        {
            if(!getVisited(cx,cy))
            {
                clearEdgeCell(x,y,d);
                recurseDepthFirst(cx,cy);
            }
        }
    }
}

// generateDepthFirstMaze -- Generates a maze using a randomized depth-first
search of the maze
// starting at a random location.
// This algorithm generates a 'perfect' maze, in which there is exactly 1 path
between any two randomly
// selected cells.
void CMaze::generateDepthFirstMaze()
{
    // First, select a random starting cell
    if(!nodes) return;

```

```

    unsigned int sx, sy;
    sx = g_randTarget(cell_width);
    sy = g_randTarget(cell_height);

    setAllEdges();
    recurseDepthFirst(sx, sy);
}

```

For a given (x,y) cell location, once the maze is generated you can call a function, **getEdgePattern**, that will return an unsigned char value representing the pattern of edges belonging to that cell. If the north edge is represented by the value of 1, for example, and the south edge is represented by the value of 8, then a cell with the north and south edges closed and the east and west edges open would have the edge pattern of 10. The 4 cardinal directions are represented as bits to facilitate this process. This allows the level generation algorithm to query the maze for a value, and use that value to determine which pre-generated room pattern to copy into the level grid for a given layout graph node. With 4 cardinal directions, this classifies each cell of a maze into 1 of 16 categories, so 16 categories of room templates must be used. (rotation can cut down the work required here, by allowing the generation script to rotate patterns in 90 degree increments before copying into the level grid).

If you attempt to use the above class, you will get some errors. In my shuffling functions, I call `std::random_shuffle` with a custom random number generator that is part of my standard library package (vxnlib ftw!). The generator wraps up a mersenne twister from `boost::random`, and provides calls for performing real distribution number generation in the range of 0,1 or integer distribution in the range of 0,N. You can either replace the generator with one of your own in the `random_shuffle` calls, or use my hackish random stuff here:

```

#include <cstdlib>
#include <ctime>
#include <boost/random/mersenne_twister.hpp>
#include <boost/random/uniform_int.hpp>
#include <boost/random/uniform_real.hpp>
#include <boost/random/variante_generator.hpp>

boost::mt19937 global_mersenne;
boost::uniform_real<> real_dist(0,1);
boost::uniform_int<> int_dist(0, RAND_MAX);
boost::variante_generator<boost::mt19937&, boost::uniform_real<> >
uni_real(global_mersenne, real_dist);
boost::variante_generator<boost::mt19937&, boost::uniform_int<> >
uni_int(global_mersenne, int_dist);

int g_randInt()
{
    // Return random int
    //return(rand());
    return uni_int();
}

void g_setSeed(int Seed)
{
    //srand(Seed);
    global_mersenne.seed(static_cast<unsigned int>(Seed));
}

void g_setSeedTime()
{
    //srand(time(0));
    global_mersenne.seed(static_cast<unsigned int>(std::time(0)));
}

```

```

int g_randTarget(int Target)
{
// Return rand up to, but not including, Target
    boost::uniform_int<> local_int_dist(0, Target-1);
    boost::variate_generator<boost::mt19937&, boost::uniform_int<> >
local_uni_int(global_mersenne, local_int_dist);
    return local_uni_int();
}

float g_rand01()
{
// Return random from 0 to 1 inclusive
    return uni_real();
}

int g_randRange(int Low, int High)
{ // Return random from Low to High inclusive
    if(Low==High) return Low;

    int Range=High-Low + 1;
    int Result;
    Result=g_randTarget(Range);
    return(Low + Result);
}

int g_diceRoll(int Dice, int Sides)
{// Emulate dicerolling
    int Roll;
    int c;

    Roll=0;
    for(c=0;c<Dice;c++)
        Roll+=g_randRange(1,Sides);

    return Roll;
}

```

I created a couple of random number generators in the global namespace to make it easier to swap existing rand-based code with the new generators. Might not be the best approach, but it works for me. So sue me. If you want, you can just remove the function object parameter to `random_shuffle`, and the algorithm will default to using `rand()`.

For visualization purposes, I merely iterate the edge graph, and draw horizontal or vertical lines in a bitmap depending on the edge connectivity of the nodes, then save the bitmap out to .tga format. It's a hacked up thingy, since it's not really part of the class and was only built for quick visualization purposes while testing and debugging. So I won't show it.

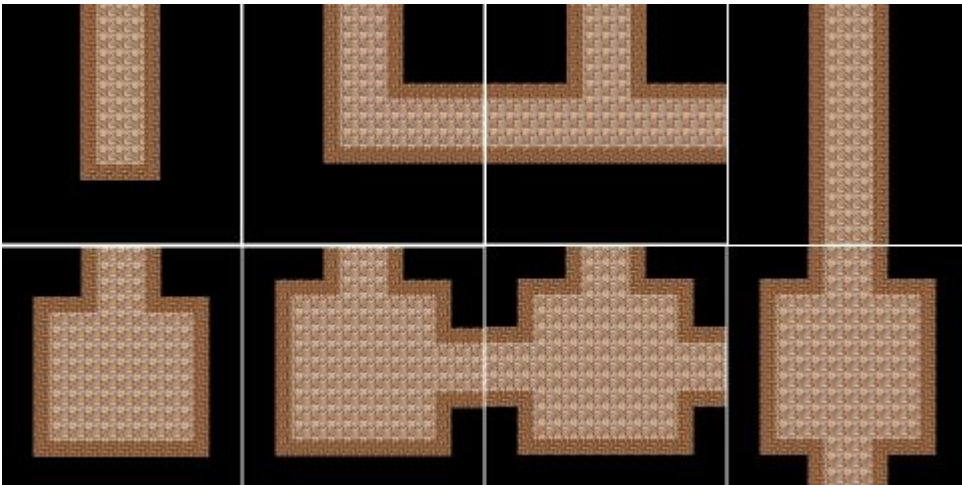
In days to come, I'll demonstrate how to use the generated maze as a level layout template, though the process here is simple enough.

Caveats: The class is a quick one, and my formal software engineering skills are nonexistent. There may be bugs, and there will certainly be better ways of doing things. I'm usually interested in hearing about them. But please try not to mire me in an endless recursion of finding better ways of doing things; I would rather move on to practical applications of things than sit around debating the merits of using `std::map` to represent a graph, and spending endless hours researching exactly how to do it. This method works well enough given the rigid constraints imposed by a graph living in the world of tile-based games.

## Adding Graphics

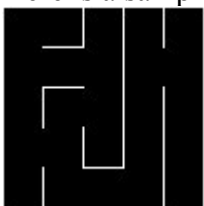
The maze class presented in the previous post provides a method called `getEdgePattern()` which queries a maze cell, analyzes its open and closed edges, and returns an unsigned char code representing that pattern. The pattern is determined by 4 bits. 1 bit for the north wall, 1 for the south, 1 for the east, and 1 for the west, for a total of up to 16 possible patterns. We can use that generated code for building our level. It is also useful to note that of these 16 patterns, several are similar to each other in shape, and differ only in orientation. Consider the piece where only the north wall is open and the others are closed. (Imagine a U shape). This piece is basically identical to the other three pieces where the south, east, and west walls are the only ones open, the only difference being the orientation of the piece. We can use this to decrease the work we have to do in converting a maze to a level, as we only have to pre-generated room patterns for 1 orientation and can then simply rotate them for the other similar configurations. Due to the nature of the mazes generated by the two algorithms presented, we can also safely ignore cell patterns of 15 (fully closed edges) since those cells are never generated. However, if you start tweaking with the maze (erasing walls, etc..) for special purposes, you may need those patterns.

Here is a sample set of room patterns, with two varieties for each category. In a real application, of course, you might want more variety. In the past, I have used as many as 12 general-case selections for each category, though that many might not be necessary. Even 4 or 5 varieties provides a great deal of randomization in room selection.

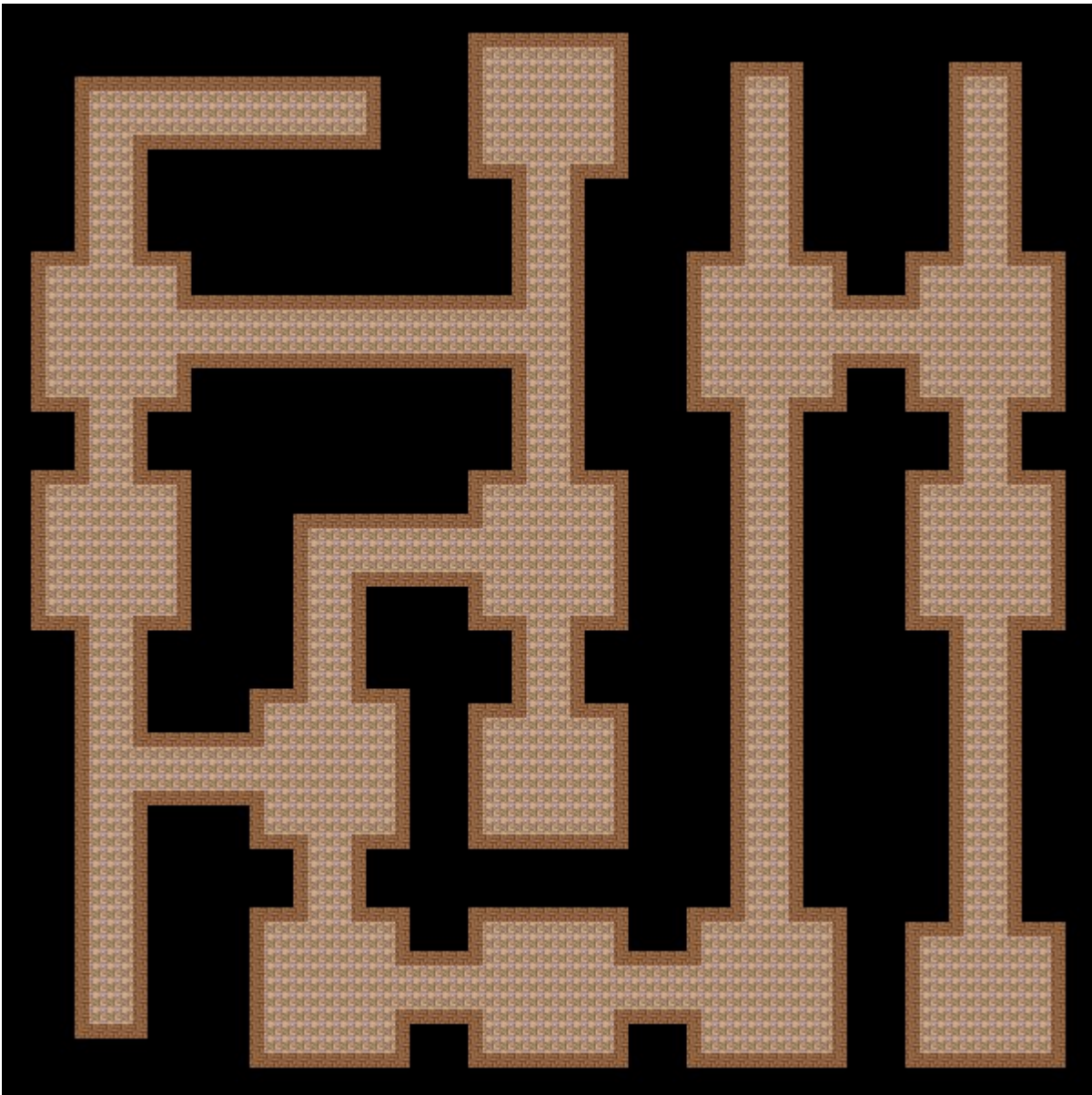


Once you have the maze constructed, building the level is a simple matter. Each cell of the maze represents some arbitrary sized chunk of tiles in the tile map, in this case 15 as each piece in the above image is 15x15 tiles. Iterate the maze structure, obtain the edge pattern code for each cell, and lookup the correct patterns for the given edge code. Perform a rotation on the pattern as needed, and copy the pattern of tiles into the tile map. And voila, you have a maze that looks less like a standard maze and more like some sort of dungeon complex. (Of course, it needs more room randomization, as well as interesting things to put in the rooms).

Here is a sample maze:



And here is one possible variation using the above pieces:



Simplicity in and of itself, really. The technique uses no tricky math whatsoever, not complicated concepts. Just simple graph theory and some pattern copying, yet the results can be an extremely useful tool for your toolbox of random level generation stuff.

EDIT: I just realized I left out 2 pieces. Sorry, my bad. Got in a hurry with the copy/pasting. There are 2 additional pieces for patterns of 0 (no walls) representing 4-way intersections. Use your imagination to know what they look like.

### **Pruning**

I've written a few journal articles as well as some other stuff about dungeon generation. In [this](#) journal entry I whipped up a quick maze/labyrinth generator class that encodes connection information between maze nodes, and in [this](#) journal post I show how the maze nodes can be encoded into a very simple tile-based dungeon map. Of course, the algorithm is very crude, but with some work put into the node pieces, and perhaps even recursive subdivision of some of the nodes to add refinement of detail if necessary, it can be workable. The technique produces space-filling mazes that fill all the nodes of the maze; by pruning side branches or otherwise eliminating nodes, more sparse mazes can be made which result in less square dungeon layouts.

My original Golem 2D isometric created dungeons as a tree. A random room was placed, selected from a pattern. Rooms had 'hot spots' or places along the perimeter where a connection such as a doorway or adjoining room could potentially be placed. When a room was placed in the map, these hotspots were enumerated and placed in a list, along with their connection and type info, designating what types of pieces could connect to it, and how much area was available in the map at that hotspot for a piece to be placed. Then a hotspot was randomly chosen from the list, and a new feature with new hotspots (or no hotspots, in the case of some features) was selected randomly and placed.

This technique worked fairly well, but it isn't really suitable for some types of levels, since it does create a branching tree structure without any loops, unless special case logic during feature selection and placement is written to allow some pieces to overlap. All paths from one branch to another pass through the root node, and this tree-like layout becomes very obvious after the player spends some time exploring.

I achieved some pretty good success with progressive refinement techniques. I would begin with selecting a choice from some finite set of large-scale layouts holding to a particular pattern for a dungeon, and progressively refine each element of the layout, adding more and more detail as I went. The layouts, and the possible paths of refinement for each piece, were determined by the type of level it was. The layouts themselves could be randomly generated, using a maze algorithm with pruning, or they could be hard-coded.

For example, consider the evil Temple of Al'kadum. The torture chambers underneath that temple are said to be quite extensive, and the 'penitent cells' are numerous, dank and cruel. There is an inner nave, where the altar to the hideous god is erected. There are kitchens where the mundane preparation of meals are carried out. There are storerooms where the monks store tithes and tributes 'donated' by the local peasantry. And there are, of course, armories; Al'kadum wishes that the congregation be devout and vigilant, and commands his priests to ensure such vigilance with the sword if need be. All of these things need to be taken into consideration when determining the layout of the Temple.

I usually start with a list of things that *have* to be in the Temple, and a list of things that *can* be in the Temple. A list of must-haves might look like: {Nave, Penitent Cells, Kitchen, Armory, Storage Cells, High-Priest's Quarters, Monk's Quarters, Torture Chambers }. A list of optionals might look like {Treasure Room, Library, Choir Chamber, Statuary Hall, Empty Rooms} and so forth. When a layout is constructed, each node or location in the layout designates the type of location it can refine down to. A block in the layout may be designated 'Penitent Cells' + 'Storage Chamber' + 'Armory', meaning that during refinement it can refine down into any one of these possibilities. Nodes are randomly selected and progressively defined; when selecting what to refine a node down into, the list of must-haves is checked first, and if one of the possibilities is on this list, then it is refined down into that, and the must-have is crossed off the list. This way, I ensure that all of the necessities are taken care of, after which I can start adding in anything from the optionals list. Note that something can be on both lists. For instance, more than one block of Penitent Cells could be generated.

The optionals list gives the unique flavor. Usually, these selections are weighted so that, for instance, Empty Rooms occurs far more frequently than Treasure Room.

Each node type then refines down even further. How many torture chambers are constructed? How are they laid out? They can either be templated with a randomly chosen layout as before, or purely randomly generated using appropriate constraints, depending upon the algorithms you have worked out. Every element of the dungeon can be refined this way, down to the smallest bit of trash in the

corner of the High Priest's toilet chamber; it's all a matter of the levels of refinement you work out.

I've found that the most important thing when designing a dungeon generator is this: make it make sense. Don't just slap up a brain-dead drunkard's walk algorithm and churn out yet another endless series of generic mazes. Every level, every dungeon, should have a purpose, and should be constructed to demonstrate that purpose. Was it built to hold prisoners? Was it built to provide shelter? A place of worship? Simple dry-goods storage? The more sense your areas and dungeons make, the more able the player will be to immerse themselves in the world you have created.

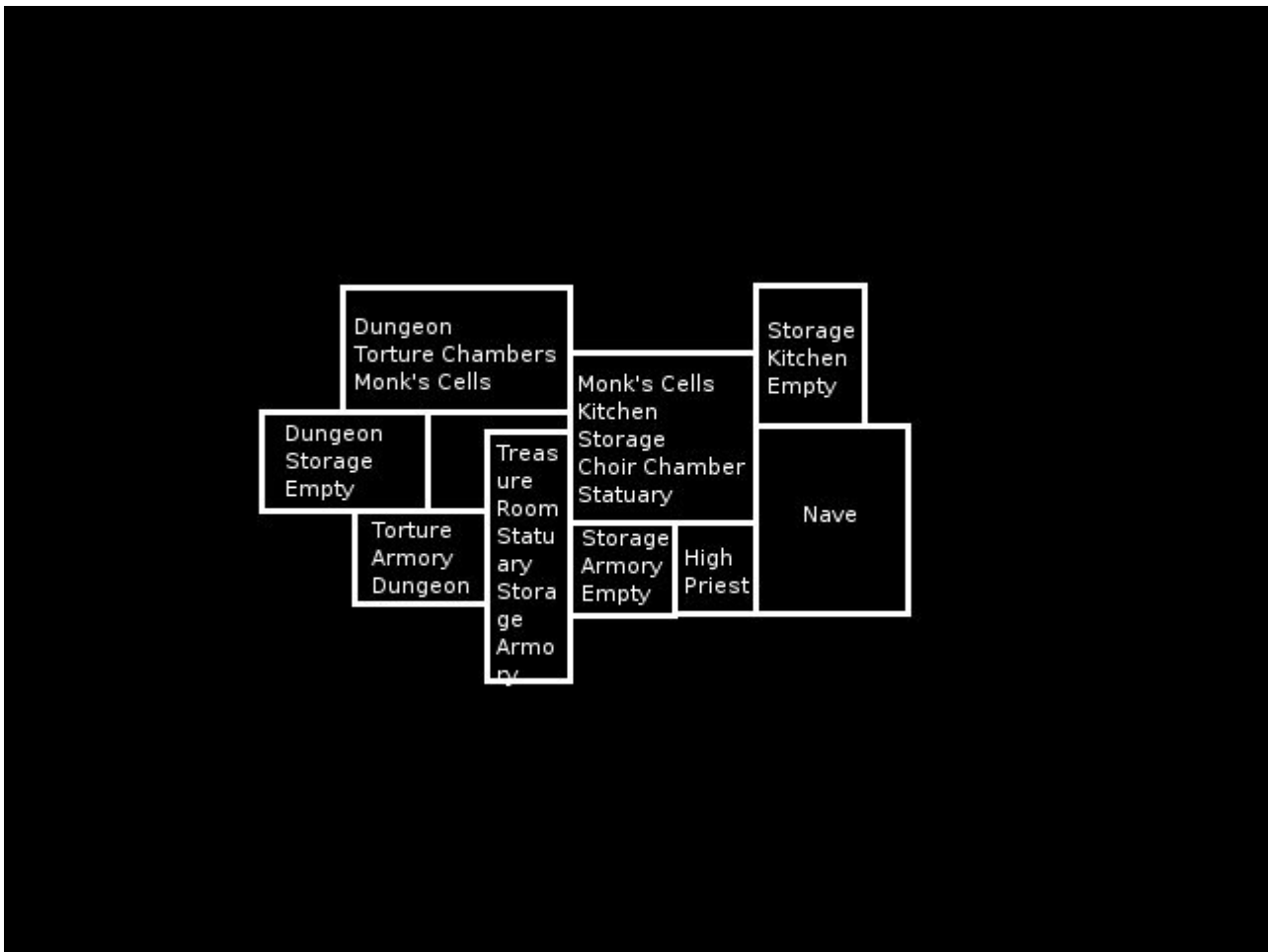
Corollary to making sense is detail: appropriate detail, and sufficient detail. After all, a dungeon is just a bunch of rooms. It's what is in the rooms that counts. This is, of course, largely a matter of artwork and asset creation, but it's also a matter of populating the dungeon in a manner that makes sense.

## **Templates**

I worked a lot with patterns, or pre-generated layouts. When starting on a level-generation routine, the first instinct is to go with 100% procedural. Even now, after all this time, I always want to go 100% procedural. However, sometimes it can simplify things to go with a few pre-generation steps. It's a tradeoff. Maximum randomness means maximum effort. In many cases, you can trade a few procedural, implicit methods for roughly equivalent non-procedural explicit methods, and still obtain sufficient randomness.

For example, going back to the Temple above. There are an infinite number of ways the Temple could be built; however, only a relatively small subset of those ways make sense. The trick lies in applying the constraints to obtain the desired result. A place to start applying the constraints is in the overall layout. As you mentioned, it can be difficult architecting the grand layout procedurally. What if the procedure makes it so monks have to travel through the torture chambers to get to their sleeping cells, or makes it so the monks have to bring prisoners through the sacred nave to get to the prisons? These are things that can add complexity to the procedural layout generator, and thus are candidates for pre-generation.

To begin constructing the level, you might start with a template that contains meta-data about section locations, types, restrictions on neighboring types, etc... Something like this, for example:



This, of course, is just a hastily thrown together example, but it shows what I mean. The physical layout of sections (not rooms; the actual layout of rooms within sections is the responsibility of the next level of refinement) is pre-generated, but each section can possibly be different things based on random selection. This removes the tricky task of procedurally generating the layout, while still providing opportunities for randomization. If a large set (I sometimes created 10 or 15 different layout templates, depending on the complexity of a given level type) of templates is provided, then that still gives you 100s of possible permutations in any given level, and saving some tricky and bug-prone coding as well. And that's not even counting the possible randomization at further levels of refinement.

The principle can extend down into the sections themselves, although I've found that it's typically easier to write fully procedural generators for these kinds of subsections than it is to write a generator for the over-all layout. The section generator is given connection data that records how and where the section connects to neighboring sections (part of the template) and uses that, plus the rectangle of space it is allocated within the level map, to construct the physical layout of rooms within that section. Maze generators can work well here, especially if constraints are applied to the generator. For instance, storage and sleeping quarters type sections might have lots of long hallways with small cubical cells tightly packed along their lengths. Dungeon sections can be made twisty to confuse escapees and funnel them down dead ends for easy re-capture.

The refinement continues as you go deeper. Say you generate a complex of torture chambers. You place some rooms, some twisty passages, maybe a secret door or two to a chamber where the 'special' victims are given the royal treatment. This chamber has an iron maiden and a rack; that chamber has a stocks and a rack full of whips and knives. (Sounds like an S&M nightmare; ugh) This other chamber has a barrel full of acid for slow dipping. All of these possibilities are selected

at random for each room, and placed according to some set of rules (or templates).

All sorts of things are subject to randomization: garbage and litter, wall-hangings, floor coverings, tile patterns, the statues placed around the perimeter of the nave depicting horrid beasts from beyonder planes, and so forth. The idol of the vicious god sits on an altar. What does that altar look like? Is it a slab of purest black basalt? Is it ornately carved of grey granite, and stained with the blood of innocents? What about the statue itself? Does it sit upon an elaborate pedestal, or does it hang from the ceiling? Is there a hidden compartment inside the pedestal? Does that compartment contain gold, a magical item, or a deadly poison trap?

One advantage of using templated layouts is that architectural peculiarities can be reused across several different establishments. If two separate buildings were built by the same builders, then it is likely that certain floor plans or architectural characteristics will be similar, even if the buildings serve different purposes. So certain templates or sub-section templates and rules can be reused for different locations. The nearby fortress, for example, might have been built by monks long ago, and probably features dungeons beneath it chillingly similar to these grim, death-reeking holes.

With a dynamically-typed language such as Lua at the heart of the generation scheme, I can easily prototype a dungeon using a library of templates, and if I come up with some clever generational scheme to do something procedurally, I can swap out the template for the procedure. If they are given the same type of interface, the swapping-out can be seamless and painless.

I haven't played a whole lot with roads and such, outside of the particulars of blending roads well with height-mapped terrains and such; and my towns have all been heavily templated in much this same manner.