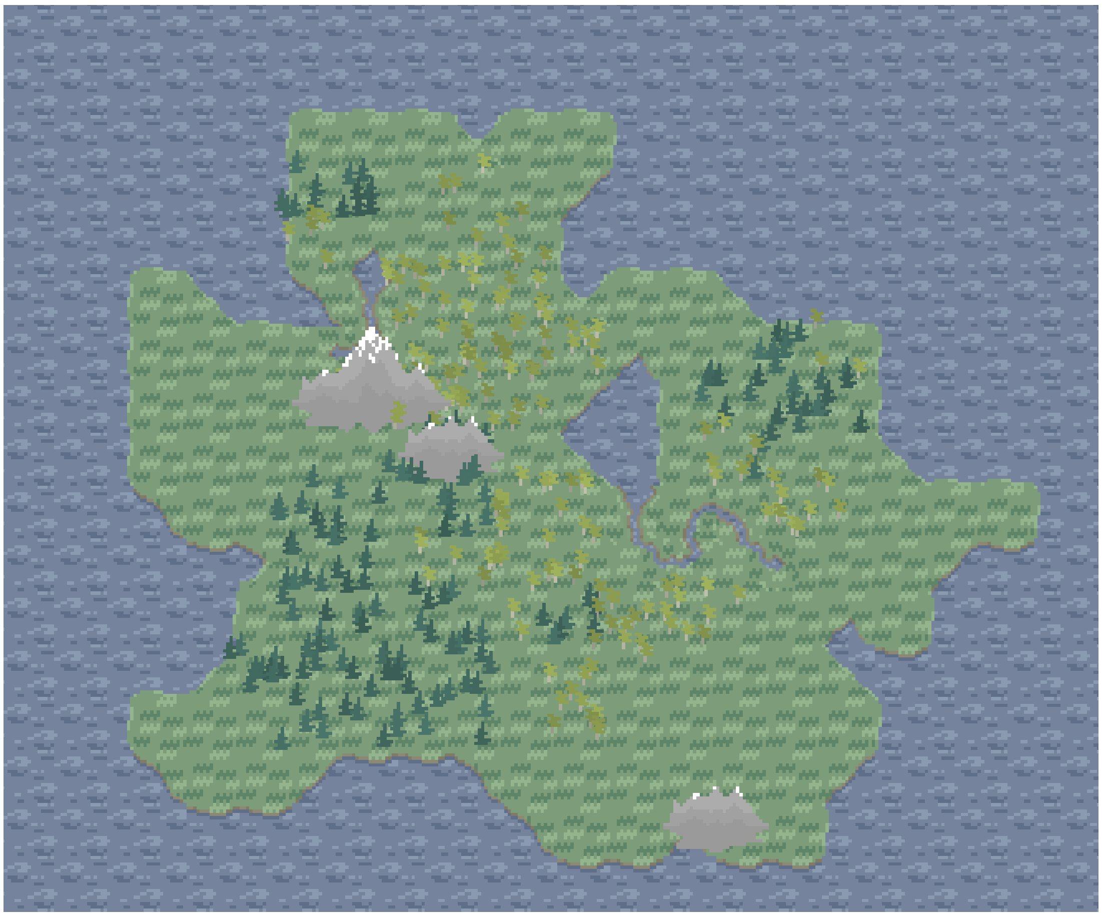
Tile Based World Generation In SDL2

Examensarbete by Gustav Bååth,
Game Creator Programmer
Yrgo - Gothenburg, Fall 2023



Special thanks to Samuel Olsson, who listened to all my whining over discord.

Table of Contents

Introduction

The purpose of this project was a way for me to learn C++ while simultaneously getting experience in backend programming without an editor, mainly in generation of tile-based content in a 2D environment.

Background

During previous game projects, I was (in short terms) always trying to push the limits of “how much randomness can we implement without losing quality”, and so I often experimented with procedural generation, trying to maximise the value of each game asset.
Since most of these implementations have been heavily reliant on the Unity or Unreal editor, I thought it would be a good idea to expand my view and broaden my skill set. Therefore, I settled on the SDL2 framework to be able to apply my previous knowledge of generation into a new programming language to be able to assist me in the learning process. I chose to work with tiles since that is kind of the “entry level” to procedural generation.

Questions

The first overarching question was: “How do I make a 2D world generator using SDL2?” This of course gets split down to several others, corresponding to each part of the production process.
“What is the structure of the C++ workflow?”
“How do I use the SDL2 Library to generate tiles?”
“How can you generate decorations and make them all fit seamlessly with each other?”

Limitations

In order to not get carried away and keep the scope doable, I imposed some limitations to keep my focus on what is relevant for the purpose:

“*Keep the graphics lightweight.*” The point is not to necessarily make something with top tier graphics, but to code the algorithms behind a system that could have.

“*Limit the worldspace.*” How big a forest or field is, is not what shows a good procedural generation system, but rather the amount of subsystems and layers that interact with each other. A good natural environment is therefore an island.

“*Remember to learn C++*” The point of this work was to learn C++ and it is a good idea to sacrifice some of the end result to experiment with several C++ solutions.

Implementation

I have split the implementation process in chronological order so that it is easier to reflect on mistakes and refactoring.

C++ Intro and SDL2 Setup

It is impossible to work with a framework for C++ without knowing C++ so step one was to wrap my head around C++ concepts and get into the right mindset.
I found that learning by doing is the best way for me to work so I kept the tutorials to a minimum and used them mostly just to grasp concepts and get started.
As for the actual development environment setup, I can not recall the exact steps for where to place and link all the header files into visual studio since I literally followed a step by step setup tutorial.
After the setup and light copying of some youtube tutorial code I started experimenting with some input actions and classes. The biggest differences between C++ and C# is memory management, pointers, class structures and some syntax differences, so that’s where I put my focus the first few days.
Something I actually found extremely helpful was to ask ChatGPT to explain concepts and why certain things worked the way they did. It was very helpful to see my own code with explanations and improvements and to ask follow up questions, so I actually wrapped my head around the C++ syntax relatively quickly.

Tilemap System

The base of my entire project is rendering tiles and storing them in a tilemap and I found a great guide on how you could do exactly this in SDL2.
The tilemap info and data is stored in the Tile and Tilemap classes.
Tile.h i holding the local data for itself, like the type of tile it is, the trees that spawn on it, as well as references to its neighbours, while Tilemap.h holds most of the declarations for generation, source images to render, as well as the actual tiles, along with data for pixel size and such.

Based on these variables, I generate a grid of tiles in a 2D array and store it in a tilemap reference in the main gameloop SDL2 uses the y-coordinate of the screen position, top to bottom and I initially stored the tilemap the opposite of this i.e right side up, and this *might* come back to haunt me down the line.

Island Generation

The landscape is generated using a random walker within a confined bounds of the tilemap centre, as is visible during my early screenshots, I had some trouble with random number generation. However after trying some different options, I got it working.
The way it works is that it adds all the tile neighbours of existing land tiles, and randomly selects one to turn into land. This also means that if several tiles have the same neighbour, they get added more times, resulting in tiles near the middle being more favoured.
Now was the time to try and make this project look a little less boring and so I got to borrow a tilemap from my friend that I did some slight modifications to over time.
After a sufficient amount of tiles have been set to land, it is time to start the rule tiling, each tile compares itself to the surrounding land tiles and sets itself to the corresponding enum type.
I had some trouble with the iteration through the edge tiles that were missing some neighbours so I had to add some extra null checks to the function, and limit which coordinates in the tilemap tried to ruletile.
The actual rendering of the different types of tiles is quite clean. The way I understand it is that SDL2 basically takes a part of an imported texture and then projects it onto a rect on the screen. Knowing this, I wrote a function that gave me the coordinates of the source image with a tile type input, then you simply add some pointer variables to the coordinates and send the info to the rendering function.

Trees and Mountains

This is where the majority of time spent is. I went through a lot of refactoring of the systems and there are still things to improve on. (More on that in the result section).
Spawning trees was the first time I got properly introduced to C++ vectors.
Being used to using C# vectors to calculate physics it felt a little strange using vectors as C# lists, however they really are just a container for values.
The first iteration of tree spawning was simply adding a “layered tile” above all existing ones, and randomly selecting some land tiles to spawn on, and then following the same procedure as with the land spawning.
I was not happy with how it looked so I decided to split the tree tiling into individual trees, able to randomly spawn with a density modifier, while still being connected to tiles.
The early iterations of this looked even worse than the first, until I realised that there

was a miscalculation with the rects i rendered to and the source image sizes, only rendering parts of each tree, within incorrect bounds.
Fixing this, and introducing some random colour shading, made things look a lot better.
There still were some issues with the rendering order, with trees with a lower y-value being randomly above others, but before trying solutions for that, I wanted to experiment with mountain generation.

I wanted to have different mountain sizes, and luckily the mountains on the existing sprite map fit seamlessly to each other. The only problem was to actually figure out how to connect them and render them properly. This turned out to be quite difficult to grasp, but I got it working with some hard coded numbers.
The mountain source is split into individual parts that connect to each other, each part is stored in the mountains vector, when the spawn mountain function is called it adds source rects to the rendering list, along with offset data for each mountain part. These parts are then rendered the same way as trees.

To fix the problem with rendering order for trees and mountains, I had to compare them all in a single rendering iteration, easier said than done.
I decided on inheritance and dynamic casting.
This would have been easy, were it not for the fact that there are some hidden rules to whether a class should be classed as inherited.
For one, you have to specify that the inheritance is public, and that the base class *needs* to have a virtual function, even though you never use, nor override it.
Both these conditions, if not fulfilled, does not give an error, but rather a runtime crash with a cryptic error message.
Though with this figured out and fixed, I got the rendering working as intended.

Lakes and Rivers

The lake generation works almost the same as the land with the difference being the extra requirement that it does not connect to the sea, this means that you have to check two steps ahead to make sure the tile neighbour's neighbour is not a water tile. Repeat this a few iterations according to the lake amount parameter, and store them in a vector for the river generation to use.
The river generation saw a whole lot of refactoring, mainly because of the inverted y axis. Most of my logic was made with a right-side-up mindset and so a lot of trial and error went into squashing bugs.
The way rivers were initially done was by generating a "river base" tile and later converting all of them based on the neighbouring tiles. The problem with this was that I needed exponentially more enum comparisons, and it was difficult to follow the logic process and debug.

I instead opted for a version with a long switch, randomising “legal” connection based on the previous connection point, and which neighbouring tile can be connected to. This is still very prone to programmer error, since every single tile is required to have the correct “connection data” and takes a while to debug.
If no connections are available the river closes and replaces the last tile.
There were more rule tiling errors than I first anticipated, and so they ended up in a somewhat bugged state, since my original generation idea kind of clashed with the refactored version, making it hard to make the rivers feel “natural”.

Result

The result is an island generation tool, which can generate different islands with contents based on code input.
This includes, land size, mountain size and amount, tree density, lake count and sizes, and rivers.
My skill set in backend programming, and SDL2 have gone from nothing to somewhat independent, and my skills with C++ have been substantially improved.
I believe I am now very capable to work on actual C++ projects as well understand and evaluate C++ code on a level almost on par with C#. I still have a long way to go, as with all code languages, but I see this project as a great head start.

Improvements

I do not believe I could have changed the process in such a way that would allow me to have done a better job, neither with the procedural generation nor how much of C++ I have learned.
It is still a fairly simple project however I still believe it held the right difficulty and pace going from nothing to something.

However in retrospect, there are a lot of things I now know how to improve, many of which are completely different systems on how I would do things from the beginning. This, of course, stems from my improved C++ skills and the mistakes I made.

Rivers

The most important improvement that could be made, actually shows in the finished product, that being river tiling as shown in this example.
When I drew the river tiles I chose a version with a full land tile behind them. This meant that I had to fine tune the generation with a lot of extra parameters, both for the rivers themselves, as well as the actual regular land tiling.
It would require less code to just have a river overlay, much like trees, linked to each tile. This would mean less focus on making them fit with the rest of the world, and more focus on actually generating natural shapes and connections.

Trees

Even though the tree generation looks quite natural and polished there are still improvements to be done on the density. As of now, the density is chosen at random within a min and max value.
A way to improve on this is to be able generate density based on neighbouring forest tiles, slightly modifying the value and making a smoother transition.

Mountains

The best way to improve the mountains would be to introduce some form of vertical connection, making mountains take up a larger space on the y-axis.
Another improvement would be to either introduce some form of transition to the water, or prevent mountains from spawning above water completely.
Preventing the spawning would require a lot of extra edgecase code, so I believe the best alternative is to modify the sprite so that I could transition well to both land and water.

Additions

There are parts that were originally planned that I didn’t have time to make, mainly because of my unfortunate absence. These include biomes, towns and roads.
I believe it would be quite interesting to try and implement some forms of conditional spawning for towns and roads, requiring me to really deep dive into the spawning logic. This would probably require a cleaner design of mainly rivertiling to avoid excess case comparisons.
This could go even deeper with logic for multiple towns and spawn conditions for roads and bridges, as well as special buildings in different biomes or environments.
I think it would be a lot of fun, and a great learning experience to try and implement some of these examples.

Bibliography

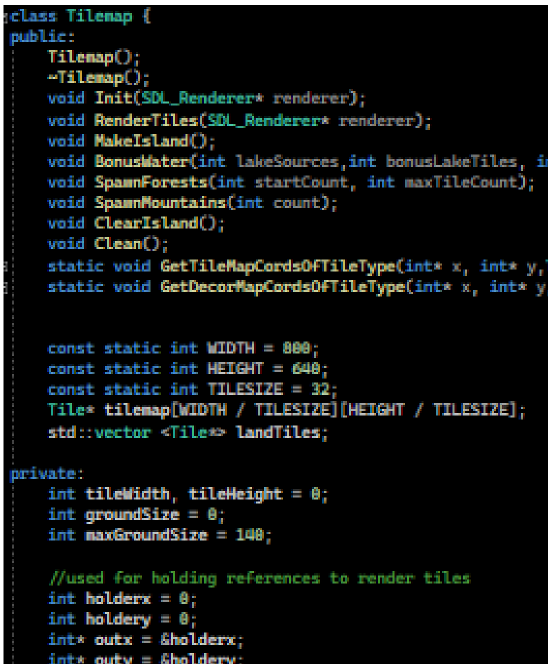
Basic SDL2 project tutorials
https://www.parallelrealities.co.uk/tutorials/

Guide to SDL2 tilemaps
https://www.youtube.com/watch?v=DNu8yUsxOnE

SDL2 setup with VS
https://www.youtube.com/watch?v=QQzAHcojEKg&list=PLhfAbcv9cehhkG7ZQK0nflGJC_C-wSLrx

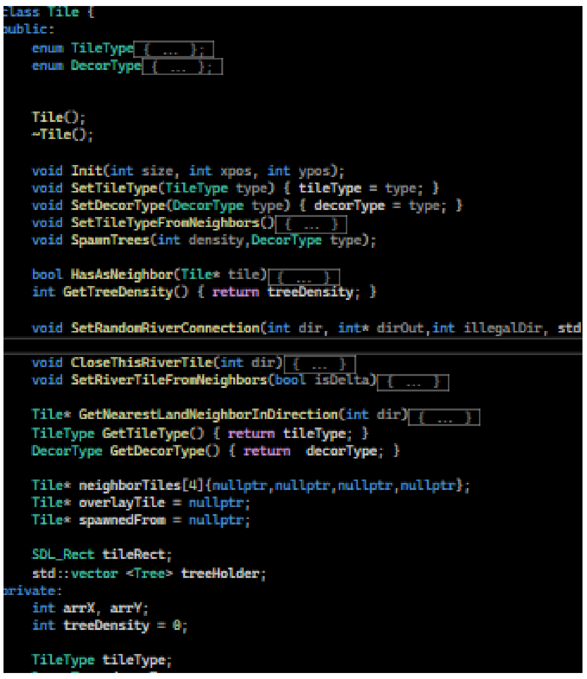
Image references

Tile.h



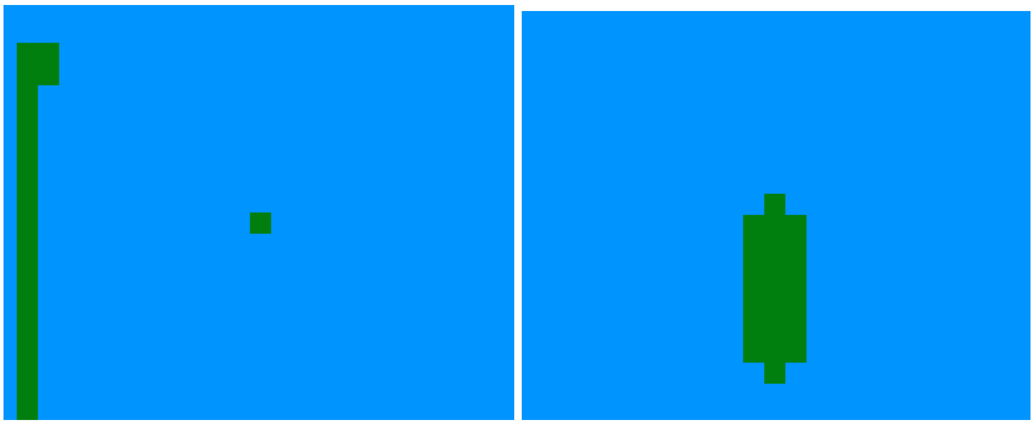
```
class Tilemap {
public:
    Tilemap();
    ~Tilemap();
    void Init(SDL_Renderer* renderer);
    void RenderTiles(SDL_Renderer* renderer);
    void MakeIsland();
    void BonusWater(int lakeSources,int bonusLakeTiles, i
    void SpawnForests(int startCount, int maxTileCount);
    void SpawnMountains(int count);
    void ClearIsland();
    void Clean();
    static void GetTileMapCordsOfTileType(int* x, int* y,
    static void GetDecorMapCordsOfTileType(int* x, int* y


    const static int WIDTH = 800;
    const static int HEIGHT = 640;
    const static int TILESIZE = 32;
    Tile* tilemap[WIDTH / TILESIZE][HEIGHT / TILESIZE];
    std::vector <Tile*> landTiles;

private:
    int tileWidth, tileHeight = 0;
    int groundSize = 0;
    int maxGroundSize = 140;

    //used for holding references to render tiles
    int holderx = 0;
    int holdery = 0;
    int* outx = &holderx;
    int* outy = &holdery;
```

Tilemap.h

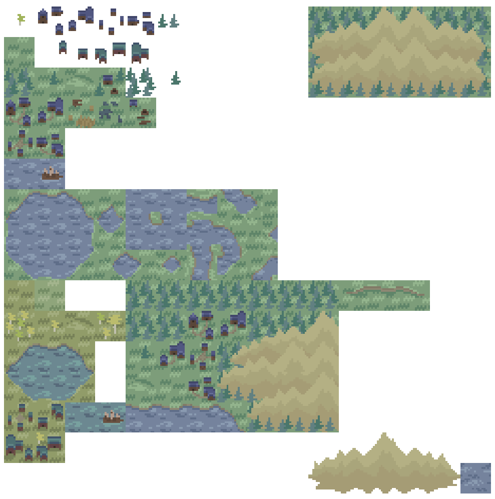


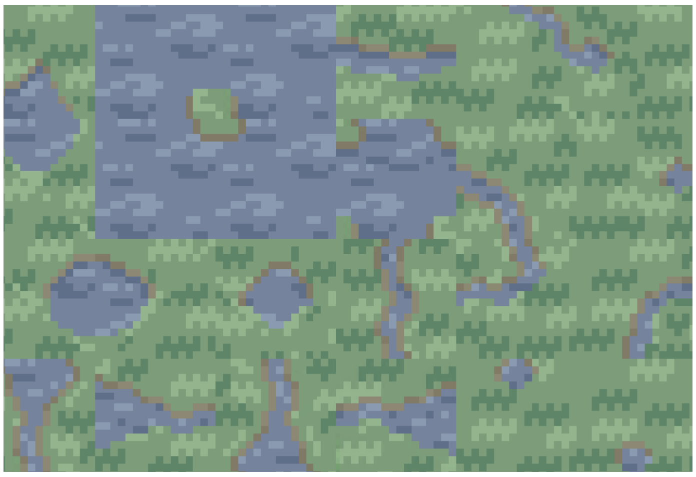
```
lass Tile {
ublic:
    enum TileType { ... };
    enum DecorType { ... };


    Tile();
    ~Tile();

    void Init(int size, int xpos, int ypos);
    void SetTileType(TileType type) { tileType = type; }
    void SetDecorType(DecorType type) { decorType = type; }
    void SetTileTypeFromNeighbors() { ... }
    void SpawnTrees(int density,DecorType type);

    bool HasAsNeighbor(Tile* tile) { ... }
    int GetTreeDensity() { return treeDensity; }

    void SetRandomRiverConnection(int dir, int* dirOut,int illegalDir, std

    void CloseThisRiverTile(int dir) { ... }
    void SetRiverTileFromNeighbors(bool isDelta) { ... }

    Tile* GetNearestLandNeighborInDirection(int dir) { ... }
    TileType GetTileType() { return tileType; }
    DecorType GetDecorType() { return  decorType; }

    Tile* neighborTiles[4]{nullptr,nullptr,nullptr,nullptr};
    Tile* overlayTile = nullptr;
    Tile* spawnedFrom = nullptr;

    SDL_Rect tileRect;
    std::vector <Tree> treeHolder;
rivate:
    int arrX, arrY;
    int treeDensity = 0;

    TileType tileType;
```

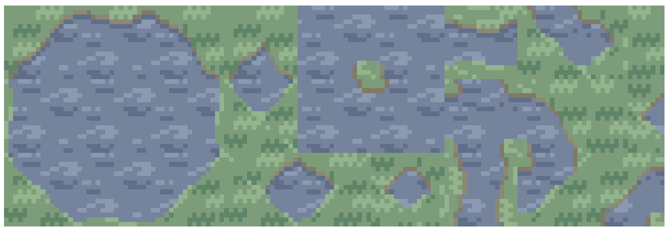
First stage of land gen



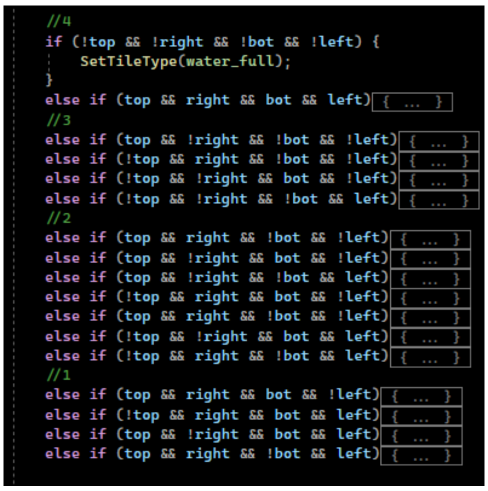
Tilemap



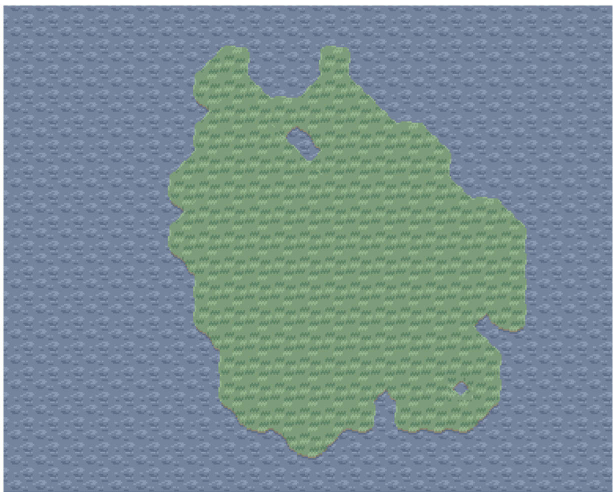




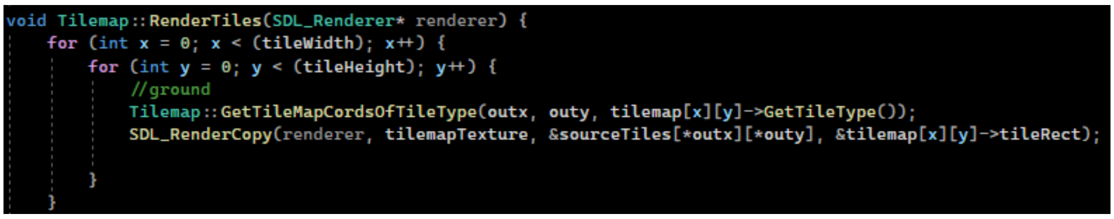
Rule Tiling



```
//4
if (!top && !right && !bot && !left) {
    SetTileType(water_full);
}
else if (top && right && bot && left){ ... }
//3
else if (top && !right && !bot && !left){ ... }
else if (!top && right && !bot && !left){ ... }
else if (!top && !right && bot && !left){ ... }
else if (!top && !right && !bot && left){ ... }
//2
else if (top && right && !bot && !left){ ... }
else if (top && !right && bot && !left){ ... }
else if (top && !right && !bot && left){ ... }
else if (!top && right && bot && !left){ ... }
else if (top && right && !bot && !left){ ... }
else if (!top && !right && bot && left){ ... }
else if (!top && right && !bot && left){ ... }
//1
else if (top && right && bot && !left){ ... }
else if (!top && right && bot && left){ ... }
else if (top && !right && bot && left){ ... }
else if (top && right && !bot && left){ ... }
```

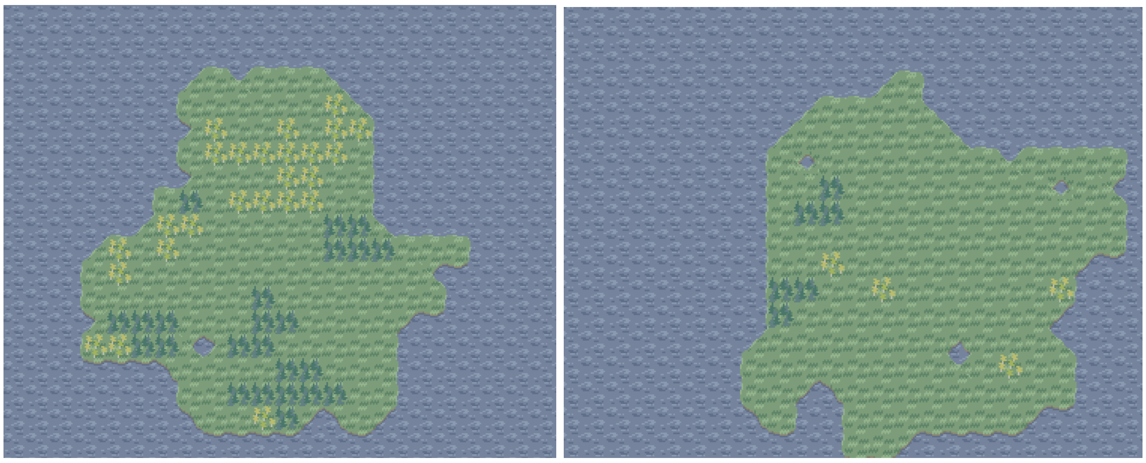


Tilemap Rendering

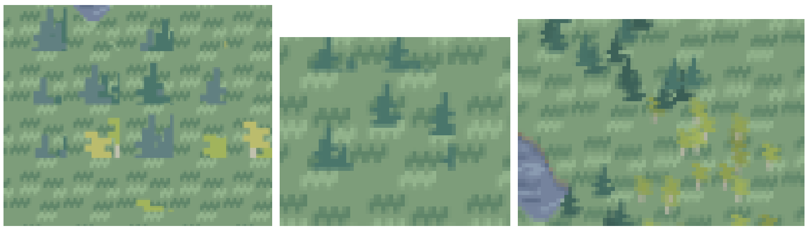


```
void Tilemap::RenderTiles(SDL_Renderer* renderer) {
    for (int x = 0; x < (tileWidth); x++) {
        for (int y = 0; y < (tileHeight); y++) {
            //ground
            Tilemap::GetTileMapCordsOfTileType(outx, outy, tilemap[x][y]->GetTileType());
            SDL_RenderCopy(renderer, tilemapTexture, &sourceTiles[*outx][*outy], &tilemap[x][y]->tileRect);

        }
    }
```

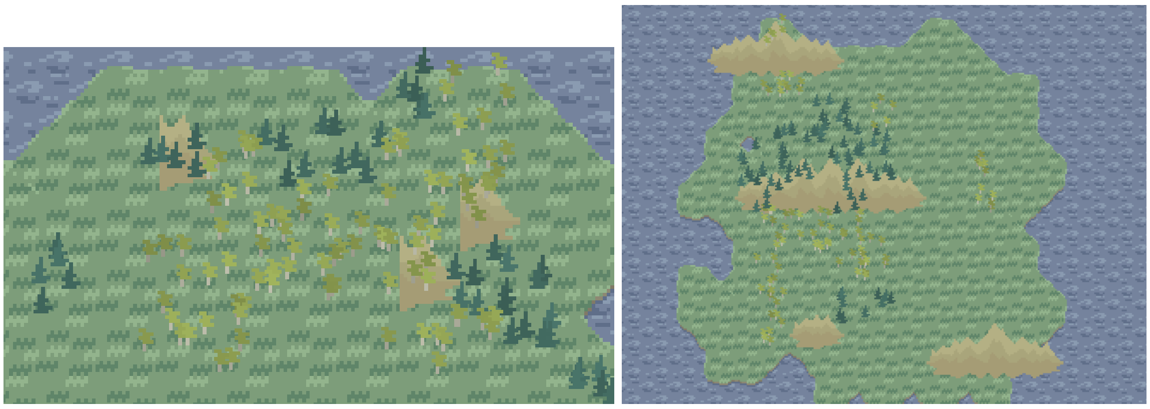
Early Trees



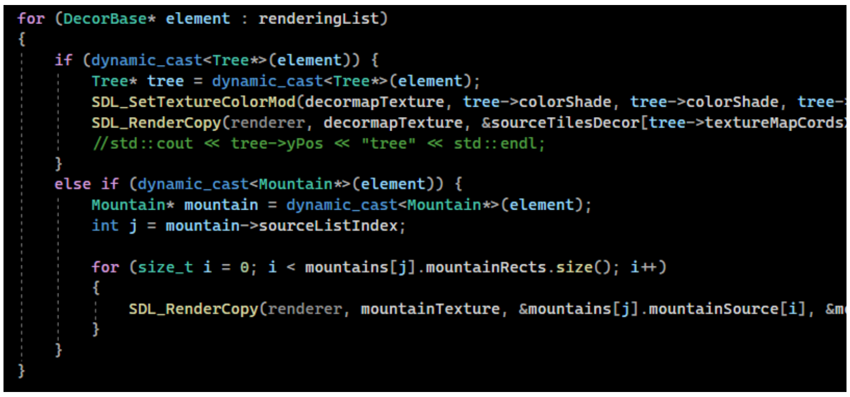
Trees iteration 1



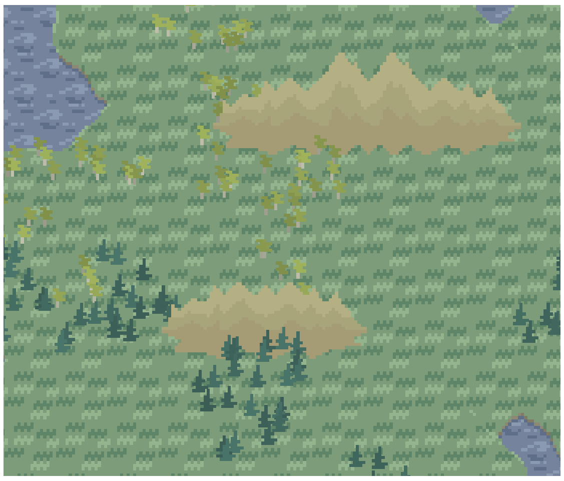
Mountains

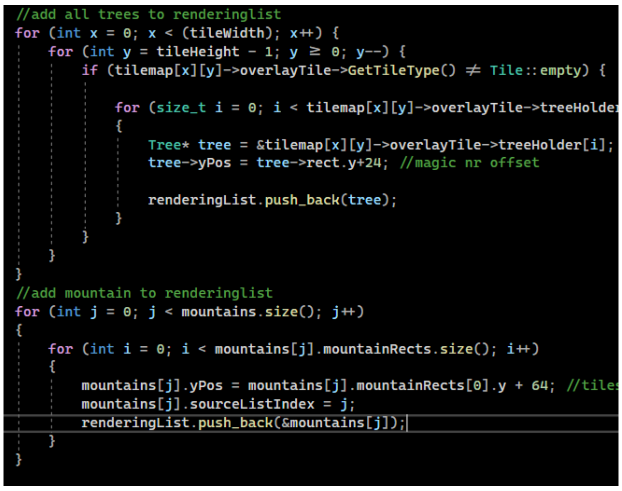


Fixed Rendering



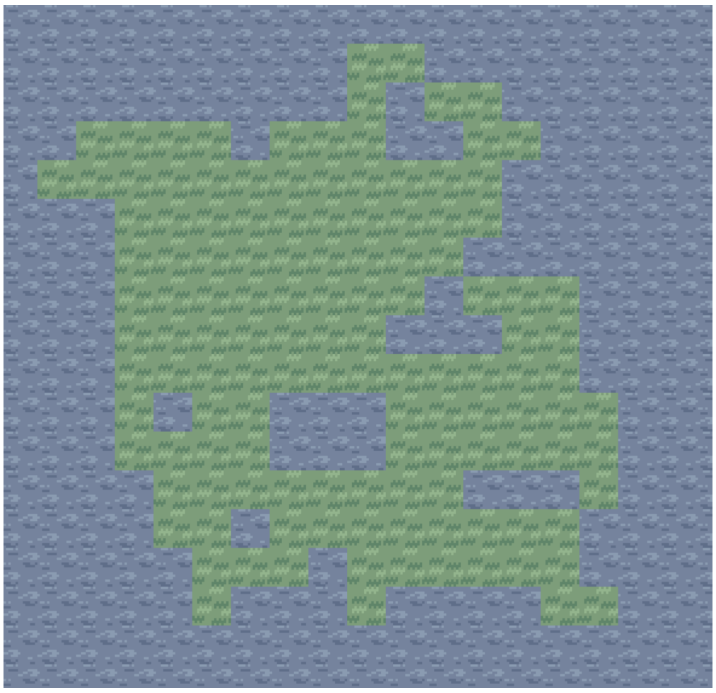
```
for (DecorBase* element : renderingList)
{
    if (dynamic_cast<Tree*>(element)) {
        Tree* tree = dynamic_cast<Tree*>(element);
        SDL_SetTextureColorMod(decormapTexture, tree->colorShade, tree->colorShade, tree-
        SDL_RenderCopy(renderer, decormapTexture, &sourceTilesDecor[tree->textureMapCords
        //std::cout << tree->yPos << "tree" << std::endl;
    }
    else if (dynamic_cast<Mountain*>(element)) {
        Mountain* mountain = dynamic_cast<Mountain*>(element);
        int j = mountain->sourceListIndex;

        for (size_t i = 0; i < mountains[j].mountainRects.size(); i++)
        {
            SDL_RenderCopy(renderer, mountainTexture, &mountains[j].mountainSource[i], &m
        }
    }
}
```



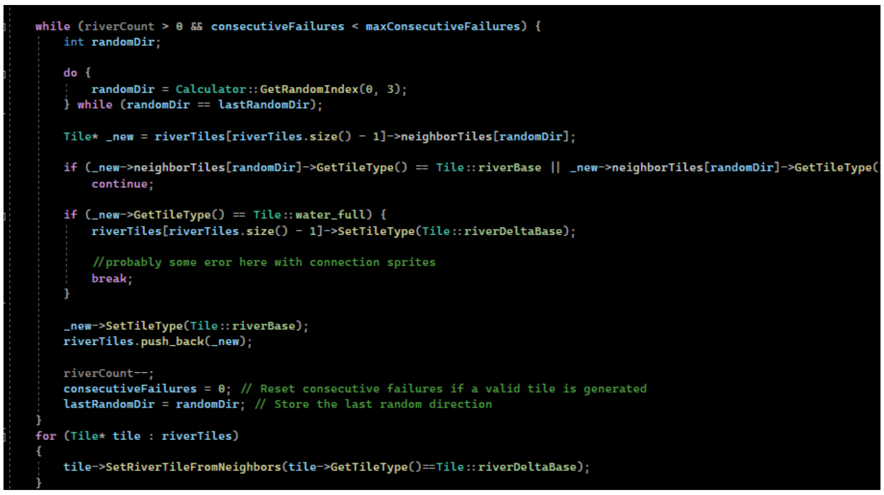


```
//add all trees to renderinglist
for (int x = 0; x < (tileWidth); x++) {
    for (int y = tileHeight - 1; y ≥ 0; y--) {
        if (tilemap[x][y]->overlayTile->GetTileType() ≠ Tile::empty) {

            for (size_t i = 0; i < tilemap[x][y]->overlayTile->treeHolde
            {
                Tree* tree = &tilemap[x][y]->overlayTile->treeHolder[i];
                tree->yPos = tree->rect.y+24; //magic nr offset

                renderingList.push_back(tree);
            }
        }
    }
}
//add mountain to renderinglist
for (int j = 0; j < mountains.size(); j++)
{
    for (int i = 0; i < mountains[j].mountainRects.size(); i++)
    {
        mountains[j].yPos = mountains[j].mountainRects[0].y + 64; //tile
        mountains[j].sourceListIndex = j;
        renderingList.push_back(&mountains[j]);
    }
}
```

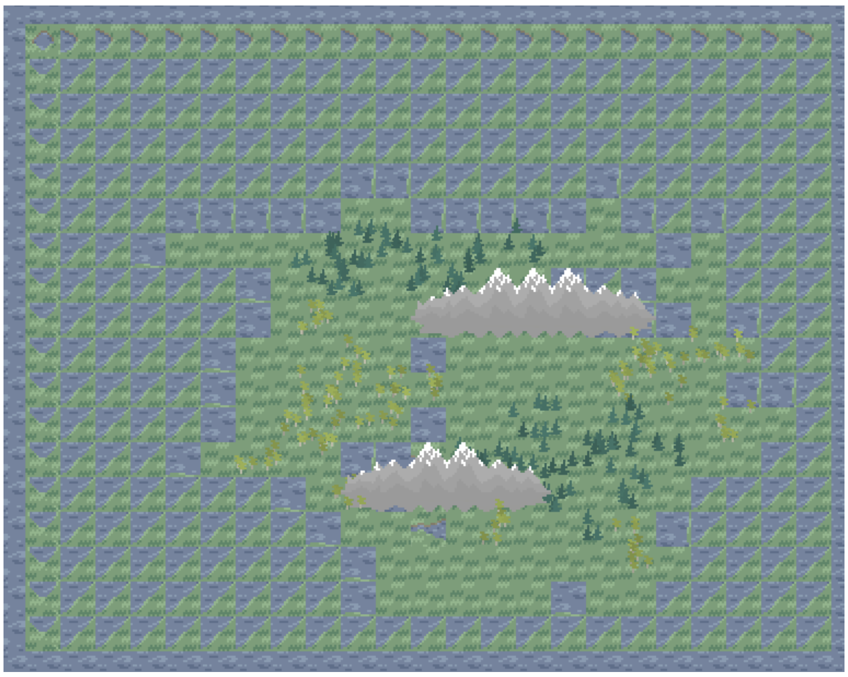
Lake generation basics



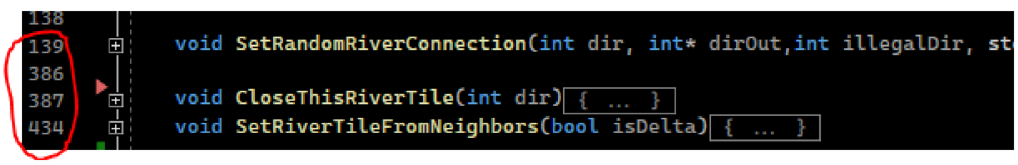
Broken river generation



```cpp
while (riverCount > 0 && consecutiveFailures < maxConsecutiveFailures) {
    int randomDir;

    do {
        randomDir = Calculator::GetRandomIndex(0, 3);
    } while (randomDir == lastRandomDir);

    Tile* _new = riverTiles[riverTiles.size() - 1]->neighborTiles[randomDir];

    if (_new->neighborTiles[randomDir]->GetTileType() == Tile::riverBase || _new->neighborTiles[randomDir]->GetTileType(
        continue;

    if (_new->GetTileType() == Tile::water_full) {
        riverTiles[riverTiles.size() - 1]->SetTileType(Tile::riverDeltaBase);

        //probably some eror here with connection sprites
        break;
    }

    _new->SetTileType(Tile::riverBase);
    riverTiles.push_back(_new);

    riverCount--;
    consecutiveFailures = 0; // Reset consecutive failures if a valid tile is generated
    lastRandomDir = randomDir; // Store the last random direction
}
for (Tile* tile : riverTiles)
{
    tile->SetRiverTileFromNeighbors(tile->GetTileType()==Tile::riverDeltaBase);
}
```



Unpolished river generation



```cpp
138
139     void SetRandomRiverConnection(int dir, int* dirOut,int illegalDir, st
386
387     void CloseThisRiverTile(int dir) { ... }
434     void SetRiverTileFromNeighbors(bool isDelta) { ... }
```