МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА - Российский технологический университет»

РТУ МИРЭА

ЗАДАНИЕ НА ПРАКТИКУ ПО ПОЛУЧЕНИЮ ПРОФЕССИОНАЛЬНЫХ УМЕНИЙ И ОПЫТА ПРОФЕССИОНАЛЬНОЙ ДЕЯТЕЛЬНОСТИ

студенту 3 курса учебной группы КМБО-01-18 Института кибернетики

Алиеву Мишану Хаммад оглы

(фамилия, имя и отчество)

Место и время практики: Институт кибернетики, кафедра высшей математики.
Время практики: с «09» февраля 2021 по «31» мая 2021.
Должность на практике: практикант.

1. **ЦЕЛЕВАЯ УСТАНОВКА:** «Процедурная генерация игровой карты».

2. СОДЕРЖАНИЕ ПРАКТИКИ:

2.1. Изучить: общие принципы процедурной генерации игровых карт.
2.2. Практически выполнить: разработать модель игровой карты, зависящей от входных параметров, и реализовать алгоритм её процедурной генерации.
2.3. Ознакомиться: с существующими методами процедурной генерации игровых карт.

3. ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: визуализировать реализации модели игровой карты при одинаковых входных параметрах.

4. ОГРАНИЗАЦИОННО-МЕТОДИЧЕСКИЕ УКАЗАНИЯ:

4.1. Привести краткие теоретические сведения о процедурной генерации игровых карт.
4.2. Разработать модель игровой карты, зависящей от входных параметров, и реализовать алгоритм её процедурной генерации.
4.3. Визуализировать реализации модели игровой карты при одинаковых входных параметрах.
4.4. Написать вывод о проделанной работе.

Заведующий кафедрой
высшей математики _____ Ю.И. Худак
(подпись)

«__»_____20__г.

СОГЛАСОВАНО
Руководитель практики от кафедры:
«__»_____20__г. _____ В.Б. Федоров
(подпись)

Задание получил:
«__»_____20__г. _____ М.Х. Алиев
(подпись)

ИНСТРУКТАЖ ПРОВЕДЕН:

РТУ МИРЭА, Институт кибернетики, кафедра высшей математики

Вид мероприятия	**ФИО ответственного, подпись, дата**	**ФИО студента, подпись, дата**
Охрана труда	В.Б. Федоров _____ «10» февраля 2021 г.	М.Х. Алиев _____ «10» февраля 2021 г.
Техника безопасности	В.Б. Федоров _____ «10» февраля 2021 г.	М.Х. Алиев _____ «10» февраля 2021 г.
Пожарная безопасность	В.Б. Федоров _____ «10» февраля 2021 г.	М.Х. Алиев _____ «10» февраля 2021 г.
Правила внутреннего распорядка	В.Б. Федоров _____ «10» февраля 2021 г.	М.Х. Алиев _____ «10» февраля 2021 г.

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА - Российский технологический университет»

РТУ МИРЭА

Институт кибернетики

Кафедра высшей математики

ОТЧЁТ ПО ПРАКТИКЕ ПО ПОЛУЧЕНИЮ ПРОФЕССИОНАЛЬНЫХ УМЕНИЙ И ОПЫТА ПРОФЕССИОНАЛЬНОЙ ДЕЯТЕЛЬНОСТИ

(указать вид практики)

Тема практики: Процедурная генерация игровой карты.

приказ университета о направлении на практику
от «09» февраля 2021 г. № 490-С

Отчет представлен к рассмотрению:
студент группы КМБО-01-18 _____ М.Х. Алиев
«__» _____ 2021г.

Отчет утвержден.
Допущен к защите:

Руководитель практики от кафедры _____ В.Б. Федоров
«__» _____ 2021г.

Москва 2021

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА - Российский технологический университет»

РТУ МИРЭА

РАБОЧИЙ ГРАФИК ПРОВЕДЕНИЯ ПРАКТИКИ ПО ПОЛУЧЕНИЮ ПРОФЕССИОНАЛЬНЫХ УМЕНИЙ И ОПЫТА ПРОФЕССИОНАЛЬНОЙ ДЕЯТЕЛЬНОСТИ

студента Алиева Мишана Хаммад оглы 3 курса группы КМБО-01-18
очной формы обучения, обучающегося по направлению подготовки 01.03.02 «Прикладная математика и информатика», профиль – «Математическое моделирование и вычислительная математика».

Неделя	Сроки выполнения	Этап	Отметка о выполнении
1	09.02.2021	Выбор темы практики. Прохождение инструктажа по технике безопасности.	
1	09.02.2021	Вводная установочная лекция.	
2-10	16.04.2021	Изучение литературы по теме практики.	
11-13	07.05.2021	Разработка модели карты и выбор метода её процедурной генерации.	
14-16	25.05.2021	Реализация метода и написание отчёта.	
22	05.07.2021	Представление отчетных материалов по практике и их защита. Передача обобщенных материалов на кафедру для архивного хранения.	
		Зачетная аттестация.	

Согласовано:

Заведующий кафедрой _____ Ю.И. Худак

Руководитель практики от кафедры _____ В.Б. Федоров

Обучающийся _____ М.Х. Алиев

Оглавление

Введение 3
Постановка задачи 4
Краткие теоретические сведения 5
Разработка модели карты 12
Реализация алгоритма генерации 14
Описание алгоритма 14
Подготовка к реализации 14
Реализация класса «Map» 16
Реализация структуры «House» 19
Реализация класса «District» 19
Дополнительное задание 22
Вывод 26
Список литературы 27
Приложение 28

Введение

Процедурная генерация – метод автоматического создания данных, использующий различные алгоритмы, вычислительную мощность и возможности псевдорандома техники (далее будем опускать первую часть таких слов, т.е. корень «псевдо», и под терминами «случайно» и т.п. будем подразумевать «псевдослучайно»), а также заданные человеком параметры.

В современном мире процедурная генерация используется во многих областях программирования: в компьютерной графике, в видеоиграх, в синтезе звуков и т.д. Я бы хотел продемонстрировать использование данной методики в реализации модели карты для видеоигры.

Постановка задачи

Разработать модель карты города и реализовать алгоритм её генерации для каждой новой игровой сессии. Карта должна состоять из параллельных пластов прямоугольных районов (горизонтальные линии районов должны быть параллельны друг другу), в которых есть по несколько домов. От каждой двери дома провести тропинку, идущую к более крупной дороге, окружающей район. Размеры карты, домов, дорог, а также количество районов и домов в них реализовать в виде входных параметров, от которых также зависит генерация.

Краткие теоретические сведения

Задача требует генерации новой карты, зависящей от параметров, при каждой новой игре. Воспользуемся процедурной генерацией карт.

Существует большое число методов процедурной генерации карт, но ни один из них не является универсальным решением: часто применяются модернизированные под данную ситуацию методы, или их смешение, или даже создание программистом своего собственного алгоритма генерации. Рассмотрим наиболее известные способы процедурной генерации карт, где примерами будут карты подземелий, генерируемые этим методами:

1) BSP-деревья (BSP tree method)

   В основе метода лежит случайное создание прямоугольных комнат на карте, соединенных между собой коридорами. На рисунке 1.1 показан пример уже сгенерированной рандомной карты, а на рисунке 1.2 показана та же карта, но уже в игре (добавлены тайлы – квадратные растровые изображения равной величины, которыми как плиткой покрывают поверхность отображаемой карты).

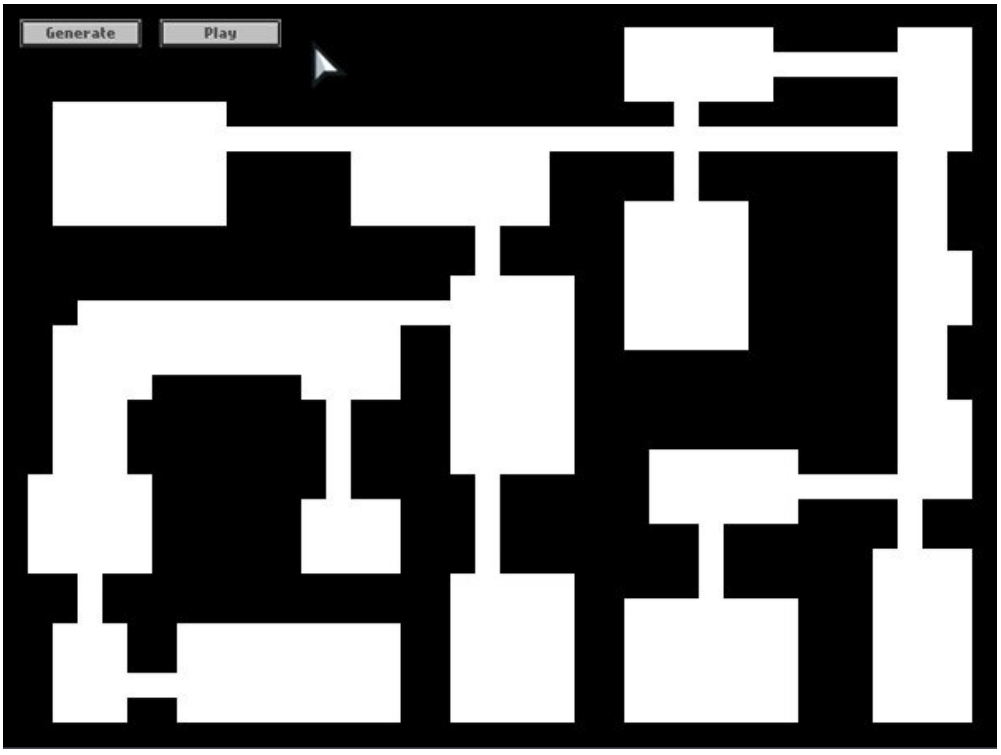

Рисунок 1.1. Сгенерированная с помощью BSP-деревьев карта

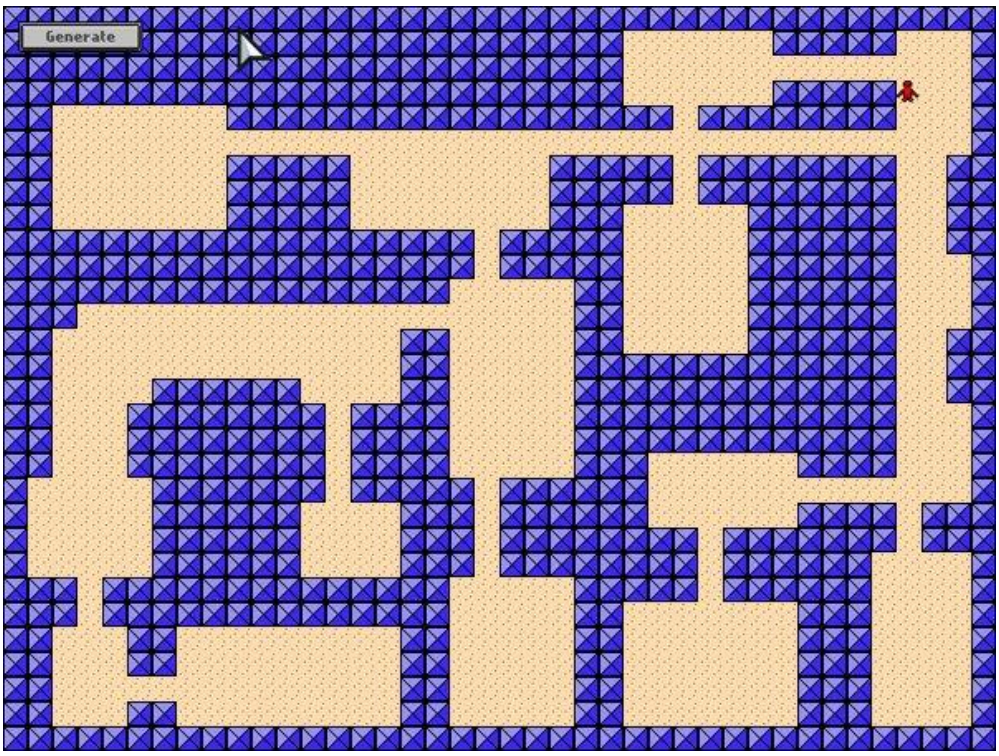

Рисунок 1.2. Сгенерированная с помощью BSP-деревьев карта в игре

Рассмотрим процесс генерации карты с рисунка 1.1. В начале она случайно разбивается две части, затем аналогично разбивается каждая из двух частей и т.д., пока любая часть на карте не достигнет примерно заданного минимума. Получаем карту, разбитую на прямоугольные области (см. рисунок 1.3), в которых затем генерируются по комнате со случайным размером и положением (см. рисунок 1.4). После удаления разделительных линий (см. рисунок 1.5) остаются только комнаты, которые затем соединяются коридорами разных размеров (в данном случае каждая комната имеет до двух коридоров), и получается финальное подземелье (см. рисунок 1.1).

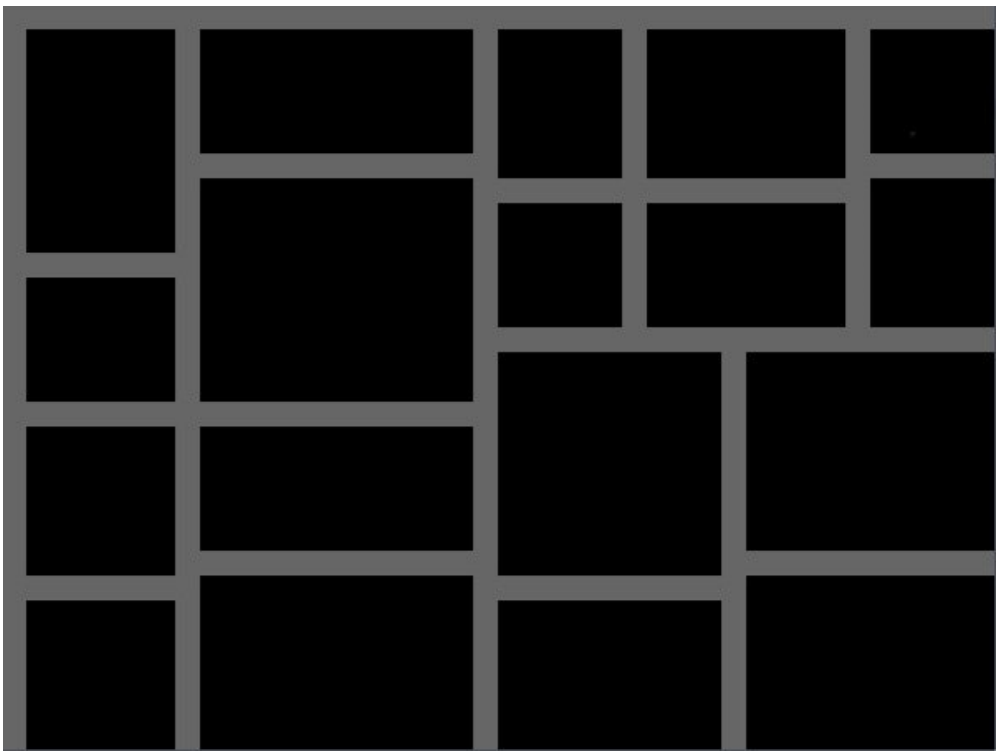

Рисунок 1.3. Карта, разбитая на прямоугольные области

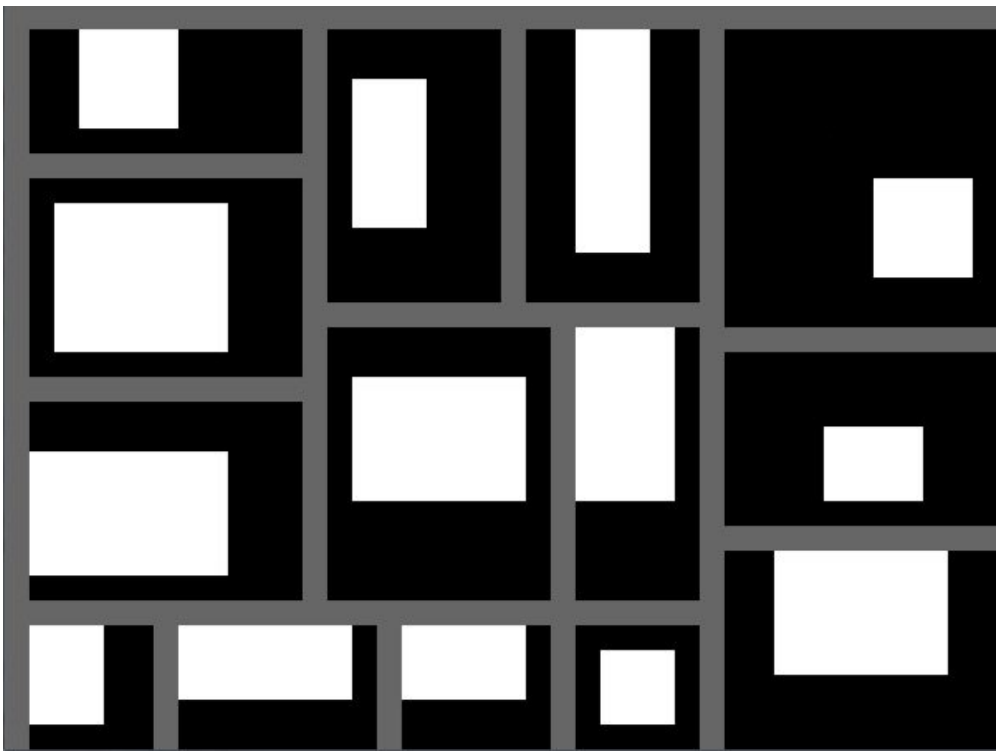

Рисунок 1.4. Прямоугольные области с комнатами случайного размера и положения внутри

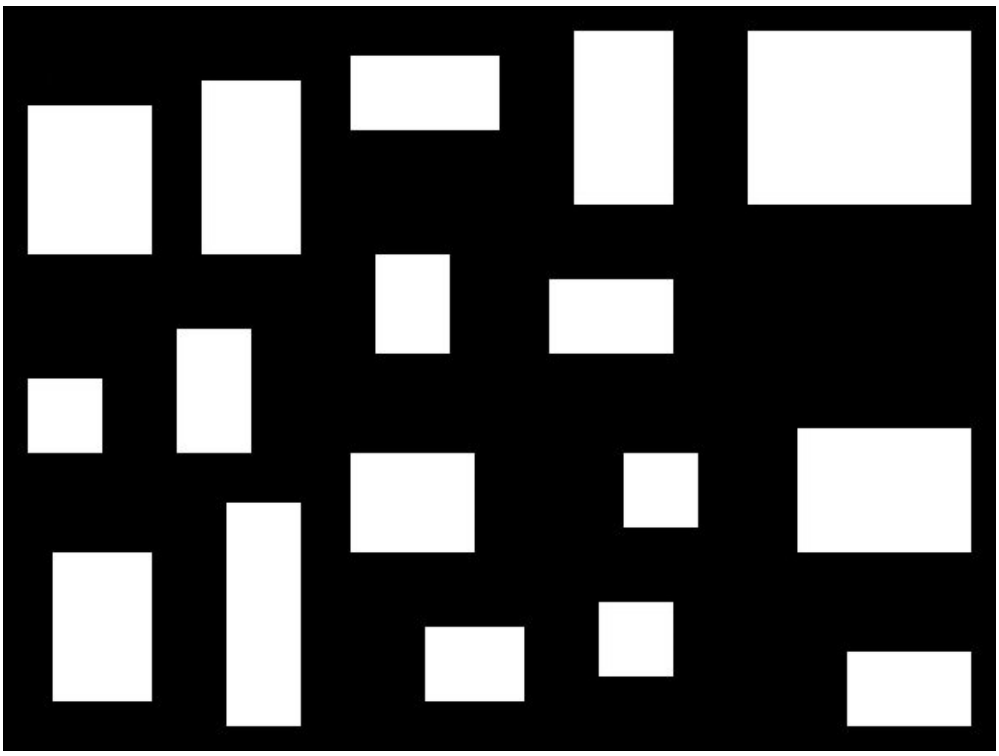

Рисунок 1.5. Прямоугольные области без разделительных линий с комнатами случайного размера и положения

2) Алгоритмы туннелирования (tunneling algorithms)

Суть методов – прокладывание по сплошной карте туннелей и комнат. Один из наиболее известных алгоритмов туннелирования является алгоритм «походки пьяницы» («Drunkard's Walk»). В качестве разъяснения приведем его псевдокод. Введем следующие понятия: «стена» – непроходимое препятствие в будущей карте, «пустая зона» - область, по которой можно будет двигаться. Тогда получим следующий алгоритм:

1. *инициализировать все ячейки карты как стены.*
2. *выбрать ячейку карты в качестве отправной точки.*
3. *превратить выбранную ячейку карты в пустую зону.*
4. *пока недостаточное количество стен превращено в пустую зону*
    a. *сделать один шаг в случайном направлении*
    b. *если новая ячейка карты – стена*
        i. *превратить новую ячейку карты в пустую зону*
        ii. *увеличить счётчик пустых зон.*

Получим результат, показанный на рисунке 2.1, где символ «#» обозначает «стену», «·» - «пустая зона», а за «<» и «>» обозначены соответственно начальное и конечные положения копателя.

Рисунок 2.1. Результат работы алгоритма «Drunkard’s Walk»

3) Клеточные автоматы

Основная идея метода заключается в том, чтобы заполнить случайным образом карту уже известными из пункта 2) «стенками» и «пустыми зонами», а затем по определенному правилу получать новые итерации этой карты. Пример такого правила: элементарная ячейка является «стеной», если область 3×3 с центром в ней содержит как минимум 5 «стен».

На рисунке 3.1 приведен пример. Черным обозначены «стенки», синим – «пустые зоны», и ставится вопрос, что будет находится в клетке с координатами (x, y), выделенной желтой рамкой. В области 3×3 с центром в ячейке (x, y), выделенной красным цветом, 4 «стенки» и 5 «пустых зон» (в самой ячейке изначально находится «пустая зона»), следовательно, ячейка (x, y) останется «пустой зоной».

Таким образом, каждая итерация делает каждую элементарную ячейку более похожей на своих соседей. На рисунке 3.2 приведен пример результата работы клеточного автомата.

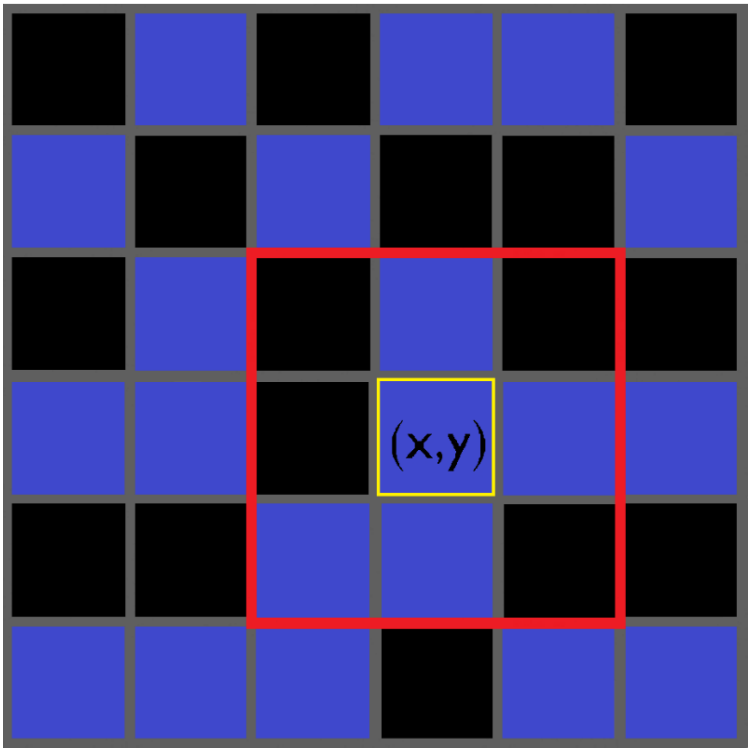

Рисунок 3.1. Пример итерации клеточного автомата

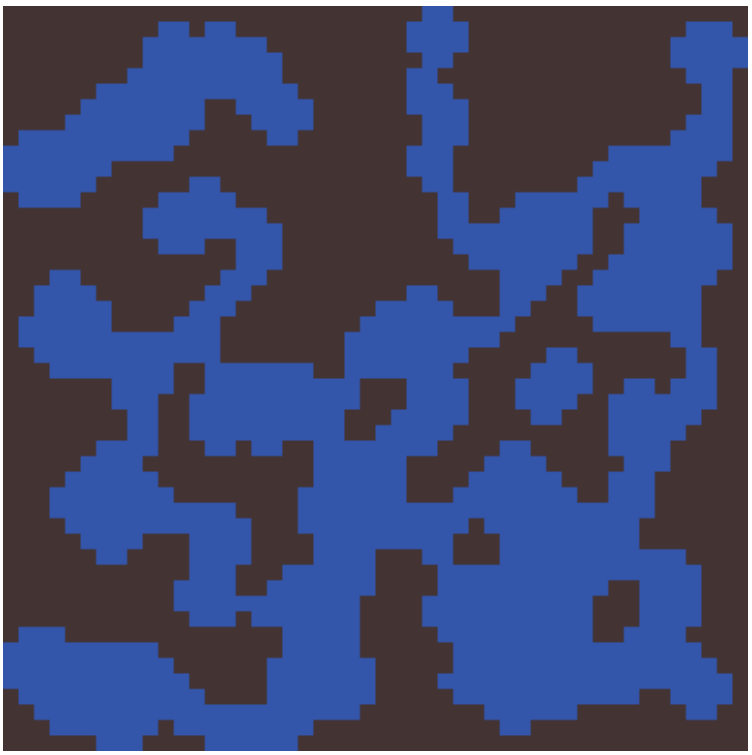

Рисунок 3.2. Пример результата работы клеточного автомата

Теперь рассмотрим применимость алгоритмов к нашей цели. Алгоритмы из пунктов 2) и 3) не подходят, так как в задаче требуется чёткая структура карты, а алгоритм из пункта 1) отсеивается из-за необходимости расположения пластов районов (линий районов) параллельно друг другу, но это условие

никак не мешает использовать метод генерации случайных комнат из 1) внутри прямоугольных частей, составляющих всю карту.

Таким образом, реализуем свой собственный алгоритм, вобравший в себя некоторые черты метода BSP-деревьев.

Разработка модели карты

Распишем данные условия.

Дано: карта состоит из пластов параллельных прямоугольных районов, в которых есть по несколько домов; у каждого дома есть дверь, от каждой двери идёт тропинка, ведущая к дороге, окружающей район.

Из начального условия становится ясно, что общая структура карты будет выглядеть как на рисунке 4.1, где прямоугольниками изображены районы города.

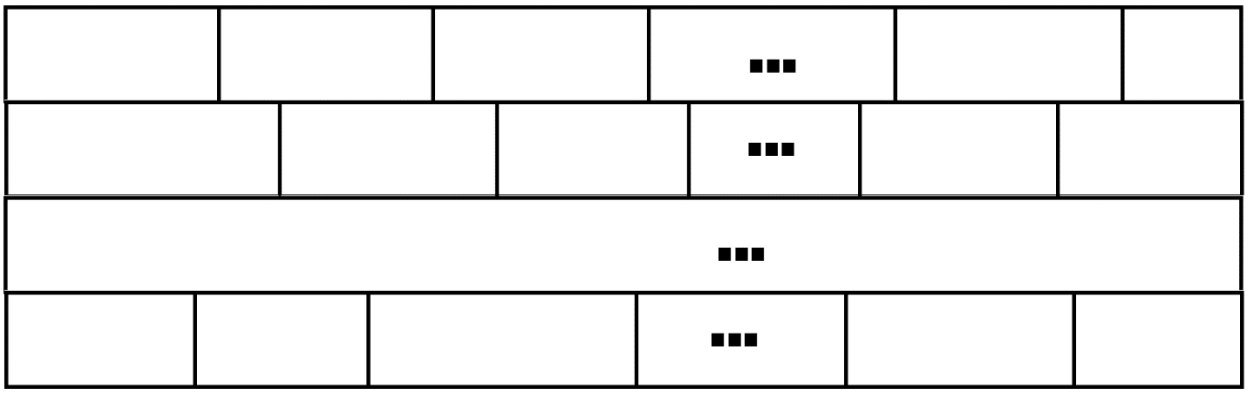

Рисунок 4.1. Общая структура карты города (версия 1)

Для удобства границы города обозначим отдельным символом (см. рисунок 4.2), например, «*».

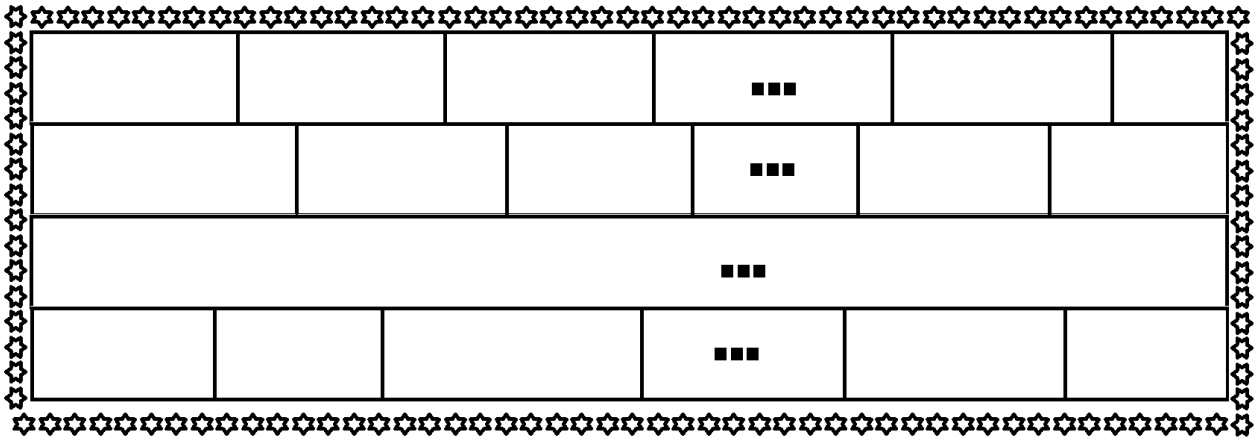

Рисунок 4.2. Общая структура карты города (версия 2)

Добавим дома (- прямоугольники меньших размеров нежели район) с обозначенными дверями и тропинками, ведущими к основной дороге (см. рисунок 4.3).

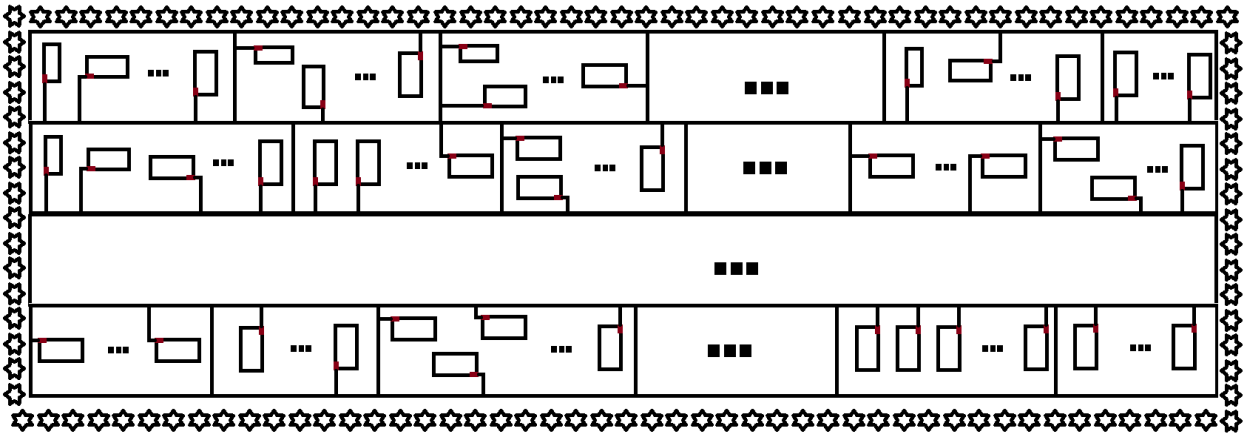

Рисунок 4.3. Общая структура карты города (версия 3)

Распланировав карту, перейдем к способу её хранения в вычислительном устройстве. Хранить карту будем в виде матрицы символов, где будут приняты следующие обозначения:

- «*» - границы города,
- «H» - дороги (и тропинки, и основные дороги),
- «D» - дверь дома,
- «1» - границы дома,
- «2» - внутренняя часть дома,
- «0» - пустое пространство на карте.

Реализация алгоритма генерации

Описание алгоритма

Для алгоритма будут нужны следующие параметры: ширина и высота карты (без учёта размеров дорог), количество районов в ширину, количество районов в высоту, процент искажения размеров района, размер дорог, процентная зависимость количества домов в районе от его размеров.

Разъясним некоторые параметры. Разделяя ширину карты на количество районов в ней, можно получить средний размер ширины района, и если генерировать карту при одинаково заданных её размерах, то будут получаться однообразные «сетки» районов, что не соответствует постановке задачи. Для внесения разнообразия вводится параметр «процент искажения размеров района», который меняет их размеры на $\pm(\text{процент искажения размеров района}) * (\text{средний размер района})$. Аналогично для высоты карты. Размер дорог - количество символов «H», используемое для обозначения дороги.

Очевидно, что размеры дома должны как-то соотносится с размерами района – для этого и вводится параметр «процентная зависимость количества домов в районе от его размеров», определяющий максимальный размер дома (как по ширине, так и по высоте).

Теперь перейдем к алгоритму:

I. *Получить все значения параметров; создать пустую карту, опираясь на введенные её размеры и размер дорог; обозначить границы карты.*
   Сгенерировать все размеры районов.
II. *Пока карта не заполнена полностью:*
   i. *Сгенерировать район с определенными параметрами: создать пустой район по заданным размерам, обвести его дорогой определённого размера, сгенерировать дома, а также тропинки от дверей до дороги.*
      Для генерации дома: получить размеры и начальную точку генерирования, сгенерировать координату для двери.
   ii. *Нарисовать район на карте.*

Подготовка к реализации

Алгоритм будет реализован на языке программирования C++. В качестве генератора случайных чисел используем генератор mersenne twister (более точно – mt19937), основанный на свойствах простых чисел.

Введем систему координат следующим образом: ось X направим вправо, ось Y направим вниз, тогда из особенности хранения массива в C++ элемент с координатами (x, y) будет находится в массиве карты с индексами $[y][x]$. Шириной (width) прямоугольника (- геометрической основы карты, района и

дома) будем называть длину горизонтальной линии данного прямоугольника, то есть она будет изменяться по оси X; высотой (height) – длину вертикальной линии, которая будет изменяться, соответственно, по оси Y. Также введем численные обозначения для направлений движения (сторон света): 1 – движение вверх (на север), 2 – движение влево (на запад), 3 – вниз (на юг), 4 – вправо (на восток). Все введенные обозначения можно увидеть на рисунке 5.

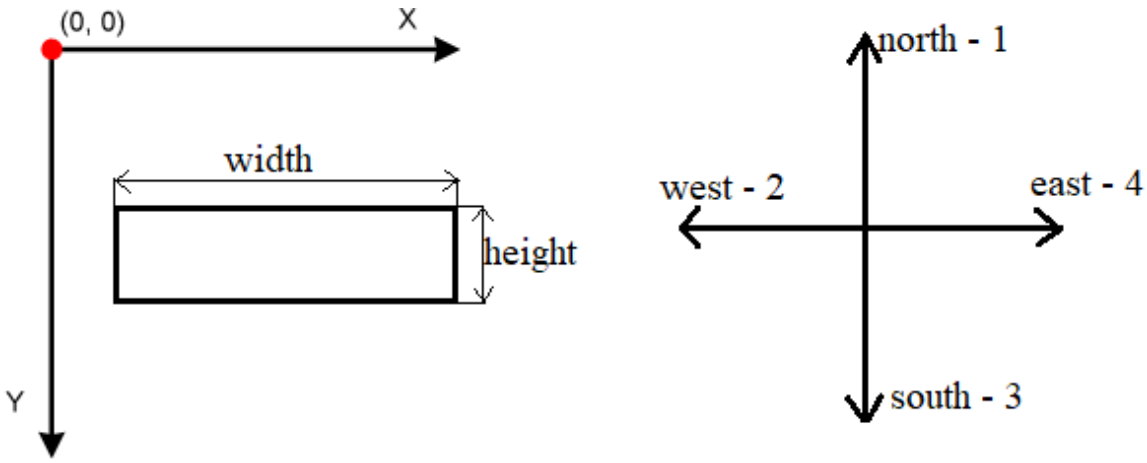

Рисунок 5. Введённая система обозначений

Из постановки задачи видно, что основные объекты для реализации – сама карта, районы и дома. Для элементов «карта» и «район» реализуем классы «Map» и «District» соответственно, а для элемента «дом» – структуру «House» (почему именно структура, станет понятнее позже), причем общая их взаимосвязь будет «матрешечная», то есть структура «дом» будет вложена в класс «район», а класс «район» будет вложен в класс «карта» (см. рисунок 6). Такая вложенность используется, так как генерация карты позиционируется как обособленная часть реализации игры, и её инструменты нигде больше в игре использоваться не будут (класс «Map» инкапсулирует класс «District», который в свою очередь инкапсулирует структуру «House»).

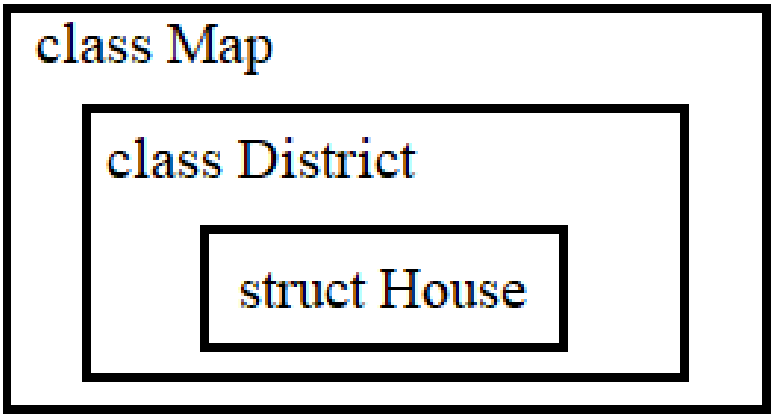

Рисунок 6. «Матрёшечная» структура

Реализация класса «Map»

Рассмотрим реализацию класса «Map». В «private» зоне класса опишем следующее:

1) std::mt19937 mapRNG - генератор случайных чисел класса;
2) const int m_width, m_height, m_wideCount, m_heightCount, m_roadSize - константные переменные, вводимые в конструкторе как параметры генерации, означающие соответственно ширину карты, высоту карты, количество районов в ширину, количество районов в высоту, ширину дорог в символах;
3) const double m_diffPercent, m_housePercent - константные переменные, вводимые в конструкторе как параметры генерации, означающие процент искажения размеров района относительно среднего и процентную зависимость количества домов в районе от его размеров;
4) int m_mapWidth, m_mapHeight - высчитываемые ширина и высота карты (учитывает ячейки под дороги и границы карты);
5) int curX = 1, curY = 1 - переменные для отрисовки карты, означающие текущее положение на карте;
6) std::vector<std::vector<char>> m_map - сама карта, реализуемая в виде вектора векторов элементов символьного типа (матрицы из элементов символьного типа);
7) std::vector<int> m_h - вектор размеров линий районов (вводится не двумерный массив, так как из условия районы должны объединятся в слои, параллельные друг другу);
8) std::vector<std::vector<int>> m_w - массив размеров районов в каждой линии;
9) class District – класс для создания районов.

Модификатор доступа «public» будут иметь конструктор и основные функции:

1. void shuffling(std::vector<int>&) – функция (метод класса) перемешивания элементов вектора. Функция работает следующим образом: создается копия начального вектора, а затем каждый его элемент заменяется случайным положительным элементом из копии, а использованный элемент копии обнуляется.
2. std::vector<int> splitting(int commomLen, int count) - функция разбиения отрезка длины commomLen на части в количестве count, возвращающая вектор длин отрезков разбиения. Рассмотрим суть работы функции:

   Пусть нужно разбить отрезок AB длины l на m частей, причём длины отрезков разбиения должны быть случайными.

   Пусть $\tilde{l} = int(l/m)$ – средняя целочисленная длина разбиения отрезка AB, $\Delta = \tilde{l} \cdot \alpha$, где α – коэффициент возможного отклонения (аналог m_diffPercent).

   Возьмем отрезок $[\tilde{l} - \Delta, \tilde{l} + \Delta]$, на котором и будем генерировать m случайных значений длин частей отрезка AB, но здесь же сразу возникает проблема: сумма длин всех частей может превышать l.

Решение: генерировать не все значения случайно, а только $int(m/2)$, другие $int(m/2)$ получать из соотношения $2 \cdot \tilde{l} - \bar{l}$, где $\bar{l} \in [\tilde{l} - \Delta, \tilde{l} + \Delta]$ – случайное полученное значение. При чётном m получим все нужные значения, если же m – нечётное, то длину оставшегося одного куска отрезка AB можно принять за $\tilde{l}$.

Все вышеперечисленное верно, если $\tilde{l} = l/m$; если же последнее неверно, то оставшиеся кусочки из $l - \tilde{l} \cdot m$ можно случайно распределить между всеми отрезками, образующими большой отрезок AB.

Для большей произвольности разбиения полученные отрезки разбиения дополнительно перемешиваются в функции из пункта 1.

3. void printMap(std::string name) const – функция записи карты (-двумерного массива с элементами символьного типа) в файл с именем «name».
4. void draw(std::vector<std::vector<char>> &arr) – функция вставки двумерного массива arr в массив карты, начиная с координат curX, curY (см. пункт 5) из «private» зоны класса). После внедрения массива arr, координаты curX, curY пересчитываются:

curX = curX + arr[0].size() - m_roadSize,

где arr[0].size() – ширина двумерного массива, m_roadSize – ширина дороги. Перемещение можно рассмотреть на рисунке 7.1.

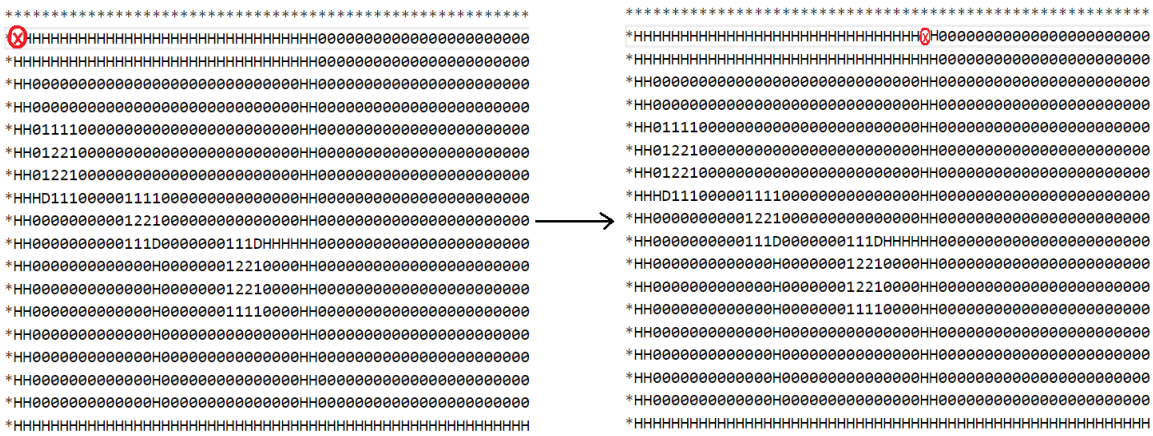

Рисунок 7.1. Смещение переменной curX

Возникает вопрос: зачем вычитать из curX ширину дороги? Ответ: массивы, поступающие на зарисовку специфичны, а именно, они окаймлены дорогами одинаковой ширины, и, чтобы не дублировать дороги, а делать их общими, они накладываются друг на друга. Можно заметить, что координата curY при таком перемещении не изменяется.

Далее в функции проверяется попадание переменной curX на границу карты:

m_map[curY][curX + m_roadSize] == '*'.

Если утверждение неверно, то функция заканчивает работу, иначе пересчитываются обе координаты:

curX = 1,

curY = curY + arr.size() - m_roadSize,

то есть происходит смещение в начало по оси X (позицию нулевого столбца занимает граница карты, поэтому curX = 1) и спуск вниз по оси Y на длину высоты района – она одинаковая для всей линии по условию (см. рисунок 7.2):

Рисунок 7.2. Смещение переменной curX и curY после подхода к границе

5. std::vector<std::vector<char>> getMap() – функция, возвращающая копию матрицы (вектора векторов с элементами символьного типа) карты из класса.
6. void mapInit() - функция, инициализирующая двумерный массив карты: рассчитываются реальные размеры карты (пункт 4) из «private» зоны класса), заносятся границы в виде символов «*» и запускается механизм отрисовки карты (создаются элементы класса «District» и с помощью функции draw(arr) из пункта 4. вставляются в карту).
7. Map(const int, const int, const int, const int, const double, const int, const double) - конструктор класса, в котором инициализируются переменные из пунктов 2) и 3) «private» зоны класса, устанавливается «зерно» для генератора случайных чисел 1), высчитываются размеры районов, а затем вызывается метод mapInit() из пункта 6. После добавления всех районов, матрица карты заносится в текстовый файл с названием «Map.txt» с помощью функции из пункта 3. для наглядности карты.

Реализация структуры «House»

Теперь рассмотрим реализацию структуры «House». В качестве способа хранения данных выбрана «структура», так как по умолчанию в ней любой член доступен из любого места в коде (автоматически стоит модификатор доступа «public»), а также это добавляет разнообразие в реализацию алгоритма. Опишем:

1. std::mt19937 houseRNG - генератор случайных чисел класса;
2. std::pair<const int, const int> xy – структура, хранящая в себе координаты x и y верхней левой вершины дома, имеющего вид прямоугольника, в отдельной матрице района (то есть не на общей карте, а в отдельном двумерном массиве класса «District»).
3. std::pair<int, int> doorXY - структура, хранящая в себе координаты x и y двери дома в отдельной матрице района. В отличие от координат левой верхней вершины имеет не константный тип, так как подсчитывается во время работы программы, а не инициализируется сразу, получая значения извне.
4. const int w, h – ширина и высота дома (прямоугольника);
5. bool intersect(const House &) const – метод структуры, проверяющий пересечение двух объектов типа «House». Проверяются два следующих типа условия для каждой координаты:
    a. начальная координата (x или y) первого дома (прямоугольника) в системе счисления (см. рисунок 5) больше начальной координаты второго дома (x или y) + ширина (высота) второго дома + 1 (добавление «1» гарантирует, что дома будут находится на расстоянии как минимум в 1 ячейку матрицы друг от друга);
    b. начальная координата второго дома в системе счисления больше начальной координаты первого дома + ширина (высота) первого дома + 1;

    Данные четыре условия гарантируют непересечение домов. Функция возвращает false, если дома не пересекаются, true, если пересекаются.
6. void doorMarking() – метод, случайного определения расположения двери на границах дома. Выбирается случайная сторона света, а затем на отрезке $[x, x + w]$ или $[y, y + h]$ генерируется число, которое и будет одной из координат. Вторая координата высчитывается из выбора стороны света (соответственно выбора стороны прямоугольника).
7. House(const int, const int, const int, const int) – конструктор структуры. Инициализирует переменные из пунктов 2. и 4., устанавливает «зерно» для генератора случайных чисел 1., запускает функцию генерации координат двери из пункта 6.

Реализация класса «District»

Модификатор доступ «private» будут иметь следующие члены класса:

1) std::mt19937 districtRNG - генератор случайных чисел класса;
2) const int m_dWidth, m_dHeight, m_roadSize, m_houseCount, m_a, m_b – константные переменные, обозначающие соответственно ширину, высоту района (с учётом ширины дорог), ширину дороги, количество домов в районе, минимальный размер дома, максимальный размер дома.
3) std::vector<std::vector<char>> m_district – карта района, реализуемая в виде вектора векторов элементов символьного типа;
4) struct House – структура «House» (рассмотрена выше);
5) std::vector<House> m_houses – вектор объектов типа структуры «House» для хранения всех домов, которые есть в районе.

Модификатор доступ «public» будут иметь следующие методы класса:

1. void init() - метод инициализации массива m_district: устанавливает его размер, автоматически заполняет его символами «0», добавляет дороги по краям со всех четырех сторон в размере m_roadSize.
2. void generate() – метод генерации домов внутри района. По введённому количеству домов m_houseCount делается 1000 попыток генерации дома и встройки его в район до первого успешного случая. Случай считается успешным, если сгенерированный дом (прямоугольник) не пересекается с другими домами (прямоугольниками) в районе (функция проверки пересечения – метод структуры «House» intersect(const House &)) и встраивается в его массив. Если по истечении 1000 попыток дом не устанавливается, то ничего не происходит и количество домов уменьшается на один. Успешно сгенерированные дома добавляются в массив m_houses.

   В особенность генерации также входит то, что дома находятся как минимум на расстоянии одной клетки от дороги и минимальный размер дома равен 3 × 3 элементарных ячеек.

   После генерации всех домов, они переносятся на карту, где выделяются дверь («D»), границы дома («1») и внутренняя часть («2»). Пример встроенного дома можно увидеть на рисунке 8.

```
*HHHHHHHHHHHHHHHHHHHH
*HHHHHHHHHHHHHHHHHHHH
*HH0000000000000000000
*HH0000000000000000000
*HH0000000000000000000
*HH00000001111D00000000
*HH00000001222100000000
*HH00000001111100000000
*HH0000000000000000000
*HH0000000000000000000
```

Рисунок 8. Пример встроенного дома

3. std::vector<std::vector<char>> getDistrict() const – функция, возвращающая копию матрицы района из класса;
4. void print(std::string name) const – функция записи района (- двумерного массива с элементами символьного типа) в файл с именем «name».
5. std::vector<std::pair<int, int>> dfs(const House &) const – функция поиска в ширину графа с восстановлением пути. В качестве графа берется двумерный массив m_district, за вершину – элементарная его ячейка со значениями «0» или «H». Будем считать, что две вершины графа связаны, если они находятся или сверху, или снизу, или справа, или слева относительно друг друга. Поиск останавливается, если обнаружена вершина со значением «H» (даже если это не край района, а другая тропинка – это случай объединения тропинок). Затем идет восстановление пути.

   Функция возвращает вектор с координатами ячеек, составляющих тропинку от двери дома до основной дороги
6. void buildRoads() - функция построения дорог для всех домов района. Для каждого дома запускается dfs(const House &), ищущий путь, который затем наносится на карту.
7. District(const int, const int, const int, const int, const int, const int) - конструктор класса, в котором инициализируются переменные из пункта 2), устанавливается «зерно» для генератора случайных чисел 1). Далее вызываются методы из пунктов 1. и 2.

Дополнительное задание

В качестве дополнительного задания нужно визуализировать реализации модели карты при одинаковых параметрах.

Введём следующие параметры: height = 45, width = 60, wideCount = 3, heightCount = 3, diffPercent = 0.25, roadSize = 2, housePercent = 0.33. Полученные реализации карты можно увидеть на рисунках 9.1-9.3.

Как видно из всех трёх реализаций модели, общая структура карты сохраняется, но изменяются размеры районов, количество и размеры домов, положения их дверей, что и требовалось по заданию.

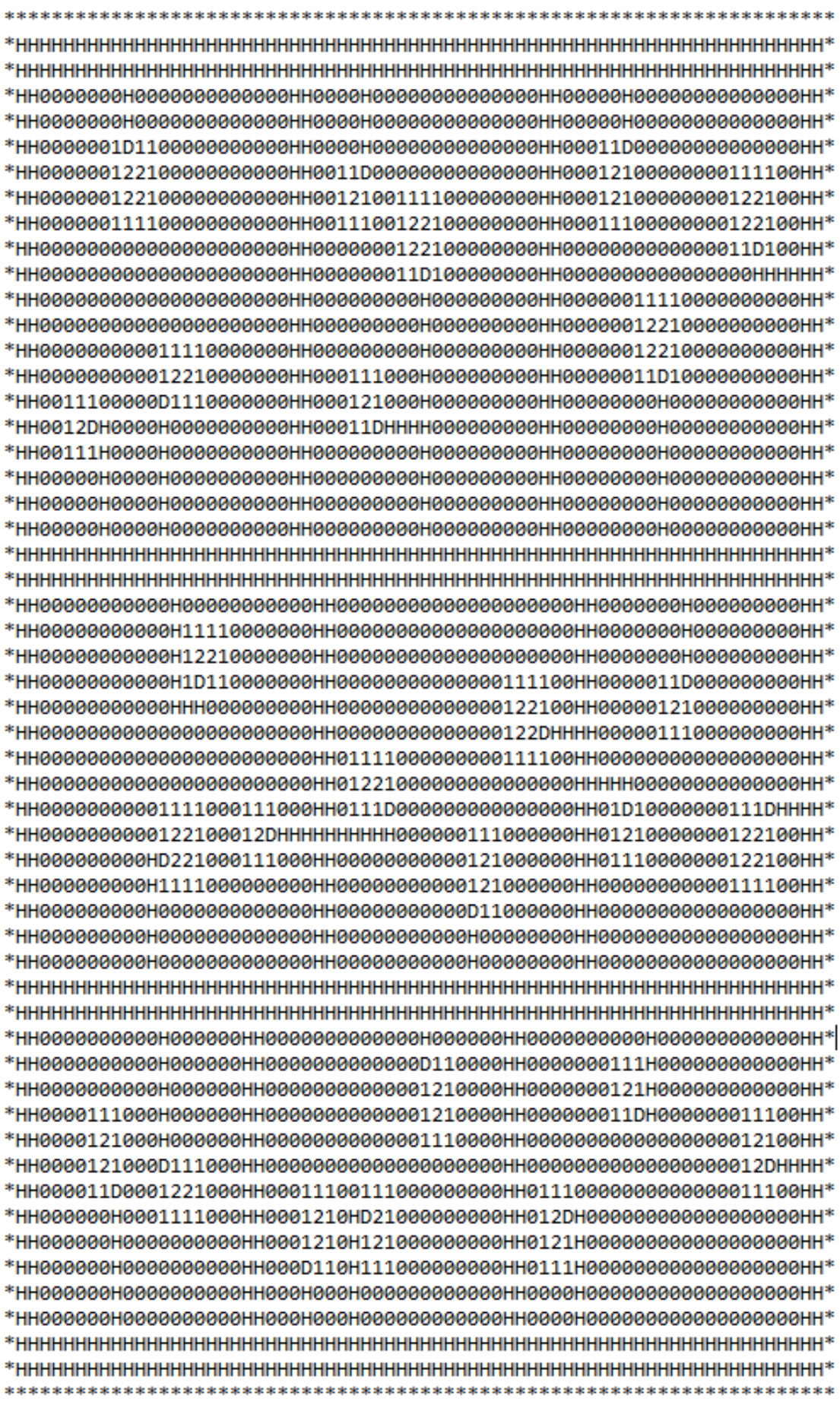

Рисунок 9.1. Первая реализация модели карты

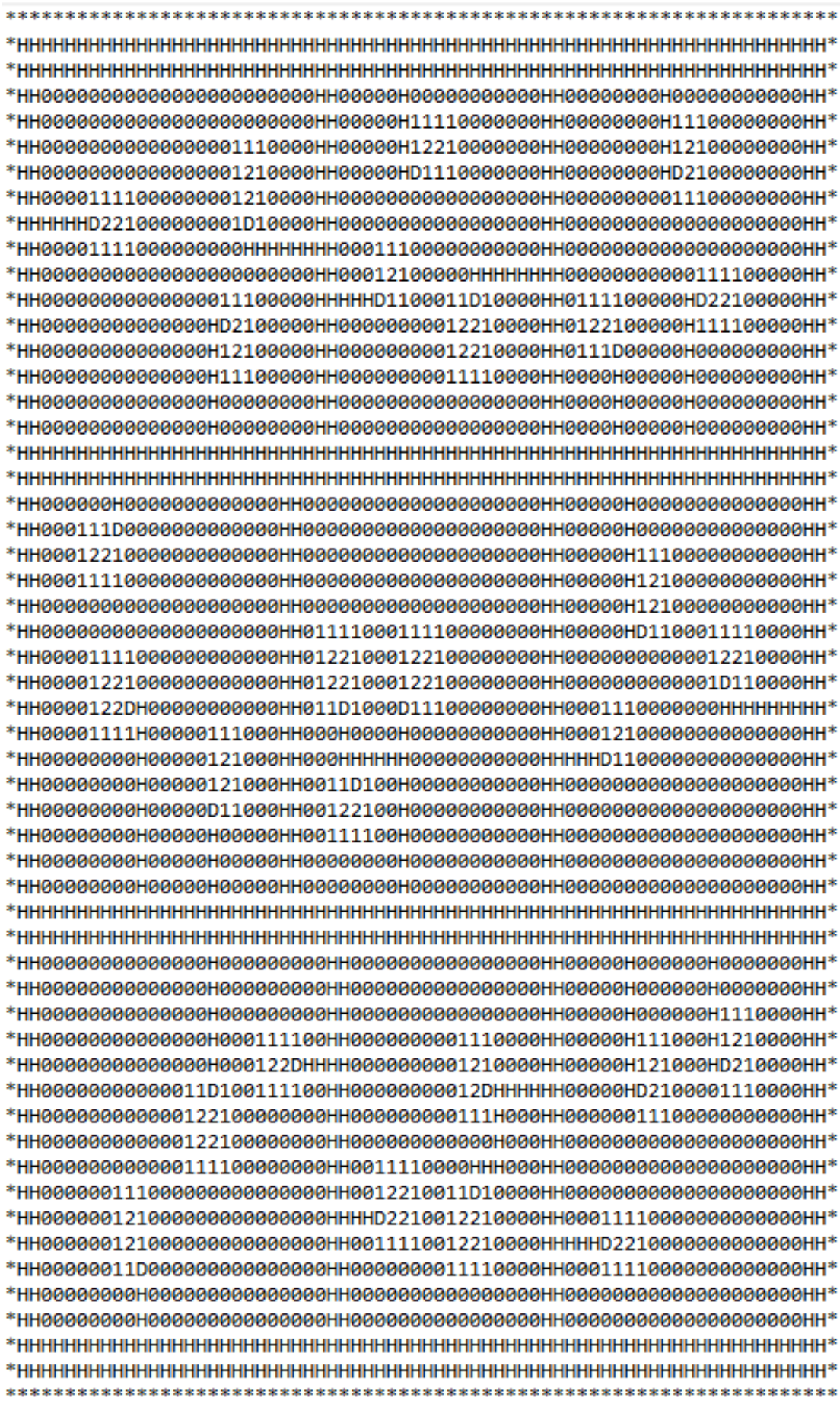

Рисунок 9.2. Вторая реализация модели карты

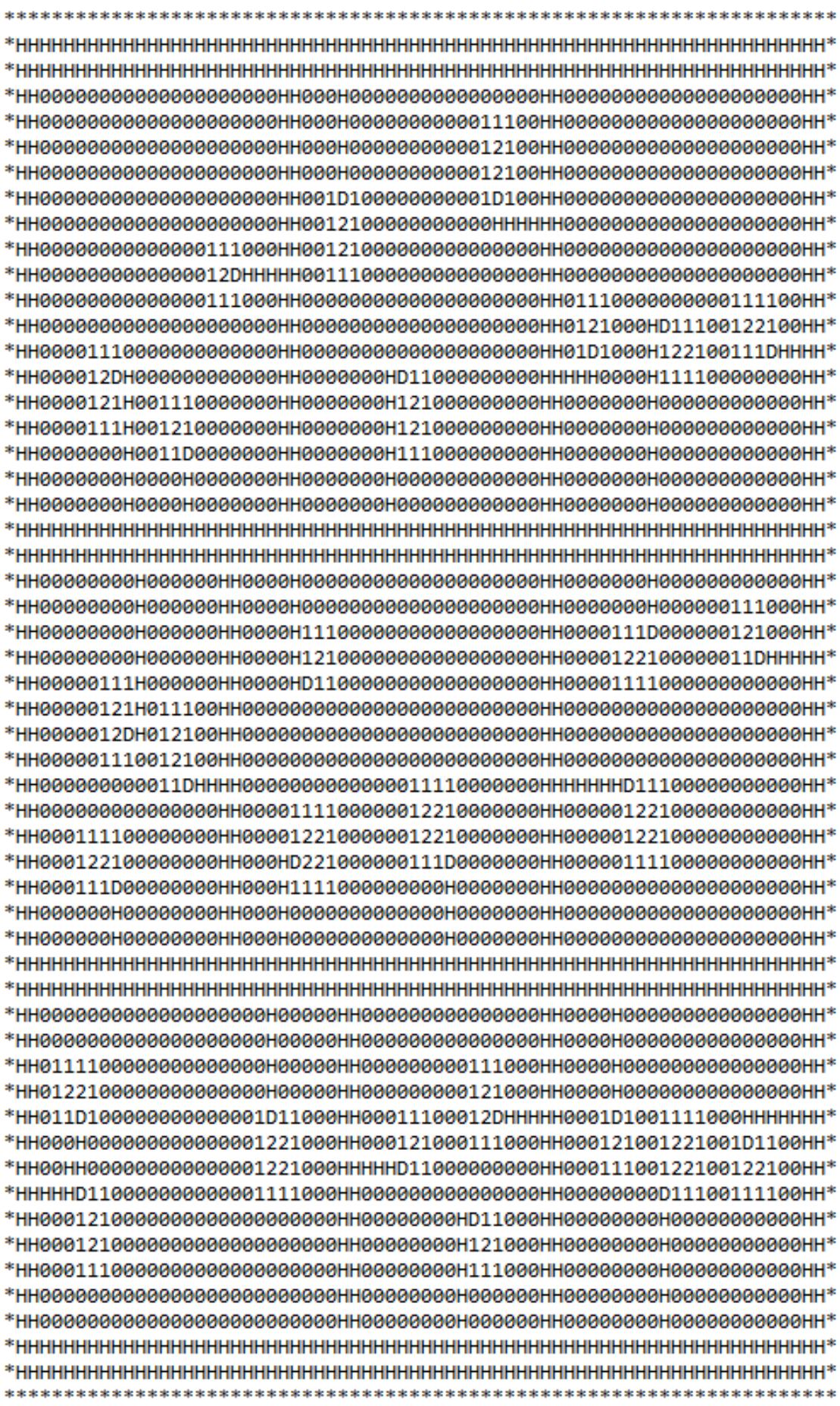

Рисунок 9.3. Третья реализация модели карты

Вывод

Во время работы над практикой я изучил общие принципы процедурной генерации игровых карт, разработал модель игровой карты, зависящей от входных параметров, реализовал алгоритм её процедурной генерации и, как дополнительное задание, визуализировал различные реализации карты при одинаковых входных параметрах.

Впервые столкнувшись с действительно большим проектом, разработкой которого я занимался в течение 4 месяцев, я получил много опыта, научился формулировать цели, расставлять приоритеты и изучать большое количество материалов.

Список литературы

1. // Процедурная генерация подземелий в roguelike – PatientZero, 14 мая 2018 год
   https://habr.com/ru/post/354826/
2. // How to Use BSP Trees to Generate Game Maps – Timothy Hely, 1 ноября 2013 год
   https://gamedevelopment.tutsplus.com/tutorials/how-to-use-bsp-trees-to-generate-game-maps--gamedev-12268
3. // Random Walk Cave Generation
   http://www.roguebasin.com/index.php?title=Random_Walk_Cave_Generation
4. // Cellular Automata Method for Generating Random Cave Like Levels
   http://www.roguebasin.com/index.php?title=Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels
5. // Generate Random Cave Levels Using Cellular Automata - Michael Cook, 23 июля 2013 год
   https://gamedevelopment.tutsplus.com/tutorials/generate-random-cave-levels-using-cellular-automata--gamedev-9664
6. // Процедурная генерация карты (часть 1) - Артем Гуревич, 11 ноября 2013 года (Обновление: 13 ноября 2013 года)
   https://gamedev.ru/code/articles/levelgen1

Приложение

Код алгоритма находится в 4 файлах.

В include.hpp:

```
#include <algorithm>
#include <chrono>
#include <fstream>
#include <iostream>
#include <random>
#include <string>
#include <vector>
#include <queue>
#include <utility>
```

В map.hpp:

```
#pragma once

#include "include.hpp"

class Map {
private:
    std::mt19937 mapRNG;
    const int m_width, m_height, m_wideCount, m_heightCount, m_roadSize;
    const double m_diffPercent, m_housePercent;
    int m_mapWidth, m_mapHeight;
    int curX = 1, curY = 1;
    std::vector<std::vector<char>> m_map;
    std::vector<int> m_h;
    std::vector<std::vector<int>> m_w;

    class District {
    private:
        std::mt19937 districtRNG;
        const int m_dWidth, m_dHeight, m_roadSize, m_houseCount, m_a, m_b;
        std::vector<std::vector<char>> m_district;
        struct House {
            std::mt19937 houseRNG;
            std::pair<const int, const int> xy;
            std::pair<int, int> doorXY;
            const int w, h;
            House(const int, const int, const int, const int);
            bool intersect(const House &) const;
            void doorMarking();
        };
        std::vector<House> m_houses;
    public:
        District(const int, const int, const int, const int, const int, const int
);
```

```
        void init();
        void generate();
        void buildRoads();
        std::vector<std::pair<int, int>> dfs(const House &) const;
        std::vector<std::vector<char>> getDistrict() const;
        void print(std::string) const;
    };
public:
    Map(const int, const int, const int, const int, const double, const int, cons
t double);
    std::vector<int> splitting(int, int);
    void shuffling(std::vector<int>&);
    void mapInit();
    void draw(std::vector<std::vector<char>> &);
    std::vector<std::vector<char>> getMap();
    void printMap(std::string) const;
};
```

B map.cpp:

```
#include "map.hpp"

Map::District::House::House(const int curX, const int curY, const int curW, const
 int curH) :
xy(curX, curY), w(curW), h(curH) {
        houseRNG.seed(std::chrono::steady_clock::now().time_since_epoch().count()
);
        doorMarking();
}

bool Map::District::House::intersect(const House &r) const{
    return !(r.xy.first > (xy.first + w + 1) || xy.first > (r.xy.first + r.w + 1)

        || r.xy.second > (xy.second + h + 1) || xy.second > (r.xy.second + r.h +
1));
}

void Map::District::House::doorMarking() {

    int worldSide = 1 + houseRNG() % 4;

    switch (worldSide)
    {
    case 1:
        doorXY = std::make_pair(xy.first + houseRNG() % w, xy.second);
        break;

    case 2:
        doorXY = std::make_pair(xy.first, xy.second + houseRNG() % h);
        break;
```

```
    case 3:
        doorXY = std::make_pair(xy.first + houseRNG() % w, xy.second + h - 1);
        break;

    case 4:
        doorXY = std::make_pair(xy.first + w - 1, xy.second + houseRNG() % h);
        break;

    default:
        std::cout << "Error: wrong world side!" << std::endl;
        break;
    }
}

Map::District::District(const int curHeight, const int curWidth, const int curRoa
dSize, const int curHouseCount,
const int curA, const int curB) : m_dHeight(curHeight + 2 * curRoadSize), m_dWidt
h(curWidth + 2 * curRoadSize),
m_roadSize(curRoadSize), m_houseCount(curHouseCount), m_a(curA), m_b(std::max(cur
B, m_a)) {

    districtRNG.seed(std::chrono::steady_clock::now().time_since_epoch().count())
;
    init();
    generate();
}

void Map::District::init() {
    m_district = std::vector<std::vector<char>>(m_dHeight, std::vector<char>(m_dW
idth, '0'));
    for (size_t i = 0; i < m_dHeight; ++i)
        for (size_t j = 0; j < m_dWidth; ++j) {
            if (i < m_roadSize || j < m_roadSize || i >= m_dHeight - m_roadSize |
| j >= m_dWidth - m_roadSize)
                m_district[i][j] = 'H';
        }
}

void Map::District::generate() {
    for (size_t i = 0; i < m_houseCount; ++i)
        for (size_t j = 0; j < 1000; ++j) {
            const int w = m_a + districtRNG() % (m_b - m_a + 1), h = m_a + distri
ctRNG() % (m_b - m_a + 1);
            const House house{
                static_cast<const int>(3 + districtRNG() % (m_dWidth - w - 6)), /
/ x
                static_cast<const int>(3 + districtRNG() % (m_dHeight - h - 6)),
// y
                w,
```

```
                h
            };

            bool intersection = false;
            for(size_t k = 0; k < m_houses.size(); ++k) {
                intersection = house.intersect(m_houses[k]);
                if (intersection) break;
            }

            if (!intersection) {
                m_houses.push_back(house);
                break;
            }
        }

    for (const House &house : m_houses) {
        for (size_t y = 0; y < house.h; ++y)
            for (size_t x = 0; x < house.w; ++x)
                if (x != 0 && x != (house.w - 1) && y != 0 && y != (house.h - 1))
                    m_district[house.xy.second + y][house.xy.first + x] = '2';
                else
                    m_district[house.xy.second + y][house.xy.first + x] = '1';
        m_district[house.doorXY.second][house.doorXY.first] = 'D';
    }

    buildRoads();
}

void Map::District::buildRoads() {
    std::vector<std::pair<int, int>> path;
    for (const House &house : m_houses) {
        path = dfs(house);
        for (size_t i = 0; i < path.size() - 1; ++i) {
            m_district[path[i].second][path[i].first] = 'H';
        }
    }
}

std::vector<std::pair<int, int>> Map::District::dfs(const House &house) const {
    std::queue<std::pair<int, int>> q;
    std::vector<std::vector<int>> cameFrom(m_district.size(), std::vector<int>(m_
district[0].size(), 0));
    for (size_t i = 0; i < m_district.size(); ++i)
        for (size_t j = 0; j < m_district[0].size(); ++j) {
            if (m_district[i][j] != '0') cameFrom[i][j] = -1;
            if (m_district[i][j] == 'H') cameFrom[i][j] = 8;
        }
    q.push(house.doorXY);

    std::pair<int, int> temp;
```

```
    while(!q.empty()) {
        temp = q.front();
        q.pop();

        if (m_district[temp.second][temp.first] == 'H') break;

        if (cameFrom[temp.second + 1][temp.first] == 0 || cameFrom[temp.second +
1][temp.first] == 8) {
            q.push({temp.first, temp.second + 1});
            cameFrom[temp.second + 1][temp.first] = 1;
        }
        if (cameFrom[temp.second][temp.first + 1] == 0 || cameFrom[temp.second][t
emp.first + 1] == 8) {
            q.push({temp.first + 1, temp.second});
            cameFrom[temp.second][temp.first + 1] = 2;
        }
        if (cameFrom[temp.second - 1][temp.first] == 0 || cameFrom[temp.second -
1][temp.first] == 8) {
            q.push({temp.first, temp.second - 1});
            cameFrom[temp.second - 1][temp.first] = 3;
        }
        if (cameFrom[temp.second][temp.first - 1] == 0 || cameFrom[temp.second][t
emp.first - 1] == 8) {
            q.push({temp.first - 1, temp.second});
            cameFrom[temp.second][temp.first - 1] = 4;
        }
    }

    std::vector<std::pair<int, int>> path;
    while(temp != house.doorXY) {
        switch (cameFrom[temp.second][temp.first])
        {
        case 1:
            temp = {temp.first, temp.second - 1};
            path.push_back(temp);
            break;
        case 2:
            temp = {temp.first - 1, temp.second};
            path.push_back(temp);
            break;
        case 3:
            temp = {temp.first, temp.second + 1};
            path.push_back(temp);
            break;
        case 4:
            temp = {temp.first + 1, temp.second};
            path.push_back(temp);
            break;
        default:
            std::cout << "Error: wrong side!" << std::endl;
```

```
            break;
        }
    }
    return path;
}

std::vector<std::vector<char>> Map::District::getDistrict() const {
    return m_district;
}

void Map::District::print(std::string name) const {
    std::ofstream fout(name);
    for(size_t i = 0; i < m_district.size(); ++i) {
        for(size_t j = 0; j < m_district[i].size(); ++j)
            fout << m_district[i][j];// << '\t';
        fout << "\n";
    }
    fout.close();
}

Map::Map(const int height, const int width, const int wideCount, const int height
Count, const double diffPercent,
const int roadSize, const double housePercent) : m_height(height), m_width(width)
, m_wideCount(wideCount),
m_heightCount(heightCount), m_diffPercent(diffPercent), m_roadSize(roadSize), m_h
ousePercent(housePercent) {

    mapRNG.seed(std::chrono::steady_clock::now().time_since_epoch().count());
    m_h = splitting(m_height, m_heightCount);
    for (size_t i = 0; i < m_heightCount; ++i)
        m_w.push_back(splitting(m_width, m_wideCount));

    mapInit();
    printMap("Map.txt");
}

std::vector<int> Map::splitting(int commomLen, int count) {
    int avgLen = commomLen / count;
    const int delta = static_cast<const int>(avgLen * m_diffPercent);
    int minLen = avgLen - delta, maxLen = avgLen + delta;
    std::vector<int> lens(count);

    if (! (count % 2))
        for (size_t i = 0; i < count / 2; ++i) {
            lens[i] = minLen + mapRNG() % (maxLen - minLen + 1);
            lens[i + count / 2] = 2 * avgLen - lens[i];
        }
    else {
        for (size_t i = 0; i < count / 2; ++i) {
            lens[i] = minLen + mapRNG() % (maxLen - minLen + 1);
```

```
            lens[i + count / 2 + 1] = 2 * avgLen - lens[i];
        }
        lens[count / 2] = avgLen;
    }

    for(size_t i = 0; i < commomLen - avgLen * count; ++i)
        lens[mapRNG() % count] += 1;

    shuffling(lens);
    return lens;
}

void Map::shuffling(std::vector<int> &arr) {
    std::vector<int> temp = arr;
    int j = 0;
    for(size_t i = 0; i < arr.size(); ++i){
        while(!temp[j])
          j = mapRNG() % arr.size();
        arr[i] = temp[j];
        temp[j] = 0;
    }
}

void Map::mapInit() {
    m_mapWidth = m_width + (m_wideCount + 1) * m_roadSize + 2;
    m_mapHeight = m_height + (m_heightCount + 1) * m_roadSize + 2;
    m_map = std::vector<std::vector<char>>(m_mapHeight, std::vector<char>(m_mapWi
dth, '0'));

    for (size_t i = 0; i < m_mapWidth; ++i) { m_map[0][i] = '*'; m_map[m_mapHeigh
t - 1][i] = '*'; }
    for (size_t i = 0; i < m_mapHeight; ++i) { m_map[i][0] = '*'; m_map[i][m_mapW
idth - 1] = '*'; }

    std::vector<std::vector<char>> tempArr, tempArr2, tempArr3, tempArr4, tempArr
5;
    for (size_t i = 0; i < m_w.size(); ++i)
        for (size_t j = 0; j < m_w[0].size(); ++j) {
            District temp(
                m_h[i], m_w[i][j], m_roadSize, 1 / m_housePercent, 3,
                std::min(m_width / m_wideCount, m_height / m_heightCount)*m_house
Percent
            );
            tempArr = temp.getDistrict();
            draw(tempArr);
        }
}

void Map::draw(std::vector<std::vector<char>> &arr) {
    for (size_t i = 0; i < arr.size(); ++i)
```

```
        for(size_t j = 0; j < arr[0].size(); ++j)
            m_map[curY + i][curX + j] = arr[i][j];

    curX = curX + arr[0].size() - m_roadSize;
    if (m_map[curY][curX + m_roadSize] == '*') {
        curX = 1;
        curY = curY + arr.size() - m_roadSize;
    }
}

std::vector<std::vector<char>> Map::getMap() {
    return m_map;
}

void Map::printMap(std::string name) const {
    std::ofstream fout(name);
    for(size_t i = 0; i < m_map.size(); ++i) {
        for(size_t j = 0; j < m_map[i].size(); ++j)
            fout << m_map[i][j];
        fout << "\n";
    }
    fout << "\n";
    fout.close();
}
```

B main.cpp:

```
#include "map.hpp"

int main() {
    Map Test(45, 60, 3, 3, 0.25, 2, 0.33);
}
```