Оглавление

[Введение 3](#_Toc122504859)

[Глава 1. Алгоритмы процедурной генерации 3](#_Toc122504860)

[1.1. Алгоритм, основанный на BSP-дереве 3](#_Toc122504861)

[1.2. Алгоритм, основанный на картах шумов 5](#_Toc122504862)

[1.3. Wave Function Collapse (Коллапс волновой функции) 6](#_Toc122504863)

[Сравнение 9](#_Toc122504864)

[Глава 2. Реализация WFC 9](#_Toc122504865)

[Заключение 15](#_Toc122504866)

[Список литературы 16](#_Toc122504867)

[Приложения 17](#_Toc122504868)

[Приложение 1. 17](#_Toc122504869)

[Приложение 2. 33](#_Toc122504870)

# Введение

На сегодняшний день существует проблема однообразности в игровой индустрии. Разработчики видеоигр вынуждены либо тратить годы на создание хороших текстур, рельефов, планов городов, домов и прочего вручную, либо выпускать игры с лесом из однообразных деревьев, расставленные на определенном расстоянии друг от друга, или с городами, которые копируют сами себя. Поэтому разработчики начали задумываться над тем, чтобы использовать алгоритмы процедурной генерации, и чтобы их результаты были похожи на естественные распределения в природе или на рукотворные объекты, созданные человеком. В этой работе рассмотрены три наиболее используемых алгоритма.

# Глава 1. Алгоритмы процедурной генерации

### Алгоритм, основанный на BSP-дереве

Что такое BSP-дерево[[1]](#footnote-1)?

Чтобы понять, что из себя представляет двоичное разбиение пространства, рассмотрим Рисунок 1.1.1. На нем показано двоичное разбиение множества объектов на плоскости и дерево, которое этому разбиению соответствует. В двумерном случае BSP-дерево строится с помощью рекурсивного разбиения плоскости прямыми. В данном примере это происходит так: сначала проводим прямую l1, разбивая полуплоскость выше l1 прямой l2, а ниже — прямой l3 и так далее. Прямые разбивают на части не только плоскость, но и объекты, расположенные на ней. Разбиение продолжается до тех пор, пока внутри каждой грани плоскости окажется не более одного фрагмента объекта. Этот процесс можно представить с помощью двоичного дерева. Каждый лист дерева соответствует грани разбиения, в нем хранится фрагмент объекта, находящийся внутри этой грани. Каждый узел дерева соответствует разбивающей прямой, которая хранится в этом узле.

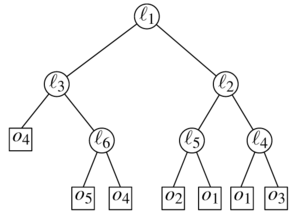
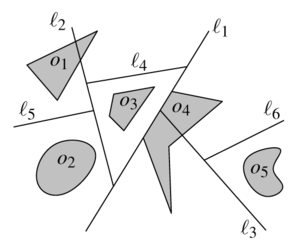


Рисунок 1.1.1. Пример разбиения и простроенного BSP-дерева.

Зачем использовать BSP-деревья для генерирования карт?

Существует множество разных способов генерирования случайной карты. Можно написать простую логику для создания прямоугольников случайного размера в случайных местах, но в результате получатся карты, заполненные пересекающимися, скученными или странно расположенными комнатами. Кроме того, немного сложнее соединить все комнаты друг с другом и сделать так, чтобы не осталось изолированных комнат. Благодаря BSP-дереву можно гарантировать создание более равномерно распределённых комнат и обеспечить их соединение.

Алгоритм

* Построение BSP-дерева в соответствии с заданным разбиением. В большинстве случаев для простоты реализации используют разбиение на прямоугольные участки.
* Создание комнат в каждом листе дерева. Комнаты могут занимать как все пространство листа, так и часть.
* Соединение комнат коридорами (если в этом есть необходимость). Поскольку уже есть встроенные связи в дереве, ими можно воспользоваться.

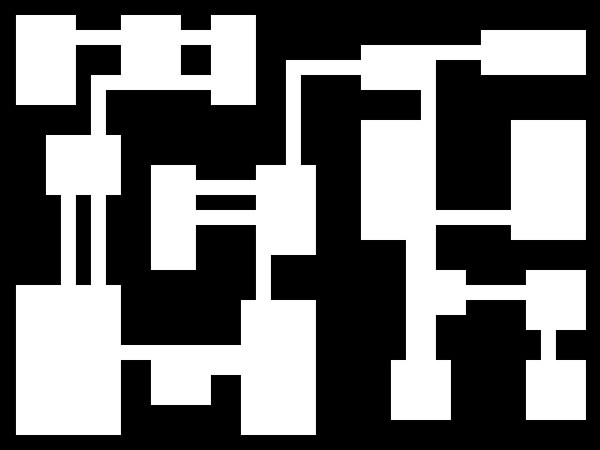


Рисунок 1.1.2 Пример результата работы данного алгоритма.

Решаемые задачи

Алгоритм генерации основанный на BSP-дереве подходит для создания двумерных карт подземелий, планировки комнат зданий и т.п.

Свойства

Сложность построения BSP-дерева , ожидаемое число элементов дерева, сгенерированных алгоритмом есть . Алгоритм гарантирует равномерное распределение объектов по заданному пространству, что может учитываться для размещения стратегических объектов на уровнях.

### Алгоритм, основанный на картах шумов

Что такое шум?

Для процедурной генерации используют численный шум. Численный шум — это тип шума, полученный путем генерации псевдослучайных чисел. Обычно используемый в качестве процедурного текстурного примитива в компьютерной графике. Обычно используется градиентной шум, примерами которого являются шум Перлина и симплексный шум. Этот метод состоит в создании решетки точек, которым присваиваются случайные значения. Затем функция шума возвращает интерполированное число, основанное на значениях окружающих точек решетки. Факт взаимосвязанности этих значений друг с другом – ключевой аспект генерации миров, поскольку логично ожидать, что участки, расположенные рядом, будут выглядеть похоже.

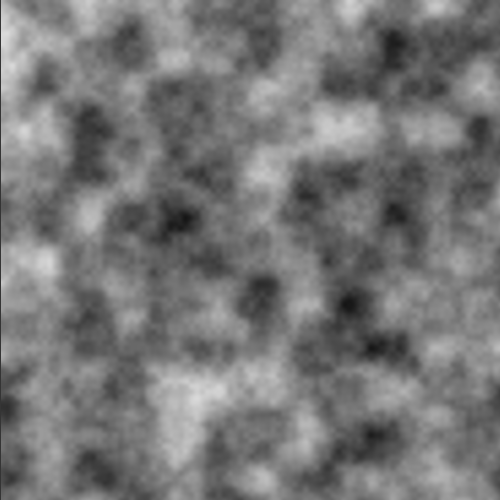


Рисунок 1.2.1. Визуальное представление карты численного шума.

Как используются карты шумов?

Карты шумов – это графические представления шума в заданном n-мерном регионе пространства. В случае процедурной генерации они представляют конкретную характеристику мира, например, высоту, температуру, влажность и т.д. Каждая точка на карте определяет конкретное значение в финальном произведении. Для генерации более интересных высот, пользуются сложением шумов с разными частотами. (См. Рисунок 1.2.2).

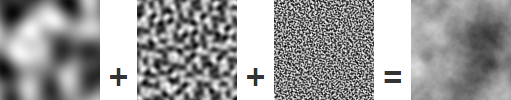


Рисунок 1.2.2. Сложение карт шумов с разными частотами.

Самый прямолинейный способ использования функции шума — использовать его напрямую как высоту. Использование шума смещения средней точки или шума Перлина — тоже примеры непосредственного использования.

Другой способ использования шума — использовать его как смещение от предыдущего значения. См. также [«случайное блуждание»](https://ru.wikipedia.org/wiki/Случайное_блуждание). Можно сделать обратное, и использовать разность между значениями шума. Это тоже можно воспринимать как модификацию функции шума.

Решаемые задачи

Генерация на картах шумов чаще всего используется для создания процедурной текстуры псевдослучайным методом или рельефа местности.

Свойства

Сложность построения карты шумов . Алгоритм реализует псевдоестественное распределение схожее с натуральным, природным.

### Wave Function Collapse (Коллапс волновой функции)

Истоки

Алгоритм Wave Function Collapse (WFC) Максима Гумина — это алгоритм генерации изображений на основе примеров, основанный на практике создания процедурного контента. Алгоритм WFC генерирует битовые изображения, локально подобные входному битовому изображению. Локальное подобие означает, что:

* (C1) Каждый паттерн NxN пикселей в выходных данных должен хотя бы раз встречаться во входных данных.
* (Слабое условие C2) Распределение паттернов NxN во входных данных должно быть подобным распределению паттернов NxN в значительно большом количестве наборов выходных данных. Другими словами, вероятность встречи определённого паттерна в выходных данных должна быть близка к плотности таких паттернов во входных данных.

Алгоритм WFC инициализирует выходное битовое изображение как полностью ненаблюдаемое состояние, в котором значение каждого пикселя является суперпозицией цветов входного битового изображения (поэтому если входное изображение чёрно-белое, то ненаблюдаемые состояния отображаются как различные оттенки серого). Коэффициентами этих суперпозиций являются вещественные, а не мнимые числа, поэтому в алгоритме не используется настоящая квантовая механика, скорее он был ею вдохновлён. Затем программа переходит в цикл наблюдения-распространения:

* На каждом этапе наблюдения среди ненаблюдаемого пространства выбирается область NxN, имеющая наименьшую энтропию Шеннона. Состояние этой области затем коллапсирует к состоянию определённости в соответствии с коэффициентами и распределением паттернов NxN во входящих данных.
* На каждом этапе распространения новая информация, полученная при коллапсе предыдущего этапа, распространяется по выходным данным.

На каждом этапе общая энтропия уменьшается, и в конце концов получаем полностью наблюдаемое состояние, коллапс волновой функции завершается.

Алгоритм

1. Считать входящее битовое изображение и подсчитать количество паттернов NxN. (необязательно) Дополнить данные паттернов повёрнутыми и отражёнными паттернами.
2. Создать массив с размерами выходных данных (называемый «wave»). Каждый элемент этого массива обозначает состояние области размером NxN в выходных данных. Состояние области NxN является суперпозицией паттернов NxN входящих данных с булевыми коэффициентами (то есть состояние пикселя в выходных данных является суперпозицией входящих цветов с вещественными коэффициентами). Коэффициент false означает, что соответствующий паттерн запрещён, коэффициент true означает, что соответствующий паттерн пока не запрещён.
3. Инициализировать волну в полностью ненаблюдаемом состоянии то есть, где все булевы коэффициенты имеют значение true.
4. Повторять следующие шаги:
   1. Наблюдение:
      1. Найти элемент волны с минимальной ненулевой энтропией. Если таких элементов нет (если у всех элементов энтропия нулевая или неопределённая), то завершить цикл (4) и перейти к шагу (5).
      2. Коллапсировать этот элемент в состояние определённости в соответствии с его коэффициентами и распределением паттернов NxN входящих данных.
   2. Распространение: распространить информацию, полученную на предыдущем шаге наблюдения.
5. К данному моменту все элементы волны или имеют полностью наблюдаемое состояние (все коэффициенты, кроме одного, равны нулю) или находятся в состоянии противоречия (все коэффициенты равны нулю). В первом случае вернуть выходные данные. Во втором случае завершить работу, ничего не возвращая.

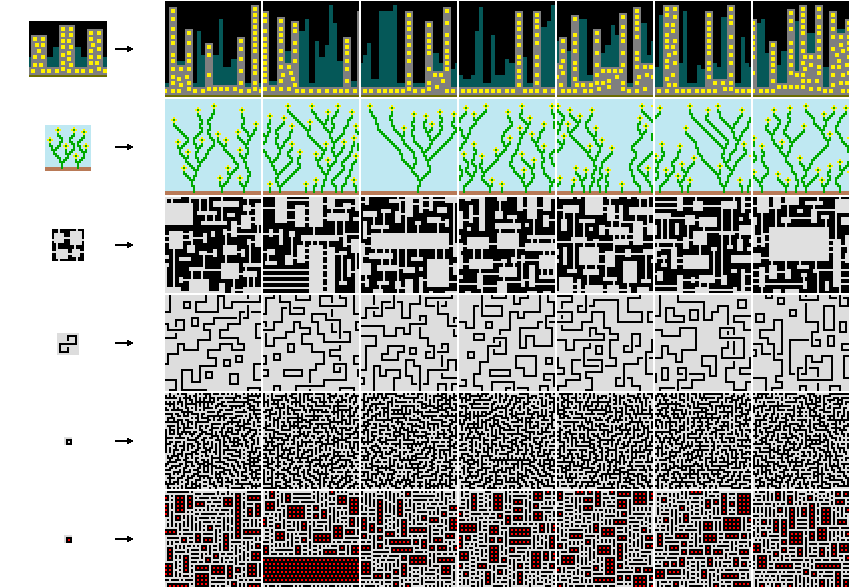


Рисунок 1.3.1. Пример работы алгоритма.

Решаемые задачи

Данный алгоритм подходит для решения многих задач, начиная с построения карт подземелий, ландшафта, фонов, до генерации стихов, из заранее подготовленных паттернов. Как видно из описанного выше алгоритма, для работы ему достаточно набора паттернов или правил их соединений, чтобы на выходе получить псевдоестественный результат.

Свойства

Точно оценить сложность построения по данному алгоритму невозможно, так как она зависит от данного набора паттернов или правил. Грубой оценкой сверху будет . Равномерность распределения регулируется только набором правил.

# Сравнение

Таблица 1. Сравнение описанных алгоритмов. (n – это количество ожидаемых фрагментов карты).

|  |  |  |  |
| --- | --- | --- | --- |
|  | Алгоритм на BSP-деревьях | Алгоритм на картах шумов | WFC |
| Сложность |  |  | Грубая оценка сверху |
| Решаемые задачи | Построение подземелий, планировка зданий и отдельных комнат | Генерация псевдоестественных текстур объектов, ландшафтов | Построение подземелий, городов, генерация ландшафта и текстур, генерация заданий из заранее подготовленных частей |
| Способы регулировки | Изменение количества разбиений пространства | Изменение частот генерации шума | Изменение весов паттернов, изменение правил соединений паттернов |

Из Таблицы 1. можно сделать вывод, что алгоритм коллапса волновой функции подходит для решения большего спектра задач и его легче регулировать, чем два других рассмотренных алгоритма.

# Глава 2. Реализация WFC

Для своей реализации алгоритма, я внес некоторые поправки в него. Теперь вектор «wave» представляет из себя линеаризованное пространство (размерность можно задать любую), каждый элемент вектора — это вектор порядковых номеров возможных для коллапсирования паттернов (далее тайлов). Для линеаризации двумерного пространства я написал отображение:

, где

*length* – ширина поля по *x*

и обратное преобразование. По аналогии можно реализовать для n-мерного пространства:



Листинг 2.1. Реализация отображения fi.

Я реализовал класс Chunk (дословно кусок карты/пространства). Поля класса:

* vector<vector<int> wave – вектор *коллекций*[[2]](#footnote-2) каждой позиции;
* vector<vector<bool> rules – правила соединения тайлов, i-ый элемент – это вектор смежности i-того тайла с другими;
* vector<Tile> chunk – линеаризованная часть карты, в который мы сохраняем результат работы генератора;
* int diversity – мощность множество уникальных тайлов;
* int length, width – размеры chunk. В дальнейшем заменить на vector<int> space\_size – вектор размеров n-мерного пространства;
* int smooth\_coef – количество применений метода сглаживания.

Приватные методы класса:

* bool Possible (int collapse\_target, int position) – метод проверки возможен ли на данной позиции (position) данный тайл (collapse\_target);
* int Entropy (int position) – метод возвращающий *энтропию*[[3]](#footnote-3) волны в данной точке;
* void InitWave () – метод инициализации wave;
* int Collapse (int position) – коллапс волны для данной позиции (position).

 Листинг 2.2. – коллапс тайла с положением position.

Коллапс происходит следующим образом: метод случайным образом выбирает возможный тайл из волны с координатами position и отчищает коллекцию оставляя в ней только выбранный тайл. Таким образом энтропия становиться равной единице, это в нашем случае означает, что данный тайл сколлапсировал;

* Tile Observation () – *наблюдает*[[4]](#footnote-4)тайл с наименьшей *энтропией.* Возвращает объект типа Tile. Возвращает ошибку invalid\_argument с сообщением "Wave realised.", если не нашел тайл с наименьшей энтропией. Это означает, что волна перешла в полностью наблюдаемое состояние;
* void Distribution (int position, vector<int> &mark) – распространяет информацию о сколлапсированном тайле по волне. Распространение основано на алгоритме прохода в ширину BFS. Всплеск происходит в вершине с координатами position. Происходить удаление невозможных тайлов из соседей тайла, пока сумма векторов правил соседних тайлов не станет равна единичному вектору;
* void Smooth () – метод сглаживания получившегося chunk. Для каждой точки пространства смотрит соседей, и если возможно, то заменяет данный тайл, на тайл с порядковым номером равным наиболее встречаемому порядковому номеру соседей;
* void WFC () – сам алгоритм, описанный выше.



Листинг 2.3. Реализация алгоритма.

Публичные методы класса:

* Chunk (vector<vector<bool>> &Rules, int Diversity, int Length = 10, int Width = 10, int Smooth\_coef = 1) – конструктор;
* void GeneredChunk () – метод вызывающий WFC ();

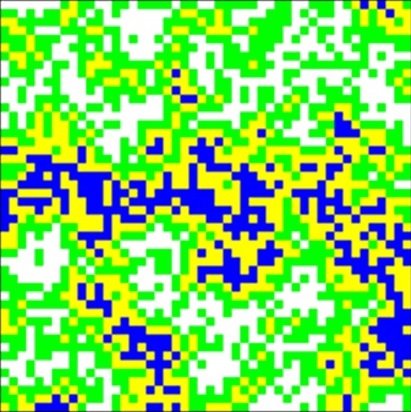


Рисунок 2.1. Результаты работы реализованного алгоритма.

* void PrintChunk () – метод вывода chunk в консоль;
* void SaveChunk () – метод сохранения chunk в файл;
* void LinkByLeft/LinkByUp/LinkByRight/LinkByBot () – метод присоединения уже реализованного chunk­1 к неопределенному chunk2. Чтобы chunk2 зависел от chunk1, присоединение происходит следующим образом: на основании информации о крайних тайлах из chunk1 наблюдается соответсвенные крайние тайлы chunk2.

Также я реализовал вспомогательные классы. Класс Tile содержит в себе информацию о том, какой в данных координатах должен находиться тайл. Класс Coord простой класс координат, нужен для хранения координат, и операциями над ними.

Реализованный алгоритм на вход получает вектор правил соединений тайлов. Правило соединения – это вектор, каждый элемент которого отображает соединение данного тайла с тайлом под порядковым номером значения вектора правил. Это один из инструментов регуляции. Также, есть возможность задать веса тайлам. Вес тайла прямопропорционально влияет на вероятность коллапса участка поля в данный тайл.

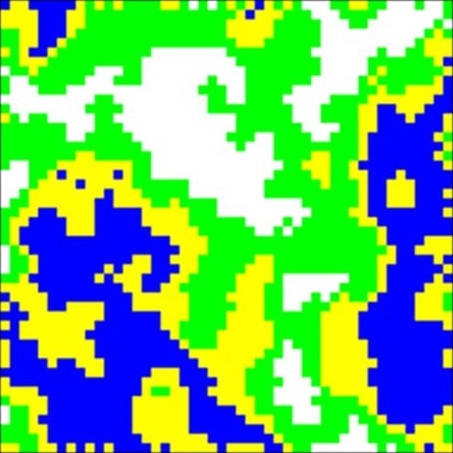


Рисунок 2.2. Результат работы алгоритма с применением сглаживания.

# Заключение

На сегодняшний день, на практике используются все описанные выше алгоритмы для решения своих задач. Так, есть жанры, в которых предпочитают использовать BSP-деревья для построения подземелья, карты шумов в основном используют геймдизайнеры для генерации текстур. WFC сравнительно новый алгоритм и сейчас все больше разработчиков переходят на него, так как он может покрыть больше задач, для решения которых алгоритм не нуждается в больших изменениях, и так как он довольно легок для понимания и реализации. Алгоритм коллапса волновой функции подходит для любых задач, в которых можно входные данные представить в виде неких паттернов и правил их соединений.

# Список литературы

1. Гумин, М. Алгоритм коллапса волновой функции [Электронный ресурс]: алгоритмы для разработки игр / М. Гумин – электрон. текст. дан. - 2016. – Режим доступа: <https://github.com/mxgmn/WaveFunctionCollapse>
2. Perlin, K. MAKING NOISE (англ.) [Электронный ресурс]: мат. моделирование / К. Перлин – 2012. – Режим доступа: http://www.noisemachine.com
3. Mark de Berg. Computational Geometry: Algorithms and Applications. — Springer Science & Business Media, 2008.
4. Шорт и Тарна Адамса, Процедурная генерация в гейм-дизайне /пер. с англ. М.С. Рыжиковой. – М.: ДМК Пресс, 2020. – 344 с.
5. Joseph S. Wisniewski. The Colors of Noise (англ.). Product Technology Partners, 1996

# Приложения

## Приложение 1.

#include <vector>

#include <iostream>

#include <random>

#include <time.h>

#include <fstream>

#include <algorithm>

#include <queue>

//Функция вывода stl 2D вектора

template <class T>

void PrintVector2D(std::vector<std::vector<T>> &v)

{

for (int i = 0; i < v.size(); i++)

{

for (int j = 0; j < v[i].size(); j++)

{

if (v[i][j] >= 0)

std::cout << " " << v[i][j] << " ";

else

std::cout << v[i][j] << " ";

}

std::cout << "\n";

}

}

//Функци вывода stl вектора

template <class T>

void PrintVector(std::vector<T> &v)

{

for (int j = 0; j < v.size(); j++)

{

std::cout << v[j] << " ";

}

std::cout << std::endl;

}

//Проверка наличия значения в векторе

template <class T>

bool In(std::vector<T> &v, T value)

{

for (int i = 0; i < v.size(); i++)

{

if (v[i] == value)

{

return true;

}

}

return false;

}

//Удаление элемента вектора по значению

template <class T>

void Remove(std::vector<T> &v, T value)

{

for (auto iter = v.begin(); iter != v.end(); ++iter)

{

if (\*iter == value)

{

v.erase(iter);

break;

}

}

}

//Сумма вектора

template <class T>

int SumVector(std::vector<T> &v)

{

int sum = 0;

for (int i = 0; i < v.size(); i++)

{

sum += v[i];

}

return sum;

}

//Поэлементное сложение двух векторов

template <class T>

void operator+=(std::vector<T> &v\_1, std::vector<T> &v\_2)

{

if (v\_1.size() != v\_2.size())

{

throw std::invalid\_argument("\nUnequil size.\n");

}

for (int i = 0; i < v\_1.size(); i++)

{

v\_1[i] = v\_1[i] + v\_2[i];

}

}

//Вспомогательный класс координат 2D

class Coord

{

protected:

int x, y;

public:

Coord(int X = 0, int Y = 0)

{

x = X;

y = Y;

}

Coord(const Coord &XY)

{

x = XY.x;

y = XY.y;

}

int getX() { return x; }

int getY() { return y; }

void setX(int X) { x = X; }

void setY(int Y) { y = Y; }

void print()

{

std::cout << "Coords: x = " << x << ", y = " << y << "\n";

}

};

//Вспомогательнай класс, хранящий в себе порядковый номер тайла (кусочка карты) ~collapse\_target~, который скаллапсировал в данных координатах ~xy~

class Tile

{

private:

Coord xy;

int collapse\_target;

public:

Tile(int X = 0, int Y = 0, int target = -1)

{

xy = Coord(X, Y);

collapse\_target = target;

}

Tile(Coord XY, int target = -1)

{

xy = XY;

collapse\_target = target;

}

Tile(const Tile &t)

{

xy = t.xy;

collapse\_target = t.collapse\_target;

}

Coord getCoords() { return xy; }

int getTarget() { return collapse\_target; }

void setCoord(int X, int Y)

{

xy = Coord(X, Y);

}

void setTarget(int target) { collapse\_target = target; }

void print()

{

std::cout << "Tile target\_collapse: " << collapse\_target << std::endl;

xy.print();

}

};

//Линеаризация координат

int Fi(Coord coords, int length)

{

return coords.getX() + coords.getY() \* length;

}

Coord ReversFi(int lin\_position, int length)

{

Coord xy = Coord(lin\_position % length, lin\_position / length);

return xy;

}

//Класс, включающий в себя методы для генерации и зранения карты

class Chunk

{

private:

std::vector<std::vector<int>> wave;

std::vector<std::vector<bool>> rules;

std::vector<Tile> chunk;

int diversity;

int length, width, smooth\_coef;

//Возможность постановки тайла

bool Possible(int collapse\_target, int position)

{

if (rules[chunk[position - 1].getTarget()][collapse\_target] &&

rules[chunk[position + 1].getTarget()][collapse\_target] &&

rules[chunk[position - length].getTarget()][collapse\_target] &&

rules[chunk[position + length].getTarget()][collapse\_target])

{

return true;

}

else

{

return false;

}

}

//Энтропия в точке position

int Entropy(int position) { return wave[position].size(); }

//Инициализация волны

void InitWave()

{

std::vector<int> init, anti;

anti.resize(0);

for (int i = 0; i < diversity; i++)

{

init.push\_back(i);

}

for (int i = 0; i < length \* width; i++)

{

if (i % length > 0 &&

i % length < length - 1 &&

i > length &&

i < length \* (width - 1))

{

wave.push\_back(init);

}

else

{

wave.push\_back(anti);

}

}

}

//Коллапс

int Collapse(int position)

{

if (Entropy(position) == -1)

{

throw std::invalid\_argument("Tile is impossible.");

return -1;

}

int target = wave[position][rand() % wave[position].size()];

wave[position].clear();

wave[position].push\_back(target);

return target;

}

//Наблюдает (находит) тайл с минимальной энтропией

Tile Observation()

{

int min\_entropy = diversity + 1;

int position = -1, target;

for (int i = 0; i < wave.size(); i++)

{

if (Entropy(i) < min\_entropy && Entropy(i) > 1)

{

min\_entropy = Entropy(i);

position = i;

}

}

if (position == -1)

{

throw std::invalid\_argument("Wave realised.");

}

try

{

target = Collapse(position);

}

catch (std::invalid\_argument e)

{

std::cout << e.what() << "\n";

return Tile(ReversFi(position, length), -1);

}

Tile tile(ReversFi(position, length), target);

return tile;

}

//Распространяет информацию о коллапсе по волне

void Distribution(int position, std::vector<int> &mark)

{

std::vector<int> temp\_mark;

std::queue<int> q;

q.push(position);

mark.push\_back(position);

temp\_mark.push\_back(position);

int c = 0;

while (!q.empty())

{

int temp\_position = q.front();

q.pop();

if ((Entropy(temp\_position) < diversity) && (Entropy(temp\_position) > 0))

{

std::vector<bool> possible\_tiles;

possible\_tiles.resize(diversity, false);

for (int i = 0; i < wave[temp\_position].size(); i++)

{

int collapse\_target = wave[temp\_position][i];

possible\_tiles += rules[collapse\_target];

}

if (SumVector(possible\_tiles) == diversity || SumVector(possible\_tiles) == 1)

{

continue;

}

// Add near tiles to queue

if (!(In(mark, temp\_position - 1) || In(temp\_mark, temp\_position - 1)))

{

q.push(temp\_position - 1);

temp\_mark.push\_back(temp\_position - 1);

}

if (!(In(mark, temp\_position + 1) || In(temp\_mark, temp\_position + 1)))

{

q.push(temp\_position + 1);

temp\_mark.push\_back(temp\_position + 1);

}

if (!(In(mark, temp\_position - length) || In(temp\_mark, temp\_position - length)))

{

q.push(temp\_position - length);

temp\_mark.push\_back(temp\_position - length);

}

if (!(In(mark, temp\_position + length) || In(temp\_mark, temp\_position + length)))

{

q.push(temp\_position + length);

temp\_mark.push\_back(temp\_position + length);

}

for (int i = 0; i < diversity; i++)

{

if (!possible\_tiles[i])

{

Remove(wave[temp\_position - 1], i);

Remove(wave[temp\_position + 1], i);

Remove(wave[temp\_position - length], i);

Remove(wave[temp\_position + length], i);

}

}

}

else

{

continue;

}

}

}

//Сглаживание

void Smooth()

{

for (int i = 2; i < width - 2; i++)

{

for (int j = 2; j < length - 2; j++)

{

std::vector<int> target\_color;

target\_color.resize(diversity, 0);

for (int l = -1; l < 2; l++)

{

for (int k = -1; k < 2; k++)

{

int position = Fi(Coord(j + k, i + l), length);

if (position == Fi(Coord(j, i), length))

{

continue;

}

target\_color[chunk[position].getTarget()] += 1;

}

}

int max = max\_element(target\_color.begin(), target\_color.end()) - target\_color.begin();

int ttl = 100;

while (!Possible(max, Fi(Coord(j, i), length)) && ttl--)

{

target\_color[max] = 0;

max = max\_element(target\_color.begin(), target\_color.end()) - target\_color.begin();

}

if (ttl)

{

chunk[Fi(Coord(j, i), length)].setTarget(max);

}

else

{

continue;

}

}

}

}

//Генерация

void WFC()

{

srand(time(NULL));

int ttl = length \* width \* 100;

Tile temp\_tile;

std::vector<int> mark;

while (ttl-- > 0)

{

try

{

temp\_tile = Observation();

}

catch (std::invalid\_argument e)

{

std::cout << e.what() << "\n";

break;

}

try

{

Distribution(Fi(temp\_tile.getCoords(), length), mark);

}

catch (std::invalid\_argument e)

{

std::cout << e.what() << "\n";

return;

}

chunk[Fi(temp\_tile.getCoords(), length)] = temp\_tile;

}

}

public:

Chunk(std::vector<std::vector<bool>> &Rules, int Diversity, int Length = 10, int Width = 10, int Smooth\_coef = 1)

{

diversity = Diversity;

length = Length;

width = Width;

rules = Rules;

smooth\_coef = Smooth\_coef;

chunk.resize(length \* width);

InitWave();

}

void GeneredChunk()

{

WFC();

}

//Вывод в консоль

void PrintChunk()

{

for (int i = 0; i < width; i++)

{

for (int j = 0; j < length; j++)

{

int position = Fi(Coord(j, i), length);

if (chunk[position].getTarget() >= 0)

std::cout << " " << chunk[position].getTarget() << " ";

else

std::cout << chunk[position].getTarget() << " ";

}

std::cout << "\n";

}

}

//Сохранение в файл

void SaveChunk()

{

std::ofstream fout("map.txt");

fout << length - 4 << " " << width - 4 << "\n";

for (int i = 2; i < width - 2; i++)

{

for (int j = 2; j < length - 2; j++)

{

int position = Fi(Coord(j, i), length);

if (position % length > 0 &&

position % length < length - 1 &&

position > length &&

position < length \* (width - 1))

{

fout << chunk[position].getTarget() << " ";

}

else

{

fout << -1 << " ";

}

}

}

}

};

int main()

{

std::vector<std::vector<bool>> rules;

int size, diversity = 6;

std::cin >> size;

rules.resize(diversity);

srand(time(NULL));

for (int i = 0; i < rules.size(); i++)

{

rules[i].resize(diversity, 0);

}

rules[0][0] = 1;

rules[1][1] = 1;

rules[2][2] = 1;

rules[3][3] = 1;

rules[4][4] = 1;

rules[5][5] = 1;

rules[0][1] = 1;

rules[1][0] = 1;

rules[1][2] = 1;

rules[2][1] = 1;

rules[2][3] = 1;

rules[3][2] = 1;

rules[3][4] = 1;

rules[4][3] = 1;

rules[4][5] = 1;

rules[5][4] = 1;

int n = 0;

// std::cin >> n;

int ttl = 1;

while (ttl--)

{

Chunk c(rules, diversity, size, size, n);

c.GeneredChunk();

// c.PrintChunk();

c.SaveChunk();

system("python show.py");

}

}

## Приложение 2.

from PIL import Image

import enum

class Color(enum.Enum):

black = -1

white = 5

green = 3

blue = 0

yellow = 2

grey = 4

light\_blue = 1

class TC(enum.Enum):

black = (0, 0, 0)

white = (255, 255, 255)

green = (0, 255, 0)

blue = (0, 0, 255)

yellow = (255, 255, 0)

colors = {-1 : (0, 0, 0), 5 : (255, 255, 255), 3 : (0, 255, 0), 0 : (0, 0, 255), 2 : (255, 255, 0), 4 : (128, 128, 128), 1 : (128, 128, 255)}

map = open('map.txt')

w, l = map.readline().split()

w = int(w)

l = int(l)

koef = 3

img = Image.new('RGBA', (w\*koef, l\*koef), 'white')

values = [int(i) for i in map.readline().split()]

def f(c, tc):

for i in tc:

if c.name == i.name:

return i.value

for i in range(w):

for j in range(l):

s = colors[values[j + i\*l]]

for k in range(3):

for m in range(3):

img.putpixel((j\*3+k, i\*3+m), s)

img.show()

1. BSP-дерево (англ. binary space partition tree) — дерево, отвечающее заданному двоичному разбиению пространства. [↑](#footnote-ref-1)
2. Коллекция – вектор возможных соединений. [↑](#footnote-ref-2)
3. Энтропия – количество возможных тайлов. [↑](#footnote-ref-3)
4. Наблюдение – нахождение тайла с наименьшей *энтропией* и его дальнейшее коллапсирование. [↑](#footnote-ref-4)