Петрозаводский государственный университет Институт математики и информационных технологий

Кафедра теории вероятностей и анализа данных

09.03.02 – Информационные системы и технологии

Отчет о практике по научно-исследовательской работе

Исследование и сравнение алгоритмов генерации уровней в игровых мирах

Выполнили:

студенты 3 курса группы 22305

О. А. Плугин подпись

И. О. Левицкий подпись

Руководитель:

Р. В. Сошкин, к. т. н., старший преподаватель

подпись

Итоговая оценка

оценка

Петрозаводск — 2024

**СОДЕРЖАНИЕ**

[Определения 3](#_Toc167738201)

[Введение 4](#_Toc167738202)

[Актуальность исследования 4](#_Toc167738203)

[Цель работы 4](#_Toc167738204)

[Задачи работы 4](#_Toc167738205)

[Теория 5](#_Toc167738206)

[Алгоритмы генерации лабиринтов 6](#_Toc167738207)

[Алгоритм двоичного дерева 6](#_Toc167738208)

[Алгоритм Sidewinder 8](#_Toc167738209)

[Алгоритм Олдоса-Бродера 10](#_Toc167738210)

[Примеры работы алгоритма 11](#_Toc167738211)

[Алгоритм Прима 12](#_Toc167738212)

[Примеры работы алгоритма 13](#_Toc167738213)

[Алгоритмы генерации комнат 14](#_Toc167738214)

[Алгоритм с использованием BSP-деревьев 14](#_Toc167738215)

[Примеры работы алгоритма 15](#_Toc167738216)

[Алгоритм генерации планов помещений 16](#_Toc167738217)

[Источники 17](#_Toc167738218)

[Приложения 18](#_Toc167738219)

[Двоичное дерево 18](#_Toc167738220)

[Sidewinder 18](#_Toc167738221)

[Олдоса Бродера 19](#_Toc167738222)

[Алгоритм Прима 20](#_Toc167738223)

[Алгоритм BSP Tree 21](#_Toc167738224)

# Введение

## Актуальность исследования

Разнообразие геймплея: В современном мире игроки ожидают увлекательного разнообразного геймплея, и одним из эффективных способов достижения этого является использование процедурной генерации уровней. Различные алгоритмы генерации уровней позволяют создавать уникальные игровые миры при каждом запуске игры. Это способствует увеличению интереса и вовлеченности игроков. Такая вариативность особенно важна для жанров игр, например, rogue like или sandbox, где повторяемость контента может быстро наскучить игрокам.

Удержание аудитории: Удержание игроков является важным фактором успеха игры. Процедурная генерация уровней может значительно увеличить время, проводимое игроками в игре, за счёт разнообразности поставляемого контента.

Экономия ресурсов разработчика: Создание большого количества уникальных уровней вручную занимает значительных финансовых и временных затрат. Алгоритмы генерации уровней позволяют разработчикам сократить эти затраты, автоматически создавая уровни.

## Цель работы

Целью работы является исследование и сравнение различных алгоритмов генерации уровней для выявления эффективности, практичности, применимости и влияния их на игровой процесс.

## Задачи работы

1. Провести обзор и классификацию существующих алгоритмов генерации уровней
2. Разработать и реализовать прототипы нескольких ключевых алгоритмов генерации уровней для демонстрации их работы и сравнения результатов
3. Оценить преимущества и недостатки описанных в работе алгоритмов по нескольким параметрам
4. Провести анализ разработанных прототипов
5. Исследовать, какие из алгоритмов лучше всего подходят

# Основы и инструменты

## Определения

Алгоритм — это точная последовательность инструкций, предназначенных для решения определённой задачи или выполнения определённой операции, чётко определенных и легко понимаемых. Алгоритмы являются базовыми строительными блоками программного обеспечения, определяя логику и порядок действий, которые должны быть выполнены компьютером для достижения желаемого результата.

Остовное дерево графа (или минимальное остовное дерево) — это подграф связного графа, который содержит все вершины исходного графа и является деревом. Иными словами, остовное дерево включает все вершины графа и наименьшее возможное количество рёбер, необходимое для поддержания связности графа, при этом не образуя циклов.

## Инструменты

# Алгоритмы генерации лабиринтов

## Алгоритм двоичного дерева

### Описание

Алгоритм двоичного дерева невероятно прост.

Для каждой клетки лабиринта он случайным образом выбирает, убрать стену в одном из двух направлений. Любая из пар направлений может быть использована, при условии использования её на всём поле – Север/Запад, Юг/Восток, Юг/Запад, Север/Восток.

Особенность алгоритма заключается в том, что ему не нужно хранить состояние лабиринта в памяти, т.к. по условиям генерации гарантируется доступ в любую ячейку поля. Это позволяет генерировать лабиринты огромных размеров без использования большого объёма памяти.

### Сложность алгоритма и затраты памяти

Размер поля равен **m \* n**.

Сложность алгоритма: **O(m \* n)**

Затраты памяти:

* **m \* n** ячеек поля
* **2 \* m \* n** стен (по 1 на направление)

### Недостатки и преимущества

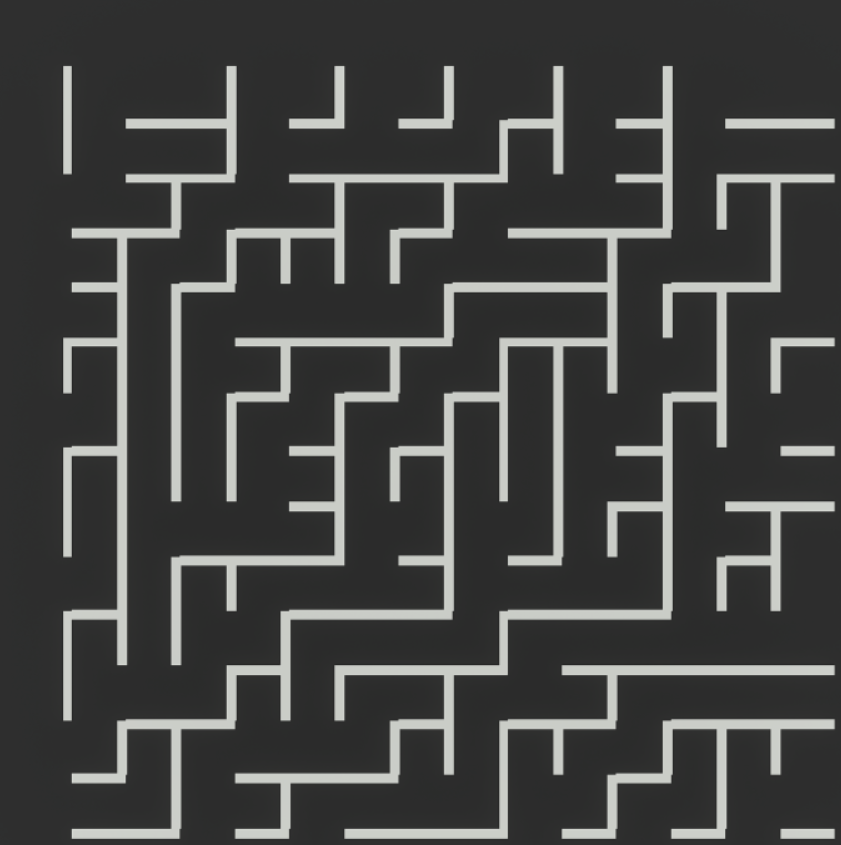
Преимущества

* Простота реализации
* Скорость работы алгоритма
* Малые затраты памяти
* Возможность генерировать лабиринты больших размеров

Недостатки

* Два прямых коридора по краям лабиринта (для гарантии доступа)
* Сильное диагональное смещение
* Однообразие структуры лабиринта
* Ограниченность в направлениях

### Примеры работы алгоритма



## Алгоритм Sidewinder

### Описание

Алгоритм Sidewinder схож с алгоритмом бинарного дерева. Он так же использует случайные числа для определения решения, но в отличие от более простого алгоритма бинарного дерева, содержит только один сквозной коридор и менее заметное смещение по диагонали.

Для каждой строки создаётся множество, называемое «пробежкой» которое может завершиться при ограничении пробежки в её направлении стеной.

По завершении пробежки случайным образом выбирается стена, разделяющая строки, из сохранённого множества, и удаляется. Это необходимо для сохранения гарантии доступа ко всем ячейкам.

### Сложность алгоритма и затраты памяти

Сложность и затраты остаются равными алгоритму бинарного дерева.

Размер поля равен **m \* n**

Сложность алгоритма: **O(m \* n)**

Затраты памяти:

* **m \* n** ячеек поля
* **2 \* m \* n** стен (по 1 на направление)

### Недостатки и преимущества

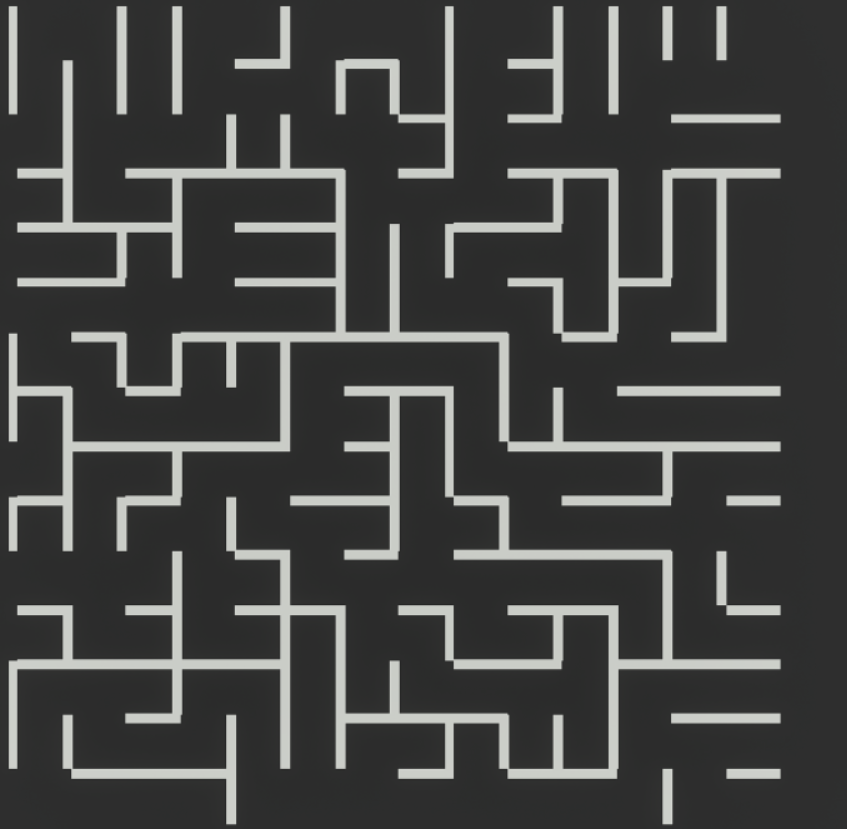
Преимущества

* Простота реализации
* Скорость работы алгоритма
* Малые затраты памяти
* Возможность генерировать лабиринты больших размеров

Недостатки

* Слабая запутанность лабиринта
* Ограниченность в направлениях

### Примеры работы алгоритма



## Алгоритм Олдоса-Бродера

### Описание

Начиная с случайной точки в поле, алгоритм блуждает по нему в поисках непосещенных вершин поля, каждый раз случайно выбирая направление. При нахождении непосещённой вершины он убирает стену между этой клеткой и клеткой, из которой пришёл.

### Сложность алгоритма и затраты памяти

Размер поля равен **m \* n**

Сложность алгоритма: **O(m \* n)**, но имеет возможность длительного зацикливания

Затраты памяти:

* **m \* n** ячеек поля
* **2 \* m \* n** стен (по 1 на направление)

### Недостатки и преимущества

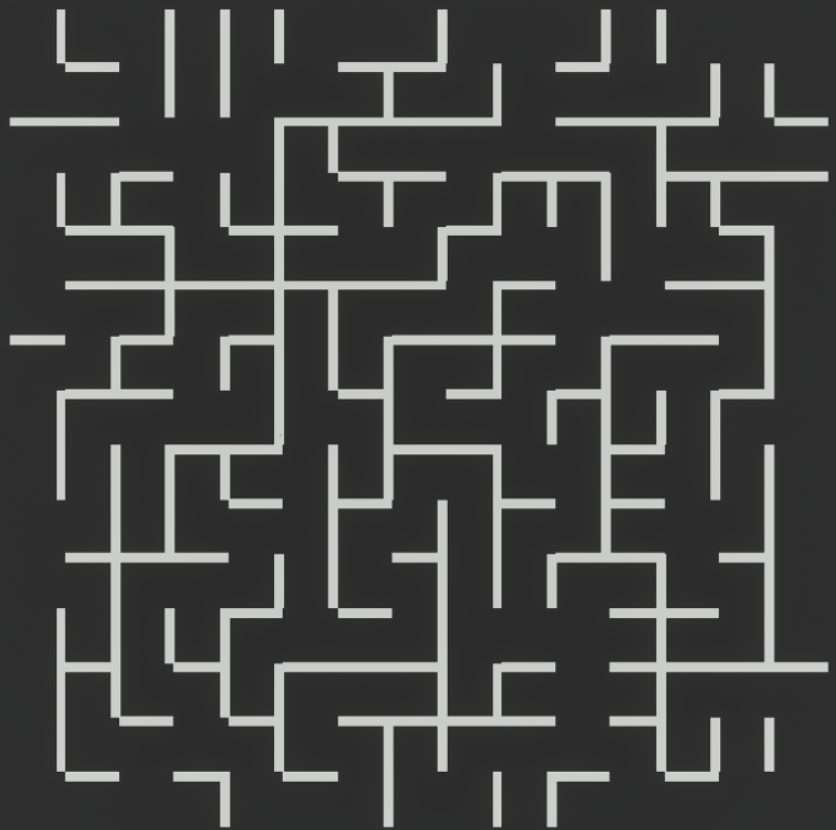
Преимущества

* Малые затраты памяти
* Хаотичность генерируемого лабиринта

Недостатки

* Большие временные затраты алгоритма даже при небольших размерах алгоритма
* Зависимость от генераторов случайных чисел

## Примеры работы алгоритма



## Алгоритм Прима

### Описание

Алгоритм основан на построении минимального остовного дерева взвешенного связного неориентированного графа.

1. Выбирается произвольная вершина графа, которая добавляется в изначально пустое множество.
2. Выбирается ребро с наименьшим весом, которое соединяет вершину, не находящуюся в множестве, с любой из вершин в множестве.
3. Найденная вершина добавляется в множестве.
4. Шаги 2 и 3 повторяются, пока не будут включены все вершины.

### Сложность алгоритма и затраты памяти

Размер поля равен **m \* n**

Количество вершин **V** = m \* n

Количество рёбер **E** = 4V

Сложность алгоритма: **O(ElogV)**

Затраты памяти:

* **m \* n** ячеек поля
* **2 \* m \* n** стен (по 1 на направление)
* **m2 \* n2** ребер

### Недостатки и преимущества

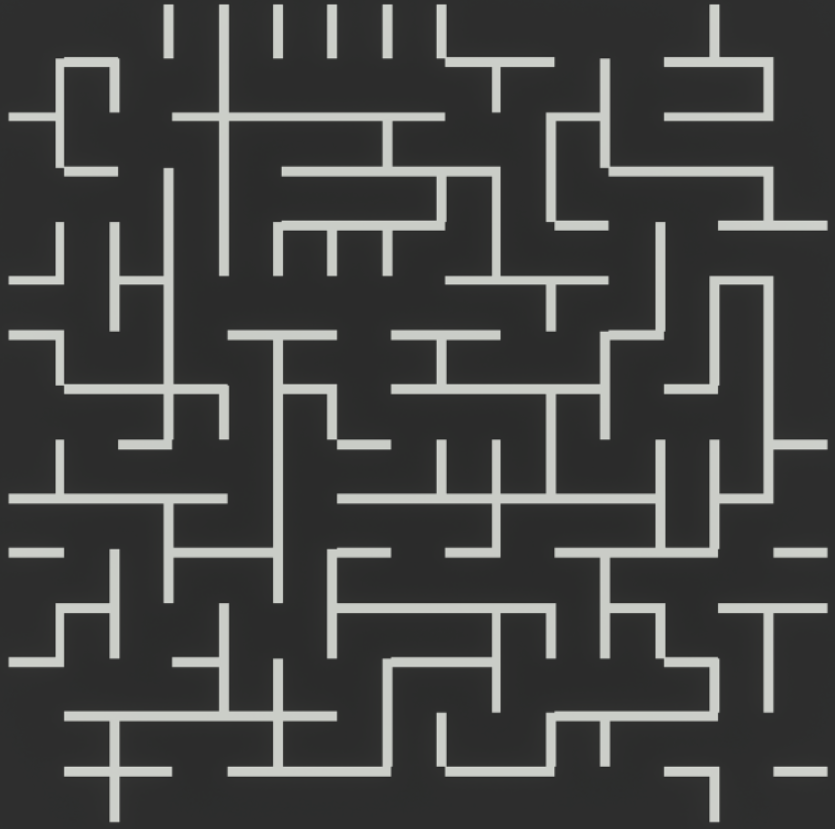
Преимущества

* Равномерное распределение путей в лабиринте
* Сложность создаваемых лабиринтов

Недостатки

* Сильное снижение производительности при увеличении размеров

## Примеры работы алгоритма



# Алгоритмы генерации комнат

## Алгоритм с использованием BSP-деревьев

### Описание

Есть начальная область заданного размера. Случайным образом она делится вертикально или горизонтально на две части, а затем процесс повторяется с полученными частями, пока не получится множество непересекающихся областей необходимого размера.

Затем в каждой области случайным образом генерируется прямоугольная комната, а после этого все комнаты соединяются между собой коридорами для перемещения.

После выполнения всех процедур не должно остаться изолированных комнат.

### Сложность алгоритма и затраты памяти

**n** – число ячеек, на которое необходимо разделить область

Сложность алгоритма: **O(n)**

Затраты памяти:

* **n** областей
* **n** комнат
* **n** коридоров

### Недостатки и преимущества

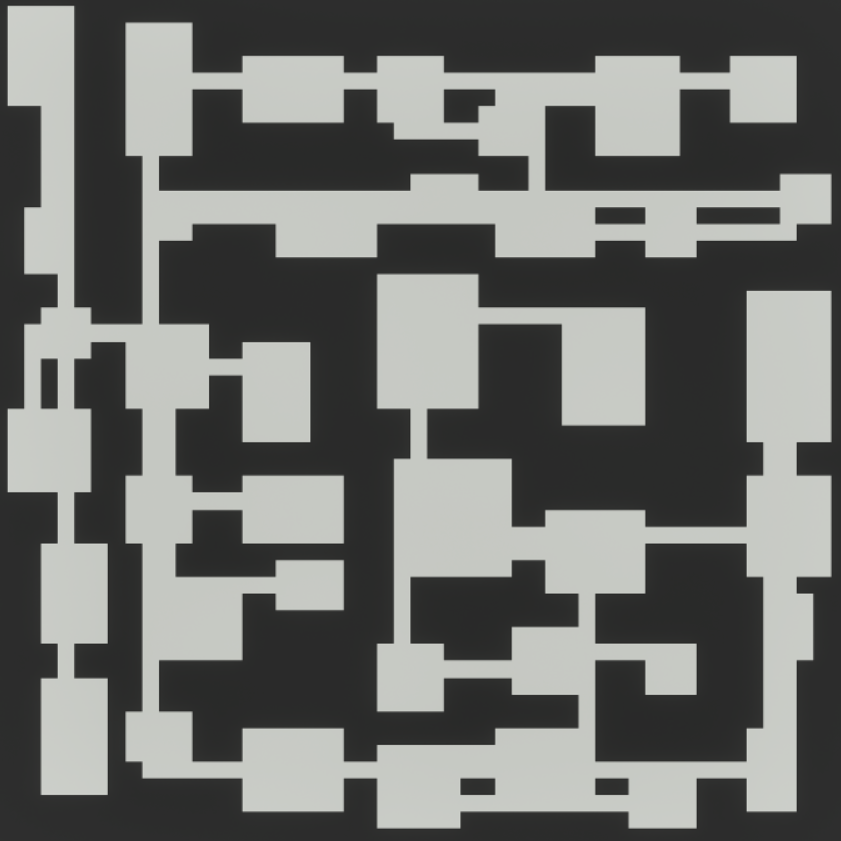
Преимущества

* Гибкость и настройка
* Иерархическая структура

Недостатки

* Создание больших пустых пространств
* Ограниченная случайность
* Сложность соединения комнат

## Примеры работы алгоритма



## Алгоритм генерации планов помещений

### Описание

Есть начальная область, а также начальные ограничения этой области.

# Сравнение алгоритмов

# Источники

1. <https://weblog.jamisbuck.org/2011/2/1/maze-generation-binary-tree-algorithm>
2. <https://upcommons.upc.edu/bitstream/handle/2117/328169/memoria.pdf>
3. <https://habr.com/ru/articles/320140/>
4. <https://habr.com/ru/articles/321210/>
5. <https://habr.com/ru/articles/332832/>
6. <https://habr.com/ru/articles/184818/>
7. <https://weblog.jamisbuck.org/2011/1/10/maze-generation-prim-s-algorithm>
8. <https://github.com/DevilishSasuke/LevelGenerationAlgorithms>

# Приложения

## Двоичное дерево

std::vector<bool>& BinaryTree::Algorithm() {

int bias = 0;

for (int y = 0; y < roomSize; ++y) {

bias = y \* roomSize;

for (int x = 0; x < roomSize; ++x) {

if (y > 0)

if (rng() % 2 == 0)

if (x != roomSize - 1) DeleteWall(bias + x, bias + x + 1);

else DeleteWall(bias - roomSize + x, bias + x);

else DeleteWall(bias - roomSize + x, bias + x);

else

if (x != roomSize - 1) DeleteWall(bias + x, bias + x + 1);

}

}

return resultGraph;

}

## Sidewinder

std::vector<bool>& Sidewinder::Algorithm() {

int bias = 0, wallToDelete = 0, cx = 0;

for (int y = 0; y < roomSize; ++y) {

bias = y \* roomSize;

for (int x = 0; x < roomSize; ++x) {

if (y > 0)

if (rng() % 2 == 0 && x != roomSize - 1)

DeleteWall(bias + x, bias + x + 1);

else {

wallToDelete = Randomize(rng, cx, x);

DeleteWall(bias - roomSize + wallToDelete, bias + wallToDelete);

if (x != roomSize - 1)

cx = x + 1;

else cx = 0;

}

else

if (x != roomSize - 1) DeleteWall(bias + x, bias + x + 1);

}

}

return resultGraph;

}

## Олдоса Бродера

std::vector<bool>& AldousBroder::Algorithm() {

enum Direction { Up = 0, Down = 1, Left = 2, Right = 3 };

Direction direction;

int unvisitedRooms = roomSizeSqr;

int ix = rng() % roomSize;

int iy = rng() % roomSize;

std::vector<std::vector<bool>> visited(roomSize, std::vector<bool>(roomSize));

visited[iy][ix] = true;

while (unvisitedRooms > 0) {

direction = static\_cast<Direction>(rng() % 4);

switch (direction) {

case Up:

HandleUp(ix, iy);

unvisitedRooms--;

break;

case Down:

HandleDown(ix, iy);

unvisitedRooms--;

break;

case Left:

HandleLeft(ix, iy);

unvisitedRooms--;

break;

case Right:

HandleRight(ix, iy);

unvisitedRooms--;

break;

}

}

return resultGraph;

}

## Алгоритм Прима

std::vector<bool>& Prim::Algorithm() {

if (graph.empty())

graph = GetRandomGraph(roomSize, rng);

std::vector<bool> selected(roomSizeSqr);

int x, y, edgeNum = 0;

selected[0] = true;

while (edgeNum < roomSizeSqr - 1) {

int min = INT\_MAX;

x = 0;

y = 0;

for (int i = 0; i < roomSizeSqr; ++i) {

if (selected[i]) {

for (int j = 0; j < roomSizeSqr; ++j) {

if (!selected[j] && graph[i][j]) {

if (min > graph[i][j]) {

min = graph[i][j];

x = i;

y = j;

} } } } }

DeleteWall(x, y);

selected[y] = true;

edgeNum++;

}

return resultGraph;

}

## Алгоритм BSP Tree

std::vector<bool>& BSPTree::Algorithm() {

auto leafs = CreateLeafs();

resultGraph = GetFloor(leafs);

return resultGraph;

}

std::vector<Leaf\*>& BSPTree::CreateLeafs() {

std::vector<Leaf\*> leafs;

int vectorSize = 0, vectorCapacity = 150;

const int MAX\_LEAF\_SIZE = 5;

Leaf root = Leaf(0, 0, roomSize, roomSize, rng);

bool splited = true;

leafs.reserve(vectorCapacity);

leafs.push\_back(&root);

vectorSize++;

while (splited && vectorSize + 2 < vectorCapacity) {

splited = false;

for (Leaf\* leaf : leafs) {

if (leaf->leftChild == NULL && leaf->rightChild== NULL)

if (leaf->width> MAX\_LEAF\_SIZE

|| leaf->height > MAX\_LEAF\_SIZE ||

(static\_cast<double>(rng()) / rng.max() > 0.25))

if (leaf->Split()) {

leafs.push\_back(leaf->leftChild);

leafs.push\_back(leaf->rightChild);

vectorSize += 2;

splited = true;

}

}

}

root.CreateRooms(roomSize);

return leafs;

}