ВВЕДЕНИЕ

Актуальность исследования процедурной генерации контента в современной игровой индустрии обусловлена стремительным ростом масштабов и сложности игровых проектов. С увеличением требований к разнообразию и объему контента традиционные методы ручного дизайна становятся всё менее эффективными. Процедурная генерация предоставляет возможность автоматического создания игровых элементов, таких как , ландшафты и диалоги, сюжетные линии, что значительно снижает затраты на разработку и повышает реиграбельность игр. Это особенно актуально в условиях ограниченных ресурсов и необходимости быстрого прототипирования.

Несмотря на широкое применение процедурной генерации, существует потребность в систематическом анализе и сравнении различных алгоритмов, используемых для создания игровых ландшафтов. Многие исследования сосредоточены на отдельных аспектах или специфических алгоритмах, что затрудняет выбор наиболее подходящего метода для конкретных задач. Кроме того, отсутствие единых критериев оценки эффективности алгоритмов усложняет их сравнение и интеграцию в игровые проекты.

Жанр rogue-like представляет собой идеальную платформу для исследования процедурной генерации, поскольку в таких играх генерация уровней является неотъемлемой частью игрового процесса. Случайное создание уровней и ландшафтов влияет на игровой баланс, сложность и интерес к повторному прохождению. Таким образом, анализ алгоритмов в контексте rogue-like позволяет установить чёткие критерии оценки и определить наиболее эффективные методы генерации.

Целью настоящей работы является разработка и экспериментальная проверка методики сравнительного анализа трёх алгоритмов процедурной генерации ландшафтных уровней: BSP-разбиения пространства, метода «шагающего пьяницы» и клеточных автоматов. Для достижения этой цели поставлены следующие задачи:

1. Изучить существующие подходы к процедурной генерации игровых ландшафтов, проанализировать успешные примеры их применения в индустрии и на этой основе обосновать выбор алгоритмов для последующего анализа.
2. Провести теоретический обзор выбранных алгоритмов, описав их основные принципы, структуру и характерные особенности.
3. Сформировать систему критериев, позволяющую объективно оценить качество и применимость алгоритмов генерации для построения игровых уровней.
4. Разработать архитектуру программного прототипа, обеспечивающего реализацию, визуализацию и тестирование выбранных алгоритмов в едином программном окружении.
5. Реализовать алгоритмы генерации в рамках разработанной архитектуры и провести их сравнительный анализ по заданным критериям.

Практическая значимость работы заключается в создании рекомендаций по выбору и настройке процедурных генераторов, которые помогут разработчикам игр сократить время разработки уровней, повысить вариативность контента и улучшить пользовательский опыт. Разработанные алгоритмы и архитектура программного обеспечения могут быть интегрированы в существующие игровые движки или служить основой для дальнейших исследований и образовательных программ в области геймдизайна и компьютерной графики.

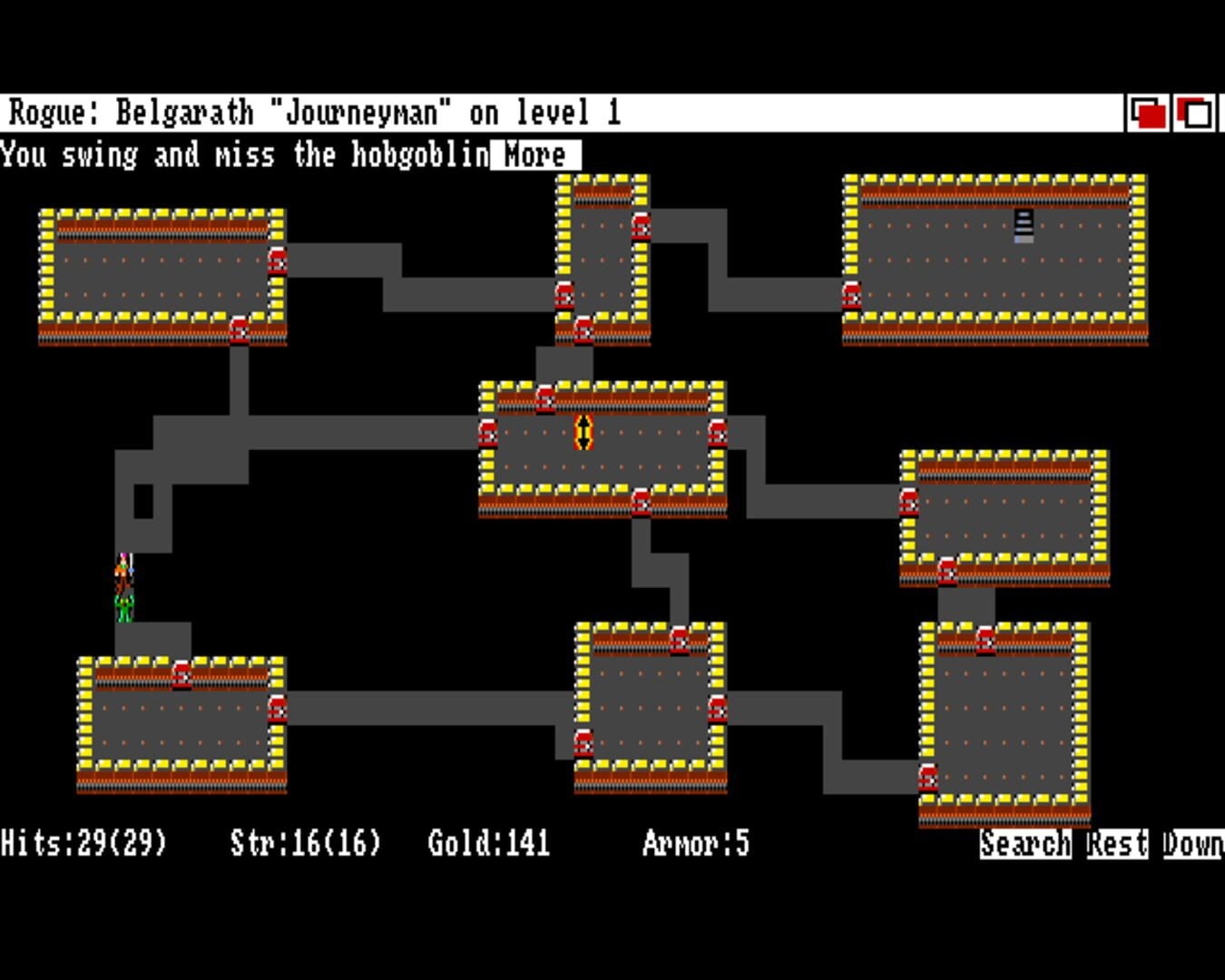
ТЕРМИНЫ

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

#### **1.1 История и развитие процедурной генерации в играх**

Методы процедурной генерации используются в игровой индустрии более четырех десятилетий и продолжают активно развиваться как с практической, так и с исследовательской точки зрения. Изначально их применяли не для того, чтобы облегчить труд разработчика. Условия для развития данных технологий создали технические ограничения компьютеров конца 70-х — начала 80-х. По сути это был всего лишь один из способов оптимизировать хранение данных: процедурная генерация позволяла упаковать целые миры в несколько килобайт памяти. Не использовать её было просто невозможно. [6] С ростом вычислительных возможностей и усложнением игровых миров процедурная генерация утратила сугубо утилитарную функцию, став полноценным элементом геймплейного проектирования и средством обеспечения реиграбельности.

Одним из первых зафиксированных применений процедурной генерации стала игра *Rogue* (1980), в которой алгоритмически создавались подземелья, меняющиеся при каждом новом прохождении. Это позволило реализовать элемент неопределенности и сделать каждую игровую сессию уникальной. Подобный подход стал основой зарождения жанра rogue-like, для которого процедурная генерация до сих пор остается системообразующим элементом.

**  
Рисунок 1 Игра “Rogue” 1980 г.**

С течением времени генеративные алгоритмы применялись в всё более масштабных проектах. Примером может служить *No Man’s Sky* (разработчик: Hello Games, 2016), в которой с помощью процедурных моделей было создано более 18 квинтиллионов уникальных планет, каждая со своими параметрами ландшафта, флоры, фауны и климата. Это стало возможным благодаря использованию сложных шумовых моделей, таких как фрактальный шум Перлина и его модификации, с применением параметрической генерации на уровне биомов.

**  
Рисунок 1.2. Игра “No man’s sky” 2016 г.**

В контексте rogue-like и близких к нему жанров процедурная генерация сохраняет своё ключевое значение. В играх *Dwarf Fortress*, *Spelunky*, *The Binding of Isaac* и *Dead Cells* применяются разные подходы — от стохастических симуляций до иерархических делений пространства. Сравнение различных алгоритмов в рамках таких игр позволяет оценивать их применимость в условиях строгих ограничений жанра — необходимость высокой реиграбельности, мгновенного создания уровня и сохранения баланса между хаосом и контролируемостью ландшафта.

Развитие процедурной генерации как инструмента игрового дизайна неразрывно связано с требованиями жанра rogue-like, где случайность и реиграбельность являются ключевыми элементами игрового процесса. Берлинская интерпретация жанра (Berlin Interpretation) подчеркивает необходимость динамической генерации уровней, перманентной смерти и непредсказуемости, что формирует строгие критерии для алгоритмов.

Критерий 1: Скорость генерации

В условиях rogue-like игрок часто перемещается между уровнями (например, при переходе на новый «этаж» подземелья). Задержки, вызванные медленной генерацией, нарушают динамику игры, требуя введения экранов загрузки или буферных зон. Для алгоритмов это означает необходимость оптимизации временной сложности — даже при размере карты 100×100 тайлов время выполнения не должно превышать 50–100 мс.

Критерий 2: Проходимость (связанность)

Уровень должен представлять собой связное пространство, где все ключевые зоны достижимы. Нарушение этого правила приводит к «мягким блокировкам» — ситуациям, когда игрок физически не может завершить уровень. Для оценки вводится метрика:

Значение ниже 90% считается критическим для жанра.

Критерий 3: количество мертвых зон и ландшафтных тупиков

Тупики и мертвые зоны создают тактические «узкие места», повышая напряженность, но их избыток разрушает баланс. В The Binding of Isaac (2011) тупики содержат бонусы, поощряя исследование, но их доля не превышает 10–15% от общей площади. Алгоритмы должны позволять регулировать этот параметр, например, через ограничение длины «слепых» ответвлений.

#### Критерий 4: Контролируемость параметров

Данный критерий определяет есть ли у алгоритма входные параметры, способные явно повлиять на остальные критерии. Пример: при увеличении параметра X увеличивается процент проходимости. Возможность тонкой настройки алгоритма — ключевое требование для адаптации под конкретные игровые механики.

Теоретический разбор выбранных алгоритмов

2.1 Двоичное разбиение пространства (BSP)

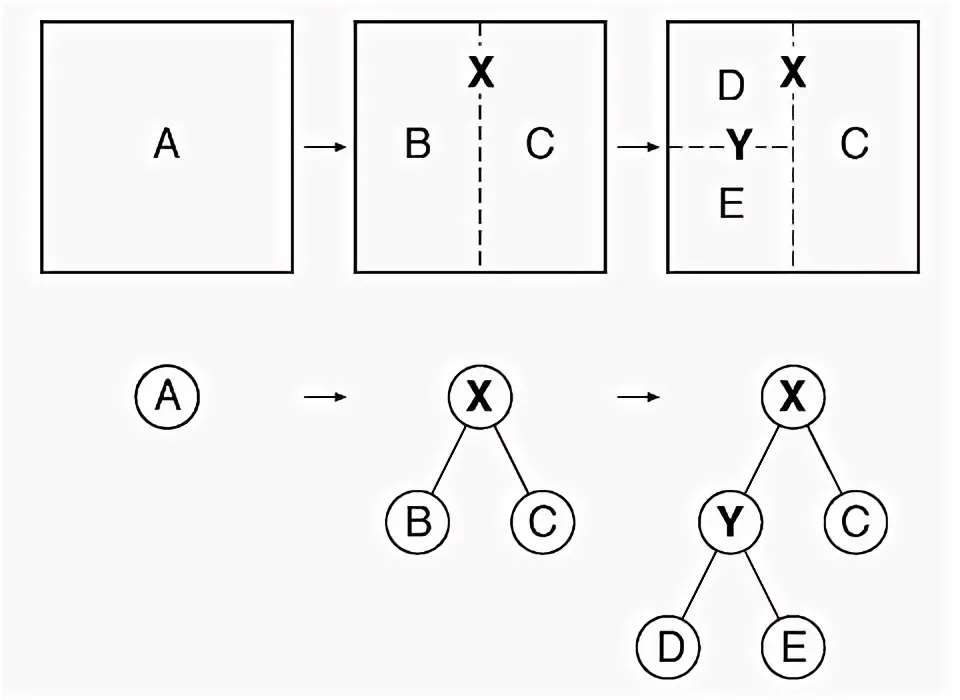
Алгоритм двоичного разбиения пространства (BSP, Binary Space Partitioning) был впервые предложен в контексте вычислительной геометрии в 1969 году Шумейкером и Гарднером для сортировки полигонов в сцене. [1]. Однако широкую известность он приобрёл в 1990-х годах благодаря использованию в игровой индустрии. В игре Doom, выпущенной в 1993 году, применялись BSP-деревья для оптимизации рендеринга сложных уровней, что позволяло эффективно определять порядок отрисовки полигонов. Как отмечает один из разработчиков игры, BSP стал ключевым инструментом для управления геометрией уровней в условиях ограниченных вычислительных ресурсов того времени.

BSP-алгоритм рекурсивно разделяет пространство на выпуклые подпространства (узлы) до достижения заданных условий остановки. В контексте генерации уровней процесс включает:

1. **Рекурсивное разбиение**: Исходная область (прямоугольник) делится на две части вдоль случайно выбранной оси (вертикальной или горизонтальной).
2. **Создание комнат**: В листовых узлах (тех, которые нельзя разделить далее) генерируются комнаты случайного размера, но не меньше минимального порога.
3. **Формирование коридоров**: Соседние узлы соединяются коридорами, обеспечивая связность уровня.

Критерии разделения включают минимальный размер области и вероятностный выбор направления разбиения. Например, если ширина области превышает удвоенный минимальный размер комнаты, разделение выполняется вертикально, в противном случае — горизонтально.

Для горизонтального разбиения формулы аналогичны. Связность между узлами обеспечивается за счёт создания коридоров в местах пересечения разделителей (рис. 2.1).

  
Рис. 2.1. Пример BSP-дерева и соответствующего уровня.

Алгоритм BSP обеспечивает высокую управляемость структурой уровней за счет строгой иерархии пространственного разбиения. Однако его применение сопряжено с рядом ограничений. Во-первых, жесткая геометрия генерируемых помещений: комнаты сохраняют прямоугольную форму, что снижает вариативность и естественность уровней.  
  
2.2 Алгоритм Drunkard's Walk (“Походка пьяницы”)

Метод «Походки пьяницы» (Drunkard's walk) – одна из вариаций метода «Случайной походки». Он получил свое название за соответствующий хаотичный узор, который образуется в результате его работы.[10]

Суть алгоритма заключается в следующем: от некоторой стартовой позиции на двумерной сетке начинается пошаговое перемещение «агента» — виртуального проходчика, который на каждом шаге выбирает новое направление движения случайным образом. При этом ячейки, по которым он проходит, помечаются как проходимые. Процесс продолжается до достижения заданного условия остановки: например, когда количество посещенных клеток достигает определённого порога или выполнено фиксированное число шагов.

Результатом работы алгоритма является связная область проходимого пространства, структура которой отличается высокой степенью нерегулярности. Это делает Drunkard Walk особенно полезным при создании нелинейных уровней, избегая однообразной прямоугольной геометрии, характерной для BSP. На **рисунке 2.2.1** представлена визуализация работы алгоритма: каждое случайное блуждание выполнено на протяжении восьми шагов, иллюстрируя характерную форму траекторий.

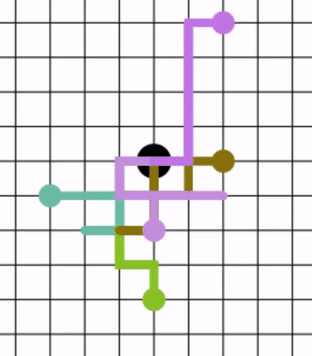


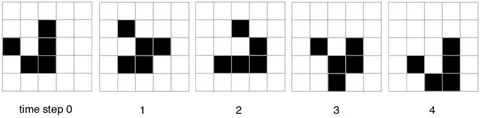
Рисунок 2.2.1. Пять случайных блужданий по восемь шагов

Как видно на изображении, даже при небольшом числе шагов получаются разнообразные траектории, что подтверждает способность алгоритма к генерации непредсказуемых и уникальных карт. Однако важно отметить, что несмотря на визуальную гибкость, алгоритм обладает и рядом ограничений. Прежде всего, он не гарантирует покрытие всей карты или соединение удалённых областей между собой. Это требует введения дополнительных механизмов постобработки, например, генерации соединительных коридоров или повторного запуска алгоритма с различными начальными точками.

Несмотря на указанные ограничения, метод «походки пьяницы» остаётся одним из ключевых инструментов процедурной генерации в жанрах roguelike, обеспечивая высокую степень стилистической свободы при минимальных вычислительных затратах.

2.3 Алгоритм “Клеточный автомат”.

Метод клеточного автомата широко используется в процедурной генерации уровней, особенно в задачах моделирования пещерных систем. Он основан на дискретной модели, где пространство представлено в виде двумерной решётки, каждая ячейка которой может находиться в одном из фиксированных состояний — как правило, «стена» или «пустота». Обновление состояний происходит по определённым правилам, зависящим от состояния соседних ячеек.(рис. 2.3.1) Согласно определению, приведенному в [6], клеточный автомат — это математическая модель, "в которой пространство и время дискретны, а поведение системы определяется локальными правилами перехода для каждой ячейки, основанными на ее соседях".

  
 Рисунок 2.3.1 Результат преобразований клеточного автомата.

Применительно к задачам генерации уровней в играх, алгоритм клеточного автомата позволяет на основе случайной инициализации карты постепенно формировать сложные, нерегулярные формы, близкие к естественным. Типичный процесс начинается с генерации карты, в которой каждая ячейка случайным образом определяется как проходимая или непроходимая (например, с вероятностью 45% — стена, 55% — проход). Далее несколько раз выполняется итерация: "если у клетки пять или больше соседей, которые являются стенами, она сама становится стеной. В противном случае — превращается в пустое пространство".

Такой подход даёт на выходе связные и органично выглядящие пещеры. Как указывает статья на Tuts [7], "несколько проходов симуляции клеточного автомата могут значительно изменить ландшафт, убирая одиночные клетки и укрепляя кластеры, создавая более цельную структуру пещеры".

Важным моментом является также то, что такие методы позволяют разрабатывать уровни, не опираясь на заранее заданную геометрию. Это даёт разработчику больше свободы в создании процедурных карт, особенно когда необходимо избежать однотипности. Однако, несмотря на высокую выразительность метода, он не всегда гарантирует связность всех частей карты. Поэтому после применения автомата может потребоваться дополнительная обработка: удаление не связанных с основным компонентом областей или их искусственное соединение.

Таким образом, алгоритм клеточного автомата представляет собой простой, но мощный инструмент генерации, способный на основе локальных правил порождать глобально согласованные и визуально насыщенные структуры.

ПРАКТИЧЕСКАЯ ЧАСТЬ.

3.1 Архитектура программного прототипа

Для достижения целей, поставленных в данной работе, необходимо создание программного комплекса, ориентированного на модульную, легко расширяемую архитектуру, обеспечивающую независимую работу нескольких алгоритмов процедурной генерации. Основной задачей при разработке такой системы является обеспечение максимальной изоляции алгоритмической логики от компонентов визуализации, постобработки и аналитики. Данный подход позволяет производить независимый и корректный анализ эффективности и поведения каждого алгоритма в отрыве от влияния сторонних модулей, таких как графическая отрисовка или сбор метрик, которые могут исказить результаты.

В качестве платформы для реализации выбран игровой движок Unity, что обусловлено его широкими возможностями по визуализации, отладке и быстрому прототипированию игровых механик. Дополнительным преимуществом является наличие развитой системы компонент и удобного инструментария для разработки пользовательского интерфейса, что позволит реализовать визуальную среду для конфигурации алгоритмов и оценки результатов их работы. Пользователь взаимодействует с интерфейсом в режиме реального времени, выбирая нужный алгоритм генерации и задавая параметры его выполнения. Интерфейс напрямую передает информацию центральному управляющему классу **LevelGenerator**, который выступает координатором всех последующих действий.

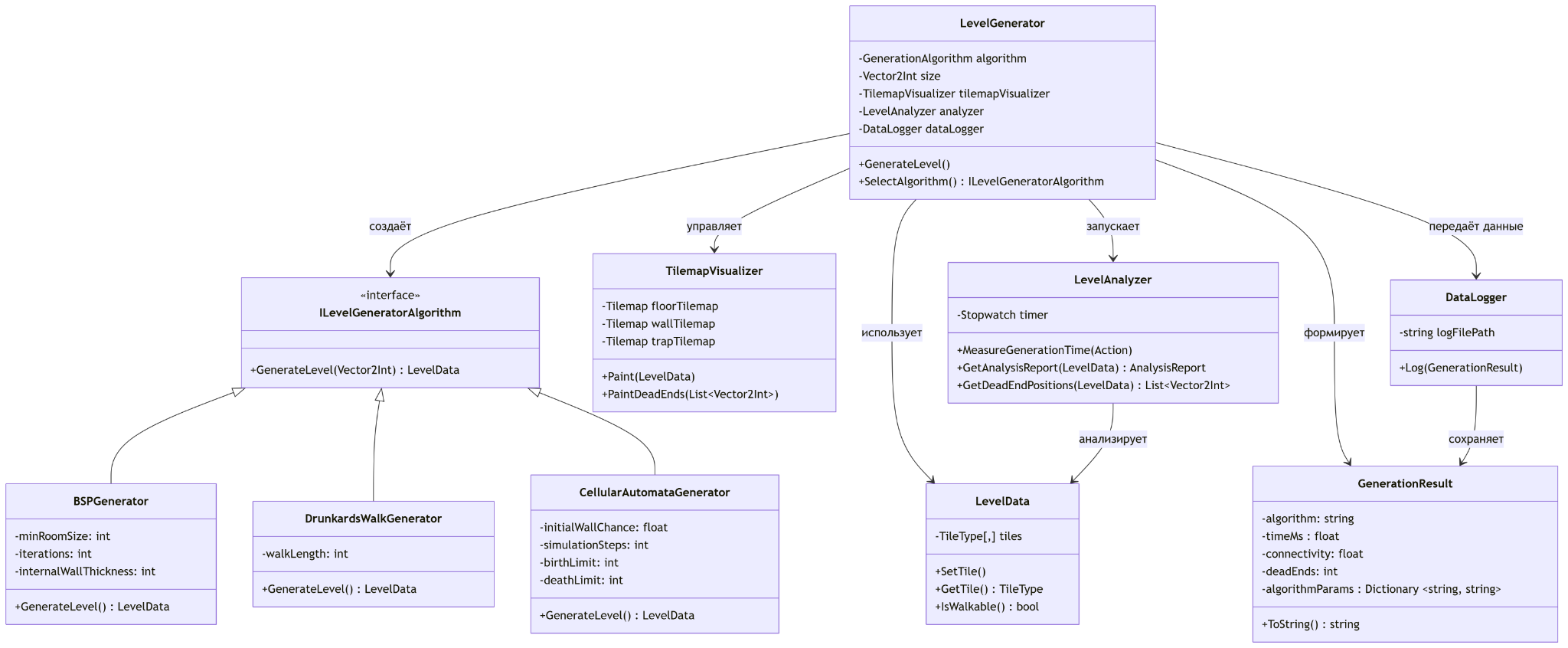
Основной задачей **LevelGenerator** является инициализация соответствующего алгоритма генерации уровня. Каждый алгоритм реализуется как отдельная сущность, удовлетворяющая общему интерфейсу, что позволяет гибко управлять реализациями, заменять и масштабировать их без изменений в окружающем коде. Такой подход реализует принцип инверсии зависимостей, при котором высокоуровневая логика не зависит от конкретных реализаций алгоритмов.

После выполнения алгоритма результат помещается в унифицированную структуру данных **LevelData**, которая содержит двумерный массив клеток, каждая из которых может принимать определенное состояние (например, "стена" или "пол"), заданное через перечисление **TileType**. Использование перечислений вместо магических чисел повышает читаемость кода и снижает вероятность ошибок при передаче данных между модулями. Унификация формата выходных данных позволяет независимо от конкретного алгоритма использовать единую визуализацию и единый подход к аналитике.

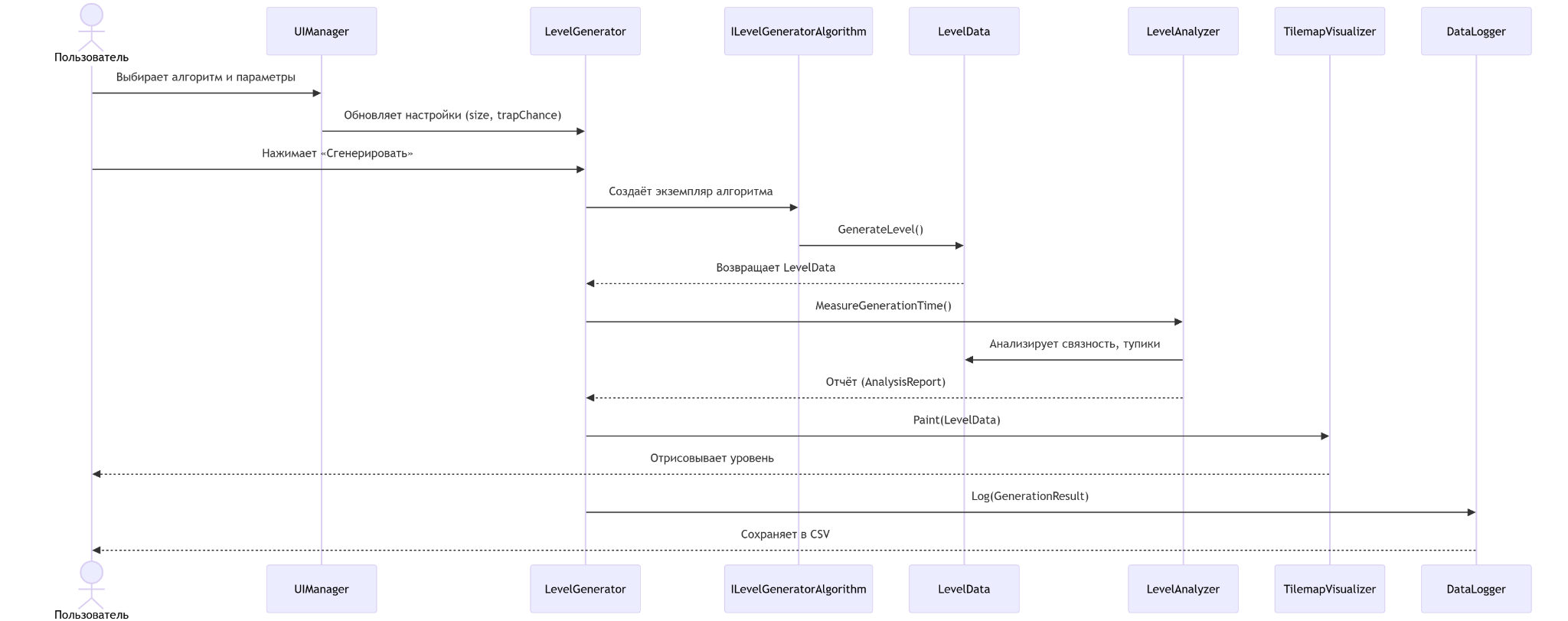
Полученные данные передаются в визуализатор, который отвечает за отображение уровня в игровом окне Unity. Так как Unity требует, чтобы все визуальные изменения происходили на сцене в основной нити исполнения, визуализация выносится в отдельный компонент, связанный с движком, что является характерной особенностью данной реализации. Обновление визуализации и повторная генерация могут происходить без перезапуска всей сцены, что существенно ускоряет итерации при тестировании и сравнении алгоритмов. Это особенно важно при массовом запуске тестов, когда необходимо быстро переключаться между различными конфигурациями.

Дополнительно, после получения результатов генерации, данные автоматически передаются в модуль анализа. Он изолирован от остальной логики и занимается исключительно вычислением ключевых метрик, таких как время генерации, доля тупиков, плотность карты, процент проходимости и другие. Результаты сохраняются в формате CSV, пригодном для последующего статистического анализа. Изоляция аналитического модуля от основной логики генерации обеспечивает чистоту эксперимента и исключает влияние побочных факторов на результат.

Таким образом, архитектура системы построена на принципах слабой связности и высокой модульности, где взаимодействие между компонентами осуществляется строго через заранее определенные интерфейсы. Это делает систему масштабируемой и удобной для дальнейшего расширения, позволяя интегрировать новые алгоритмы или изменять существующие модули без переработки всей архитектуры. На диаграмме классов (рис. 3.1) представлено взаимодействие ключевых компонентов

  
3.1 Диаграмма классов.

Последовательность операций, осуществляемых при генерации уровня, детализирована на диаграмме взаимодействий (рис. 3.2).

  
Рисунок 3.2. Диаграмма взаимодействий.

3.2 Реализация алгоритмов генерации

После описания архитектуры генерации уровней и представления интерфейса **ILevelGeneratorAlgorithm**, определяющего единый контракт для всех генераторов, имеет смысл рассмотреть конкретные реализации алгоритмов, интегрированных в проект. Начнем с бинарного разбиения пространства (Binary Space Partitioning, BSP), которое используется для формирования иерархической структуры комнат с возможностью их дальнейшего соединения. Эта реализация была выбрана в качестве базовой из-за высокой степени управляемости и компактного представления пространства.

Алгоритм BSP реализован в классе **BSPGenerator**, который инкапсулирует логику построения иерархии прямоугольных областей, генерации комнат в терминальных узлах и создания соединительных проходов. Конструктор принимает три параметра: минимальный размер комнат (**minRoomSize**), максимальное число итераций деления (**maxIterations**) и толщину внутренних стен (**internalWallThickness**). Это позволяет гибко контролировать как размерность, так и плотность структуры будущего уровня.

Ключевым этапом алгоритма является разбиение пространства. Оно осуществляется в методе **SplitNodes**, где список узлов итеративно дополняется потомками:

| bool canSplitVertical = node.rect.width >= (minRoomSize \* 2 + internalWallThickness); bool canSplitHorizontal = node.rect.height >= (minRoomSize \* 2 + internalWallThickness); |
| --- |

Листинг 3.1 Заполнение списка узлов потомками.

Выбор направления деления (горизонтальное или вертикальное) осуществляется случайным образом, однако только при наличии достаточного пространства. Это предотвращает формирование узлов, в которых невозможно разместить хотя бы одну комнату.

Пример разбиения по горизонтали:

| int splitY = Random.Range(minRoomSize, availableHeight - minRoomSize + 1); |
| --- |

Листинг 3.2. Разбиение по горизонтали.

После выбора точки разбиения создаются два дочерних узла и фиксируется стена, необходимая для последующего построения проходов.

Важно отметить, что иерархическая структура BSP-дерева обеспечивает соответствие между логическим делением пространства и геометрией уровня. Это позволяет эффективно реализовывать как создание комнат, так и прокладку соединений между ними.

После завершения этапа деления, в каждом терминальном (листовом) узле формируется прямоугольная комната, параметры которой ограничены размером узла и границами карты. Используется метод **Mathf.Clamp**, предотвращающий выход за пределы карты:

| int roomX = Mathf.Clamp(node.rect.x, ExternalWallThickness, width - ExternalWallThickness - 1);  int roomWidth = Mathf.Clamp(node.rect.width, 1, width - roomX - ExternalWallThickness); |
| --- |

Листинг 3.3 Распределение комнат в узлах

Комната формируется путем замены соответствующих тайлов на тип **Floor**. Таким образом, уровень начинает наполняться проходимыми областями.

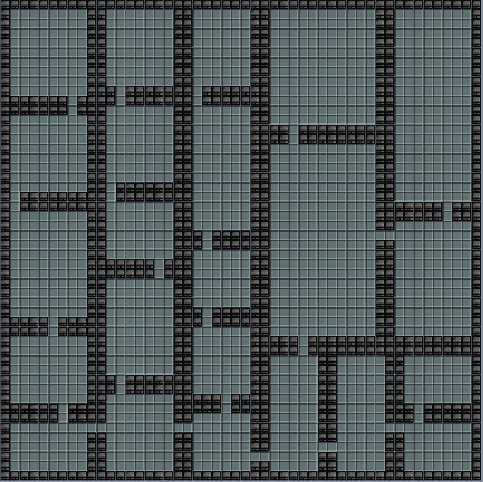
Формирование связности между комнатами реализуется в методе **CreatePassages**. Для каждого внутреннего узла, определяется пара потомков, содержащих комнаты, после чего выбирается координата соединения. Логика выбора прохода зависит от направления перегородки: для горизонтальных стен выбирается **x** в пересечении проекций комнат, а **y** соответствует координате стены. Аналогично, для вертикального деления — фиксируется **x**, а **y** варьируется.

Далее в выбранной области тайлы заменяются на Floor, обеспечивая проходимость между двумя соседними областями. При этом учитывается толщина перегородки, что позволяет реализовать не только одиночные двери, но и полноценные коридоры.

Отличительной чертой данной реализации является учет толщины стен и перегородок. В большинстве базовых реализаций BSP комната полностью заполняет область узла, в то время как здесь оставляется место под внутренние перегородки. Это не только повышает реализм, но и позволяет избежать визуального наложения элементов. Кроме того, предусмотрена защита от некорректных координат, благодаря чему алгоритм остается стабильным даже при граничных параметрах.

Также стоит отметить строгую декомпозицию логики: каждый этап (деление, размещение комнат, соединение) вынесен в отдельный метод, что упрощает модификацию и тестирование.

В результате выполнения генерации по алгоритму BSP получается уровень, состоящий из произвольно размещенных комнат, соединенных коридорами. Благодаря гибкости параметров алгоритм подходит как для построения плотных лабиринтов, так и для создания уровней с широкими, изолированными помещениями. Он также легко масштабируется на большие карты. Пример получаемого уровня представлен на рисунке 3.1, заданные параметры: число итераций = 5, минимальный размер комнаты = 5, ширина стены = 2, размер уровня 50x50.

  
рисунок 3.1 Пример работы алгоритма BSP.

### 4.2 Реализация генерации на основе алгоритма Drunkard’s Walk

Следующим рассмотренным методом процедурной генерации уровней стал алгоритм «Пьяной походки» (Drunkard’s Walk). Реализация генерации уровня методом Drunkard's Walk сводится к пошаговому перемещению агента по сетке от стартовой позиции с постепенным "протаптыванием" проходов, где каждый пройденная клетка становится частью пола. Такой подход обеспечивает формирование уровней без необходимости построения явной топологии комнат, в отличие от BSP-метода, и подходит для более органичных и свободных форм.

Класс **DrunkardsWalkGenerator** содержит главный параметр **walkLength**, определяющий количество шагов, которые совершит виртуальный «бродяга». Этот параметр напрямую влияет на плотность покрытия пола — при малом значении уровни будут фрагментированы, при слишком большом — переходить в сплошную зону пола. Значение может быть задано извне, в том числе рассчитываться как доля от общей площади уровня.

Изначально агент начинает своё движение от центра уровня. На каждой итерации цикла длиной **walkLength** происходит следующая последовательность действий:

1. Текущая клетка становится полом.
2. Позиция добавляется в множество посещенных клеток, что необходимо для последующего построения стен.
3. Выбирается случайное направление — вверх, вниз, влево или вправо.
4. Производится шаг в новом направлении.
5. Ограничение выхода за границы сохраняет рамку уровня: нельзя выйти за пределы по обеим осям.

Реализация представлена на листинге 4.2.1.

| for (int i = 0; i < walkLength; i++)  {  levelData.SetTile(position.x, position.y, TileType.Floor);  visited.Add(position);   Vector2Int direction = GetRandomDirection();  position += direction;   position.x = Mathf.Clamp(position.x, 1, size.x - 2);  position.y = Mathf.Clamp(position.y, 1, size.y - 2);  } |
| --- |

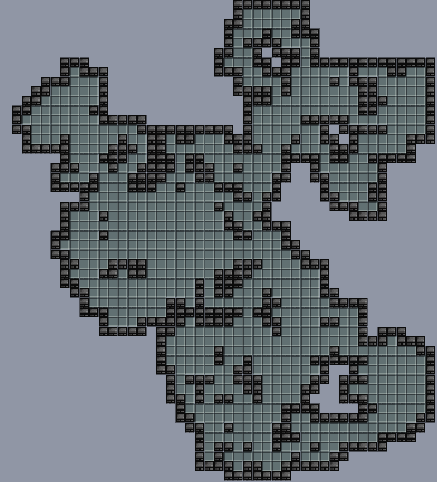
Листинг 4.2.1. Реализация заполнения уровня.

После завершения маршрута необходимо сформировать границы, окружающие сгенерированные полы. Это реализуется за счёт прохода по всем посещённым позициям и анализа их окружения в пределах соседних 8 клеток. Если соседняя клетка осталось пустой, он преобразуется в стену:

| foreach (var floorPos in visited) {  for (int dx = -1; dx <= 1; dx++)  {  for (int dy = -1; dy <= 1; dy++)  {  Vector2Int neighbor = floorPos + new Vector2Int(dx, dy);  if (levelData.GetTile(neighbor.x, neighbor.y) == TileType.Empty)  {  levelData.SetTile(neighbor.x, neighbor.y, TileType.Wall);  }  }  } } |
| --- |

Листинг 4.2.2 Формирование стен.

Такая логика гарантирует, что каждое полупроходное помещение будет изолировано от остального пространства. Таким образом, стены определяются контекстно, исходя из финального положения пола. Пример уровня сгенерированного данным алгоритмом на рисунке 4.2.1.

  
Рисунок 4.2.1 Пример генерации уровня алгоритмом Drunkard’s Walk.

В отличие от BSP, структура которых предполагает строгую иерархию, данный метод формирует уровни без жёстких геометрических ограничений. Это повышает уникальность игрового опыта, но снижает контролируемость процесса генерации.

### **4.3 Генерация уровня с использованием клеточных автоматов**

Клеточные автоматы представляют собой мощный инструмент для процедурной генерации уровней, особенно в задачах моделирования органичных подземелий и пещер. Их основное преимущество заключается в способности создавать естественно выглядящие структуры без явных геометрических повторов. В данной работе реализован алгоритм клеточного автомата, использующий простые правила рождения и смерти ячеек, что обеспечивает высокую степень визуального разнообразия при сохранении топологической связности уровня.

Моя реализация алгоритма представлена в виде отдельного класса CellularAutomataGenerator. В его составе определены ключевые параметры: **initialWallChance** — вероятность того, что на старте генерации ячейка станет стеной. **simulationSteps** — количество итераций симуляции автомата и **birthLimit** и **deathLimit** — параметры, определяющие правила обновления клеток.

Изначально создаётся двумерная карта заданного размера, каждая ячейка которой заполняется случайным образом в соответствии с заданной вероятностью. (Листинг 4.3.1)

| for (int x = 0; x < size.x; x++) {  for (int y = 0; y < size.y; y++)  {  if (Random.value < initialWallChance)  levelData.SetTile(x, y, TileType.Wall);  else  levelData.SetTile(x, y, TileType.Floor);  } } |
| --- |

Листинг 4.3.1. Начально распределение стен исходя из заданной вероятности.

После первичного заполнения, карта проходит через несколько итераций обработки, во время которых применяются правила клеточного автомата. Каждая клетка анализирует своё локальное окружение — восемь соседей по квадратной окрестности — и в зависимости от их количества принимает решение о своём будущем состоянии.

Сама симуляция реализована в методе **SimulateStep,** где создаётся новая карта, в которую поочередно копируются новые значения всех клеток.

| if (oldData.GetTile(x, y) == TileType.Wall) {  if (neighborWalls < deathLimit)  newData.SetTile(x, y, TileType.Floor);  else  newData.SetTile(x, y, TileType.Wall); } else {  if (neighborWalls > birthLimit)  newData.SetTile(x, y, TileType.Wall);  else  newData.SetTile(x, y, TileType.Floor); } |
| --- |

Листинг 4.3.2. Основная логика клеточного автомата. (Игра по правилам)

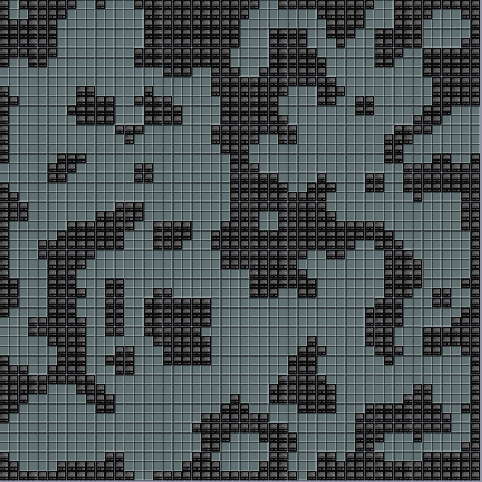
В данной логике реализованы два основных правила: eсли клетка является стеной, но окружена недостаточным количеством соседей-стен (меньше **deathLimit**), она "умирает", становясь полом. Если клетка является полом, но окружена достаточным числом стен (больше **birthLimit**), она "рождается" как стена.

Для правильной работы алгоритма необходим точный учёт соседних стен. Этот процесс реализован в листинге 4.3.3, где проверяются все смежные ячейки по квадратной окрестности, кроме самой центральной. При этом учитываются и границы карты — они интерпретируются как стены, что позволяет сохранить «замкнутость» структуры:

| if (nx < 0 || ny < 0 || nx >= data.Width || ny >= data.Height) {  count++; } else if (data.GetTile(nx, ny) == TileType.Wall) {  count++; } |
| --- |

Листинг 4.3.3. Поиск соседей.

На практике параметры входные для алгоритма параметры подбираются эмпирически, от конкретной идеи уровня. Например, значения initialWallChance = 0.45, simulationSteps = 5, birthLimit = 4 и deathLimit = 3 создают сбалансированные уровни, с хорошо выраженной структурой. (рисунок 4.3.1) Повышение количества итераций или вероятности появления стен приводит к генерации более замкнутых ландшафтов, в то время как снижение — к более открытым структурам.

  
Рисунок 4.3.1 Результат работы клеточного автомата.

3.3 Сравнительный анализ результатов  
Постановка методики тестирования. Сбор основных метрик и распределение их по критериям. Общие графики, тепловые карты. Контролируемость, разнообразность. Финальные выводы по каждому алгоритму + выбор лучшего по критериям.

СПИСОК ЛИТЕРАТУРЫ

1. Адамс, Т. Процедурная генерация в гейм-дизайне / Т. Адамс, Т. Шорт ; под ред. Т. Х. Шорт, Т. Адамса ; пер. с англ. М. С. Рыжиковой. – Москва : ДМК Пресс, 2020. – 344 с. – ISBN 978-5-97060-860-9. – Текст : электронный // Лань : электронно-библиотечная система. – URL:<https://e.lanbook.com/book/190739> (дата обращения: 24.05.2025). – Доступен для авториз. пользователей.
2. Cellular automaton [Электронный ресурс] // Wikipedia. – 2024. – URL:<https://en.wikipedia.org/wiki/Cellular_automaton> (дата обращения: 24.05.2025). – Режим доступа: свободный.
3. Make Random Cave Levels Using Cellular Automata [Электронный ресурс] // Tuts+ Game Development. – URL:<https://code.tutsplus.com/generate-random-cave-levels-using-cellular-automata--gamedev-9664t> (дата обращения: 24.05.2025). – Режим доступа: свободный.
4. Меженин, М. Г. Обзор систем процедурной генерации игр / М. Г. Меженин // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. – 2015. – № 1. – URL:<https://cyberleninka.ru/article/n/obzor-sistem-protsedurnoy-generatsii-igr> (дата обращения: 24.05.2025).
5. Мечтают ли нейроовцы о генеративных трубах? Кратко о процедурной генерации контента (PCG) в контексте игровой индустрии [Электронный ресурс] // Habr. – URL:<https://habr.com/ru/companies/nau_engine/articles/792354/> (дата обращения: 20.04.2025). – Режим доступа: свободный.
6. Случайное блуждание [Электронный ресурс] // Википедия. – URL: [https://ru.wikipedia.org/wiki/Случайное\_блуждание](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D1%87%D0%B0%D0%B9%D0%BD%D0%BE%D0%B5_%D0%B1%D0%BB%D1%83%D0%B6%D0%B4%D0%B0%D0%BD%D0%B8%D0%B5) (дата обращения: 24.05.2025). – Режим доступа: свободный.
7. Вивденко, М. В. Процедурная генерация уровней в играх [Электронный ресурс] // Донецкий национальный технический университет. – 2018. – URL:<https://masters.donntu.ru/2018/fknt/vivdenko/library/article2.htm> (дата обращения: 24.05.2025).
8. Яблоков, Е. В. Методы генерации виртуальных миров в компьютерных играх [Электронный ресурс] // Донецкий национальный технический университет. – 2015. – URL:<https://masters.donntu.ru/2015/frt/yablokov/library/article6.htm> (дата обращения: 24.05.2025).