ПРОЕКТНАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: «Исследование и визуализация алгоритмов генерации и прохождения лабиринтов»

Выполнили:

Студенты группы ИКПИ-15

2 курса факультета ИКСС

Баранов С.Н.

Смирнов С.В.

Тарасова Д.В.

Целоусов Г.А.

Преподаватель:

Дагаев А.В.

Оглавление

[Введение 4](#_Toc138611859)

[1.1. Общая информация 5](#_Toc138611860)

[1.1.1. Постановка задачи и особенности среды разработки 5](#_Toc138611861)

[Постановка задачи: 5](#_Toc138611862)

[Выбор среды разработки: 5](#_Toc138611863)

[1.1.2. Характеристика языка программирования 6](#_Toc138611864)

[1.2. Генерация лабиринта 7](#_Toc138611865)

[1.2.1. Сравнительная характеристика алгоритмов генерации 7](#_Toc138611866)

[1.2.2. Обоснование выбора алгоритма 9](#_Toc138611867)

[1.2.3. Принцип действия алгоритмов 10](#_Toc138611868)

[Recursive Backtracking: 10](#_Toc138611869)

[Алгоритм Эллера 11](#_Toc138611870)

[1.3. Прохождение лабиринта 13](#_Toc138611871)

[1.3.1. Алгоритм Ли (волновой алгоритм) 13](#_Toc138611872)

[1.3.2. Алгоритм следования вдоль стены 14](#_Toc138611873)

[2.1. Исследование алгоритмов 15](#_Toc138611874)

[2.1.1. Алгоритмы построения лабиринтов 15](#_Toc138611875)

[Анализ временной сложности 15](#_Toc138611876)

[Анализ объемной сложности 15](#_Toc138611877)

[Эмпирическое исследование 15](#_Toc138611878)

[2.1.2. Алгоритмы прохождения лабиринтов 20](#_Toc138611879)

[Анализ временной сложности 20](#_Toc138611880)

[Анализ объемной сложности 20](#_Toc138611881)

[Эмпирическое исследование 20](#_Toc138611882)

[2.2. Разработка лабиринта в графической среде 25](#_Toc138611883)

[Разработка структуры лабиринта 25](#_Toc138611884)

[2.3. Разработка механизмов поиска пути в лабиринте 27](#_Toc138611885)

[Разработка структуры поиска и отображения пути 27](#_Toc138611886)

[2.4. Реализация многопоточности 28](#_Toc138611887)

[2.5. Описание классов, методов и функций приложения 29](#_Toc138611888)

[Класс Cell 29](#_Toc138611889)

[Класс MazeGeneratorCell 29](#_Toc138611890)

[Класс Maze 29](#_Toc138611891)

[Класс Floor 30](#_Toc138611892)

[Класс MazeGenerator 30](#_Toc138611893)

[Класс HintRenderer 32](#_Toc138611894)

[Класс PlayerControls 32](#_Toc138611895)

[Класс CameraFollows 33](#_Toc138611896)

[2.6. Руководство пользователя 34](#_Toc138611897)

[Установка и запуск проекта: 34](#_Toc138611898)

[Инструкция пользователя: 35](#_Toc138611899)

[Выводы по проделанной работе 39](#_Toc138611900)

# Введение

Данная работа посвящена генерации лабиринтов и поиску маршрутов выхода из них. В данном руководстве содержится вся необходимая информация касательно реализации алгоритмов построения и прохождения лабиринтов, а также вспомогательная информация касательно задействованных при разработке технологий.

# Общая информация

## Постановка задачи и особенности среды разработки

### Постановка задачи:

В ходе данной работы необходимо достигнуть следующих целей:

1. Изучить алгоритмы генерации двумерных лабиринтов;
2. Изучить алгоритмы построения маршрута выхода из лабиринтов;
3. Изучить способы реализации многопоточного приложения;
4. Реализовать построение лабиринта в графической среде;
5. Реализовать построение маршрута выхода из лабиринта в графической среде.

### Выбор среды разработки:

В качестве графической среды для построения лабиринта был выбран игровой движок Unity. Выбор обусловлен следующими преимуществами:

1. Доступность. Среда разработки Unity является кроссплатформенным бесплатным приложением, что облегчает взаимодействие с проектом;
2. Гибкость. Данный движок обладает множеством встроенных утилит и инструментов, что позволяет упростить использование приложения;
3. Наличие визуальной среды разработки. Наличие таких возможностей, как интегрированная среда разработки и цепочка сборки, позволяет увеличить производительность разработчиков;
4. Модульная система компонентов. Организация работы в виде объединения функциональных блоков, а не наследования, упрощает работу с объектами.

В качестве среды разработки программного кода была выбрана Microsoft Visual Studio, ввиду следующих преимуществ:

1. Гибкость. За счет использования модульных языковых пакетов можно осуществить узкоспециализированную разработку на конкретных языках программирования;
2. Интегрируемость с Unity. Из-за наличия специализированных модулей языка программирования C# данная среда разработки позволяет осуществлять прямое редактирования файлов скриптов Unity;

## Характеристика языка программирования

В качестве языка программирования был выбран язык C#. Далее приведён ряд преимуществ его использования:

1. Использование в среде разработки Unity. C# является основным языком разработки на движке Unity;
2. C-подобный синтаксис. Схожесть данного языка программирования с языками C и C++ упрощает разработку на нем;
3. Возможность использования объектно-ориентированного подхода;
4. Независимость от аппаратного функционала.

# Генерация лабиринта

## Сравнительная характеристика алгоритмов генерации

Идеальный лабиринт – лабиринт без петель и без недостижимых областей; из каждой точки существует только один путь к каждой другой точке в лабиринте.

В таблице 1 представлена сравнительная характеристика ряда наиболее распространенных алгоритмов генерации лабиринтов.

*Таблица 1. Сравнительная характеристика алгоритмов генерации лабиринтов*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Алгоритм** | **% тупиков** | **Тип** | **Приоритет** | **Смещенность** | **Однородность** | **Память** |
| Одномаршрутный | 0 | дерево | стены | Нет | никогда | N^2 |
| Recursive backtracking | 10 | дерево | проходы | Нет | никогда | N^2 |
| Рекурсивное деление | 23 | дерево | стены | Нет | никогда | N |
| Двоичное дерево | 25 | множество | оба | Да | никогда | 0 |
| Алгоритм Эллера | 28 | множество | оба | Да | нет | N |
| Алгоритм Олдоса-Бродера | 29 | дерево | оба | Нет | да | 0 |
| Алгоритм Краскала | 30 | множество | оба | Нет | нет | N^2 |
| Алгоритм Прима | 32 | дерево | оба | Нет | нет | N^2 |

**Тупики.**Это приблизительный процент ячеек, являющихся тупиками в лабиринте. При достаточно высоком показателе проходов количество тупиков Recursive Backtracking может становиться ниже 1%. Наибольший вероятный процент тупиков в двухмерном ортогональном идеальном лабиринте равен 66% — это одномаршрутный проход с кучей тупиков единичной длины по обеим сторонам от него.

**Тип.**Существует два типа алгоритмов создания идеальных лабиринтов:

* Алгоритм на основе дерева выращивает лабиринт подобно дереву, всегда добавляя к тому, что уже есть, и на каждом этапе имея правильный идеальный лабиринт.
* Алгоритм на основе множеств выполняет построения там, где ему хочется, отслеживая части лабиринта, соединённые друг с другом, чтобы соединить всё и создать правильный лабиринт на момент завершения.

**Приоритет.**Большинство алгоритмов можно реализовать или как вырезание проходов, или как добавление стен. Очень немногие можно реализовать только как один или другой подход. В одномаршрутных лабиринтах всегда используется добавление стен, потому что в них задействуется разбиение проходов стенами на две части, однако базовый лабиринт можно создать любым способом. Recursive Backtracking нельзя реализовать с добавлением стен, потому что в этом случае он склонен создавать путь решения, который следует вдоль внешнего края, где вся внутренняя часть лабиринта соединена с границей единственным проходом. Рекурсивное деление нельзя использовать для вырезания проходов, потому что это приводит к созданию очевидного решения, которое или следует вдоль внешнего края, или иначе напрямую пересекает внутреннюю часть.

**Отсутствие смещенности.**Одинаково ли воспринимает алгоритм все направления и стороны лабиринта так, что последующий анализ лабиринта не может обнаружить никакой смещенности проходов. Алгоритм двоичного дерева чрезвычайно смещён, в нём легко перемещаться в один угол и сложно в противоположный. Алгоритм Эллера склонен к созданию проходов, приблизительно синхронизируя начальные или конечные края.

**Однородность**. Генерирует ли алгоритм все возможные лабиринты с равной вероятностью.

* «Да» означает, что алгоритм полностью однороден.
* «Нет» означает, что алгоритм потенциально может генерировать все возможные лабиринты в пределах любого пространства, но не с равной вероятностью.
* «Никогда» означает, что существуют возможные лабиринты, которые алгоритм никогда не сможет сгенерировать. Учтите, что только алгоритмы с полным отсутствием смещенности могут быть полностью однородными.

**Память**. Объём дополнительной памяти или стека, необходимый для реализации алгоритма. Эффективные алгоритмы требуют только битовой карты самого лабиринта, в то время как другие требуют объёма памяти, пропорционального одной строке (N), или пропорционального количеству ячеек (N2). Некоторым алгоритмам даже не нужно иметь в памяти весь лабиринт, и они могут добавлять части лабиринта бесконечно. Алгоритму Эллера нужен объём памяти для хранения строки, но большего ему не требуется, потому что достаточно хранить только одну строку лабиринта. Для рекурсивного деления требуется стек объёмом вплоть до размера строки, но ему не нужно смотреть на битовую карту лабиринта.

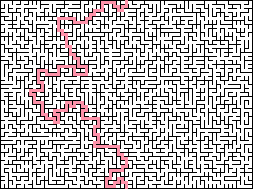
## Обоснование выбора алгоритма

Методом сравнительного анализа различных алгоритмов генерации лабиринтов были выбраны алгоритмы Recursive backtracking и Эллера, так как они гарантирует достаточный процент тупиков и не генерирует слишком простые лабиринты. Также оба алгоритма генерируют лабиринты путем удаления стен, что обобщает изначальную генерацию основы лабиринта.

## Принцип действия алгоритмов

### Recursive Backtracking:

Алгоритм требует стека, объём которого может доходить до размеров лабиринта. При вырезании он ведёт себя максимально жадно, и всегда вырезает проход в несозданной части, если она существует рядом с текущей ячейкой. Каждый раз, когда мы перемещаемся к новой ячейке, записываем предыдущую ячейку в стек. Если рядом с текущей позицией нет несозданных ячеек, то извлекаем из стека предыдущую позицию. Лабиринт завершён, когда в стеке больше ничего не остаётся. Это приводит к созданию лабиринтов с максимальным показателем текучести, тупиков меньше, но они длиннее, а решение обычно оказывается очень долгим и извилистым. При правильной реализации он выполняется быстро, и быстрее работают только очень специализированные алгоритмы. Recursive backtracking не может работать с добавлением стен, потому что обычно приводит к пути решения, следующему по внешнему краю, когда вся внутренняя часть лабиринта соединена с границей одним проходом.



*Рисунок 1* – *Лабиринт, сгенерированный методом Recursive Backtracking*

### Алгоритм Эллера

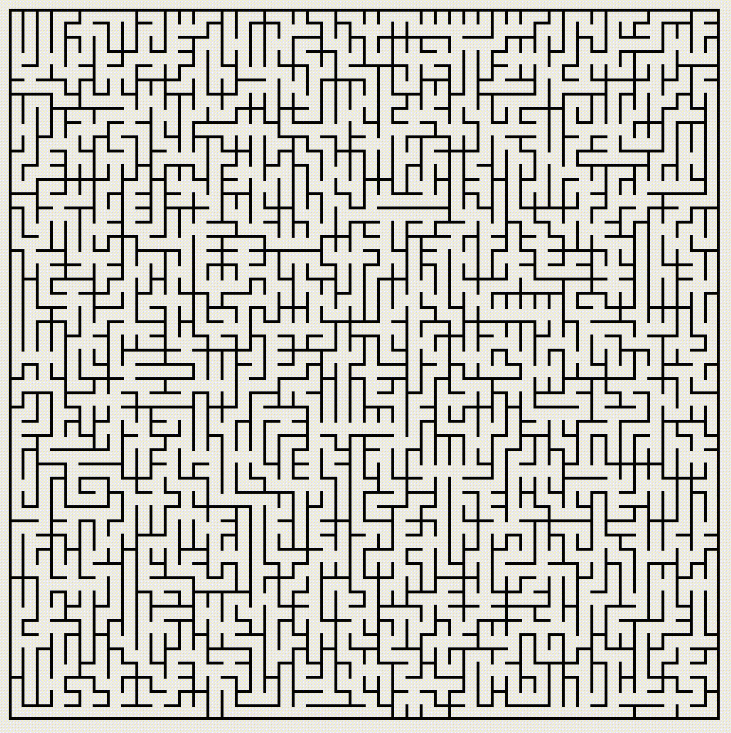
Алгоритм Эллера позволяет создавать лабиринты, имеющие только один путь между двумя точками. Сам по себе алгоритм очень быстр и использует память эффективнее, чем другие популярные алгоритмы (такие как Prim и Kruskal), требуя памяти пропорционально числу строк. Это позволяет создавать лабиринты большого размера при ограниченных размерах памяти.

Лабиринт может храниться в файле в виде количества строк и столбцов, а также двух матриц, содержащих положение вертикальных и горизонтальных стен соответственно. В первой матрице отображается наличие стены справа от каждой ячейки, а во второй - снизу.

Описание алгоритма

1. Создайте первую строку. Ни одна ячейка не будет являться частью ни одного множества.
2. Присвойте ячейкам, не входящим в множество, свое уникальное множество.
3. Создайте правые границы, двигаясь слева направо:
   1. Случайно решите добавлять границу или нет
      1. Если текущая ячейка и ячейка справа принадлежат одному множеству, то создайте границу между ними (для предотвращения зацикливаний)
      2. Если вы решили не добавлять границу, то объедините два множества в которых находится текущая ячейка и ячейка справа.
4. Создайте границы снизу, двигаясь слева направо:
   1. Случайно решите добавлять границу или нет. Убедитесь, что каждое множество имеет хотя бы одну ячейку без нижней границы (для предотвращения изолирования областей)
      1. Если ячейка в своем множестве одна, то не создавайте границу снизу
      2. Если ячейка одна в своем множестве без нижней границы, то не создавайте нижнюю границу
5. Решите, будете ли вы дальше добавлять строки или хотите закончить лабиринт
   1. Если вы хотите добавить еще одну строку, то:
      1. Выведите текущую строку
      2. Удалите все правые границы
      3. Удалите ячейки с нижней границей из их множества
      4. Удалите все нижние границы
      5. Продолжайте с шага 2
   2. Если вы решите закончить лабиринт, то:
      1. Добавьте нижнюю границу к каждой ячейке
      2. Двигаясь слева направо:
         1. Если текущая ячейка и ячейка справа члены разных множеств, то:
            1. Удалите правую границу
            2. Объедините множества текущей ячейки и ячейки справа
            3. Выведите завершающую строку

Ниже представлен пример лабиринта, сгенерированного с помощью алгоритма Эллера (Рис. 2).



*Рисунок 2 – Лабиринт, сгенерированный алгоритмом Эллера*

# Прохождение лабиринта

## Алгоритм Ли (волновой алгоритм)

Волновой алгоритм – это алгоритм, который позволяет вам найти минимальный путь на графе. В основе этого метода лежит алгоритм поиска по ширине. Алгоритм предназначен для поиска кратчайшего пути от стартовой ячейки к конечной ячейке, если это возможно, либо, при отсутствии пути, выдать сообщение о непроходимости.

Работа алгоритма включает в себя три этапа: инициализацию, распространение волны и восстановление пути.

Во время инициализации строится образ множества ячеек обрабатываемого поля, каждой ячейке приписываются атрибуты проходимости/непроходимости, запоминаются стартовая и финишная ячейки. Далее, от стартовой ячейки порождается шаг в соседнюю ячейку, при этом проверяется, проходима ли она, и не принадлежит ли ранее меченной в пути ячейке. Соседние ячейки принято классифицировать двояко: в смысле окрестности Мура и окрестности фон Неймана, отличающийся тем, что в окрестности фон Неймана соседними ячейками считаются только 4 ячейки по вертикали и горизонтали, в окрестности Мура — все 8 ячеек, включая диагональные. В данном проекте используется окрестность фон Неймана.

При выполнении условий проходимости и непринадлежности её к ранее помеченным в пути ячейкам, в атрибут ячейки записывается число, равное количеству шагов от стартовой ячейки, на первом шаге это будет 1. Каждая ячейка, меченная числом шагов от стартовой ячейки, становится стартовой и из неё порождаются очередные шаги в соседние ячейки. Очевидно, что при таком переборе будет найден путь от начальной ячейки к конечной, либо очередной шаг из любой порождённой в пути ячейки будет невозможен.

Восстановление кратчайшего пути происходит в обратном направлении: при выборе ячейки от финишной ячейки к стартовой на каждом шаге выбирается ячейка, имеющая атрибут расстояния от стартовой на единицу меньше текущей ячейки. Очевидно, что таким образом находится кратчайший путь между парой заданных ячеек.

## Алгоритм следования вдоль стены

Одним из наиболее широко известных методов поиска решения лабиринтов является метод следования вдоль стены, также известный как «правило левой/правой руки». В данном проекте было выбрано следование вдоль правой стены. Этот метод основан на внешней связности лабиринта — все стены должны быть соединены с внешней границей лабиринта. Если это так, то всегда можно найти выход из лабиринта, непрерывно следуя либо по левой, либо по правой стороне на протяжении всего лабиринта. Однако в тех случаях, когда не все стены соединены с внешними границами, этот метод не всегда будет находить решение. Этот метод/алгоритм полезен, если нам известно, что стены лабиринта связаны между собой. Алгоритм очень эффективен, так как не требует маркировки развилок или перезапуска при неудачных попытках; алгоритм просто следует по левой или правой стороне лабиринта.

# 2.1. Исследование алгоритмов

В ходе работы был проведен анализ временной и объемной сложностей алгоритмов построение и прохождения лабиринтов, а также было проведено эмпирическое исследование зависимости времени выполнения алгоритмов от количества входных данных (размера лабиринта). Далее приведены результаты данных исследований.

## 2.1.1. Алгоритмы построения лабиринтов

### Анализ временной сложности

В ходе анализа временной сложности алгоритмов построения лабиринтов были получены следующие результаты:

* Алгоритм Recursive Backtracking – ;
* Алгоритм Эллера – ;

Получены примерно равные результаты для исследуемых алгоритмов. Как и большинство алгоритмов построения лабиринтов они имеют квадратичную временную сложность.

### Анализ объемной сложности

В ходе анализа объемной сложности алгоритмов построения лабиринтов были получены следующие результаты:

* Алгоритм Recursive Backtracking – ;
* Алгоритм Эллера – ;

Получены различные результаты, т.к. Recursive Backtracking в определенном случае может понадобиться хранить информацию вплоть до ячеек (весь лабиринт). В свою очередь, алгоритм Эллера генерирует лабиринт построчно, и в каждый момент времени ему достаточно хранить лишь информацию о ячейках лишь одной строки.

### Эмпирическое исследование

В ходе эмпирического исследования алгоритмов построения лабиринтов были получены следующие результаты, отражающие время выполнения алгоритмов в зависимости от размера лабиринта.

*Таблица 2 – Время выполнения алгоритмов построения лабиринтов*

|  |  |  |
| --- | --- | --- |
| N | Recursive Backtracking | Eller |
| 10 | 44,85678256 | 44,1081 |
| 12 | 57,66062396 | 58,38043 |
| 14 | 93,32378828 | 83,18215 |
| 16 | 142,0470034 | 122,4062 |
| 18 | 206,8141486 | 188,4739 |
| 20 | 318,37795 | 282,6026 |
| 22 | 481,0004376 | 413,8553 |
| 24 | 683,9300732 | 592,6149 |
| 26 | 958,4130582 | 825,9483 |
| 28 | 1294,477279 | 1122,672 |
| 30 | 1708,574657 | 1492,561 |
| 32 | 2198,858963 | 1931,196 |
| 34 | 2782,605895 | 2456,777 |
| 36 | 3463,057496 | 3069,799 |
| 38 | 4254,475389 | 3777,677 |
| 40 | 5159,806255 | 4595,03 |
| 42 | 6202,197143 | 5521,961 |
| 44 | 7368,015152 | 6562,541 |
| 46 | 8675,049177 | 7735,522 |
| 48 | 10123,67122 | 9039,515 |
| 50 | 11724,30779 | 10484,94 |
| 52 | 13501,99761 | 12081,49 |
| 54 | 15451,58436 | 13825,3 |
| 56 | 17573,97015 | 15732,46 |
| 58 | 19896,84944 | 17810,91 |
| 60 | 22414,84876 | 20061,29 |
| 62 | 25134,25932 | 22503,54 |
| 64 | 28064,78874 | 25128,77 |
| 66 | 31220,39792 | 27955,01 |
| 68 | 34602,06777 | 30991,93 |
| 70 | 38221,45777 | 34236,08 |
| 72 | 42075,75713 | 37704,77 |
| 74 | 46176,58362 | 41400,8 |
| 76 | 50547,12439 | 45328,87 |
| 78 | 55173,68497 | 49496,19 |
| 80 | 60091,17214 | 53913,38 |
| 82 | 65293,42452 | 58584,51 |
| 84 | 70774,74675 | 63518,95 |
| 86 | 76551,65037 | 68721,42 |
| 88 | 82646,88209 | 74204,73 |
| 90 | 89060,81435 | 79964,98 |
| 92 | 95798,00588 | 86017,14 |
| 94 | 102856,6089 | 92376,5 |
| 96 | 110256,0573 | 99037,76 |
| 98 | 118018,891 | 106017,3 |
| 100 | 126132,8525 | 113311,5 |

На рис. 3, 4, 5 представлены графики зависимости времени выполнения алгоритмов от размера лабиринта. Эти графики подтверждают результаты, полученные в ходе анализа алгоритмов.

*Рисунок 3 – Графики зависимости времени выполнения алгоритмов построения лабиринтов от их размеров*

*Рисунок 4 – График зависимости времени выполнения алгоритма Recursive Backtracking от размера лабиринта*

*Рисунок 5 – График зависимости времени выполнения алгоритма Эллера от размера лабиринта*

Несмотря на то, что в ходе анализа были получены одинаковые верхние оценки сложности алгоритмов (Big O) одинаковые – – графики для этих алгоритмов различаются, т.к. они имеют разную асимптотическую сложность Θ (Имеют разные коэффициенты при и)

## 2.1.2. Алгоритмы прохождения лабиринтов

### Анализ временной сложности

В ходе анализа временной сложности алгоритмов прохождения лабиринтов были получены следующие результаты:

* Алгоритм Ли – ;
* Алгоритм правой руки – ;

Временные сложности алгоритмов приближаются к квадратичной. Как правило, алгоритмы прохождения лабиринтов имеют квадратичную сложность.

### Анализ объемной сложности

В ходе анализа объемной сложности алгоритмов прохождения лабиринтов были получены следующие результаты:

* Алгоритм Ли – ;
* Алгоритм правой руки – ;

Исследуемые алгоритмы практически не требуют дополнительной памяти. Объемная сложность алгоритмов – константная.

### Эмпирическое исследование

В ходе эмпирического исследования алгоритмов прохождения лабиринтов были получены следующие результаты, отражающие время выполнения алгоритмов в зависимости от размера лабиринта.

*Таблица 3 – Время выполнения алгоритмов прохождения лабиринтов*

|  |  |  |
| --- | --- | --- |
| N | Lee | Wall |
| 10 | 27,16890323 | 39,41663 |
| 12 | 36,27847664 | 58,69835 |
| 14 | 58,55824781 | 94,16723 |
| 16 | 100,6445167 | 152,221 |
| 18 | 165,0746306 | 267,681 |
| 20 | 267,4186107 | 428,3426 |
| 22 | 416,073862 | 646,4893 |
| 24 | 620,2189956 | 959,1255 |
| 26 | 878,5620553 | 1353,91 |
| 28 | 1210,981523 | 1840,452 |
| 30 | 1612,2466 | 2458,25 |
| 32 | 2098,234492 | 3201,176 |
| 34 | 2674,55953 | 4073,885 |
| 36 | 3350,780898 | 5096,527 |
| 38 | 4143,595848 | 6290,23 |
| 40 | 5051,887111 | 7660,473 |
| 42 | 6079,105622 | 9199,242 |
| 44 | 7241,788687 | 10938,37 |
| 46 | 8542,249264 | 12889,54 |
| 48 | 9992,38201 | 15070,75 |
| 50 | 11596,80252 | 17472,63 |
| 52 | 13363,52131 | 20132,11 |
| 54 | 15299,8528 | 23044,73 |
| 56 | 17418,53846 | 26239,49 |
| 58 | 19729,93935 | 29703,77 |
| 60 | 22231,82576 | 33466,11 |
| 62 | 24943,91627 | 37525,94 |
| 64 | 27859,93964 | 41903,39 |
| 66 | 31005,78608 | 46608,17 |
| 68 | 34371,32855 | 51666,48 |
| 70 | 37979,89492 | 57079,99 |
| 72 | 41833,45897 | 62859,33 |
| 74 | 45936,67643 | 69006,64 |
| 76 | 50297,07515 | 75556,97 |
| 78 | 54930,19894 | 82503,1 |
| 80 | 59837,94542 | 89868,18 |
| 82 | 65025,99155 | 97663,07 |
| 84 | 70509,41562 | 105879,2 |
| 86 | 76287,13793 | 114551,6 |
| 88 | 82373,48296 | 123683,8 |
| 90 | 88782,1635 | 133288,1 |
| 92 | 95507,20293 | 143379,2 |
| 94 | 102570,888 | 153980 |
| 96 | 109973,7662 | 165078,7 |
| 98 | 117721,4682 | 176703,9 |
| 100 | 125828,3424 | 188867,3 |

На рис. 6, 7, 8 представлены графики зависимости времени выполнения алгоритмов от размера лабиринта. Эти графики подтверждают результаты, полученные в ходе анализа алгоритмов.

*Рисунок 6 – Графики зависимости времени выполнения алгоритмов прохождения лабиринтов от их размеров*

*Рисунок 7 – График зависимости времени выполнения алгоритма Ли от размера лабиринта*

*Рисунок 8 – График зависимости времени выполнения алгоритма правой руки от размеров лабиринта*

Несмотря на то, что в ходе анализа были получены одинаковые верхние оценки сложности алгоритмов (Big O) одинаковые – – графики для этих алгоритмов различаются, т.к. они имеют разную асимптотическую сложность Θ (Имеют разные коэффициенты при и)

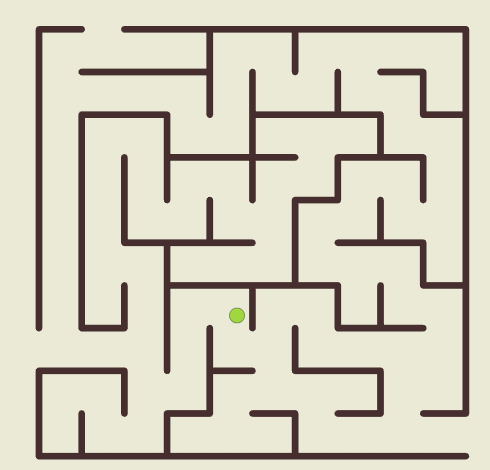
# 2.2. Разработка лабиринта в графической среде

## Разработка структуры лабиринта

Лабиринт является множеством клеток. Клетка - наименьшая структурная единица лабиринта. В целях оптимального распределения памяти клетка содержит в себе только левую и нижнюю стены.



*Рисунок 9* – *Клетка*



*Рисунок 10* – *Пример лабиринта*

Лабиринт представлен в виде двумерного массива клеток, с помощью которого осуществляется взаимодействие с отображаемым на экране лабиринтом.

Стартом в лабиринте всегда является левый нижний угол (с координатами (0, 0)). Выходы генерируются случайно, однако один из трех выходов гарантированно является самой удаленной клеткой от старта. Для перемещения по лабиринту реализован объект-игрок, управление которым осуществляется с клавиатуры.

Были разработаны следующие структуры лабиринтов:

* идеальный лабиринт с тупиками (с 3 выходами)
* лабиринт с циклами (с 3 выходами)

# 2.3. Разработка механизмов поиска пути в лабиринте

## Разработка структуры поиска и отображения пути

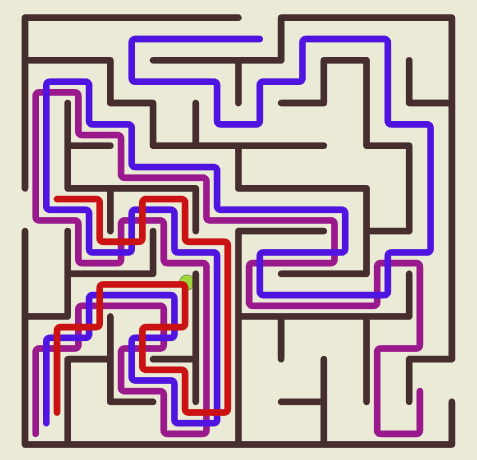
Поиск пути в лабиринте осуществляется соответствующими алгоритмами (см. подробное описание в п. 2.3). Поиск пути к каждому выходу происходит в отдельном потоке.

Для отображения процесса работы алгоритма реализована структура Floor, “закрашивающая” клетки, по которым проходит алгоритм.



*Рисунок 11 – Пример “закрашенных” клеток*

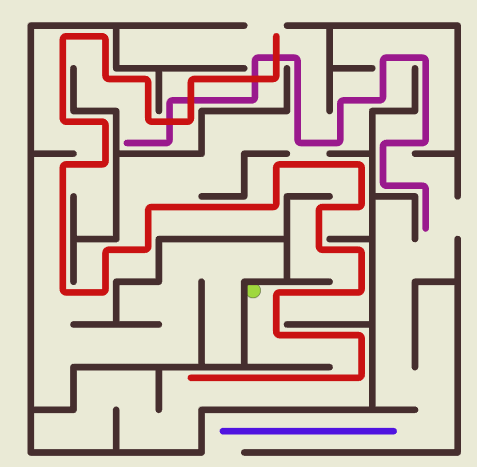
Отображение пути в лабиринте реализовано при помощи встроенного компонента Unity – LineRenderer. Он отображает линию-путь от начала лабиринта до выхода из него, полученную в результате работы алгоритма поиска маршрута. Визуализация путей происходит параллельно благодаря многопоточности.



*Рисунок 12 – Лабиринт с отрисованными линиями-путями*

# 2.4. Реализация многопоточности

Многопоточность реализована с помощью встроенного механизма Unity – Coroutine (сопрограмма). Для его работы все функции поиска пути имеют специальный тип IEnumerator. Благодаря этому механизму параллельно отрисовывается 3 пути, а также реализуются паузы в коде (при помощи ключевого слова yield).



*Рисунок 13 – Демонстрация многопоточности (каждая линия генерируется в своем потоке)*

# 2.5. Описание классов, методов и функций приложения

## Класс Cell

Класс Cell : MonoBehaviour – класс, характеризующий клетку как игровой объект.

Атрибуты:

public GameObject WallLeft – левая стенка клетки;

public GameObject WallBottom – нижняя стенка клетки.

## Класс MazeGeneratorCell

Класс MazeGeneratorCell – класс, характеризующий клетку как логический объект.

Атрибуты:

public int X – координата клетки по оси Х;

public int Y – координата клетки по оси У;

public bool WallLeft – логическое представление левой стены клетки;

public bool WallBottom – логическое представление нижней стены клетки;

public bool Visited –информация о том, была ли посещена клетка;

public int DistanceFromStart – расстояние от текущей клетки до старта.

## Класс Maze

Класс Maze – класс, характеризующий лабиринт

Атрибуты:

public MazeGeneratorCell[,] cells – двумерный массив, состоящий из клеток лабиринта;

public List<Vector2Int> finishPosition = new List<Vector2Int>(3) – список с координатами финишей.

## Класс Floor

Класс Floor : MonoBehaviour – класс, характеризующий плитку прохождения лабиринта как игровой объект.

Атрибуты:

public GameObject floor – плитка.

## Класс MazeGenerator

Класс MazeGenerator : MonoBehaviour – класс, отвечающий за генерацию и визуализацию лабиринта.

Атрибуты:

public Cell CellPrefab – модель клетки, использующаяся для генерации лабиринта методом копирования;

public Vector3 CellSize – размер клетки в пространстве;

public int Width – ширина лабиринта;

public int Height – длина лабиринта;

public Maze maze – лабиринт;

public List<List<Cell>> cells – двумерный список ссылок на объекты-клетки лабиринта;

public static float MazeRenderTimeout – константа, отвечающая за время паузы при исполнении кода;

public bool Cycles – флаг, отвечающий за создание циклов (петель) в лабиринте;

public bool Eller – флаг, отвечающий за выбор алгоритма генерации лабиринта (true – алгоритм Эллера, false – Recursive Backtracking);

public bool ReadyToSolve – флаг, показывающий готовность лабиринта к решению

public HintRenderer hintRenderer – объект класса HintRenderer, отвечающий за визуализацию прохождения лабиринта;

public HintRenderer hintRenderer1 – объект класса HintRenderer, отвечающий за визуализацию прохождения лабиринта

public HintRenderer hintRenderer2 – объект класса HintRenderer, отвечающий за визуализацию прохождения лабиринта

Методы:

public void Start() – инициализация объекта класса MazeGenerator. Вызывается автоматически при объявлении объекта данного класса;

public void Update() – обновление объекта класса MazeGenerator. Вызвается автоматически при каждом обновлении фрейма. Отвечает за отклик на нажатия клавиш на клавиатуре;

public Maze CreateMaze() – генерация основы лабиринта (лабиринт со всеми стенами, нуждающийся в их удалении);

IEnumerator GenerateMaze() – создание лабиринта алгоритмом Recursive Backtracking

private void RemoveWall(MazeGeneratorCell a, MazeGenerator Cell b) – удаление стены между клетками *a* и *b*. Используется в GenerateMaze() и EllerGenerate();

private void PlaceCycles() – размещение петель лабиринта;

private List<Vector2Int> PlaceMazeExit() – размещение выходов лабиринта. Используется в GenerateMaze() и EllerGenerate();

private void CountDistance() – расчет расстояния от начала лабиринта до каждой его клетки;

public IEnumerator EllerGenerate() – создание лабиринта алгоритмом Эллера;

private void ReplaceSetInRow(int oldSet, int newSet, int rowNum) – Изменение значения множества для элементов строки лабиринта. Используется в EllerGenerate();

private void CreateRow(int rowNum) – генерация строки лабиринта. Используется в EllerGenerate();

private void CreateVerticalConnection(int rowNum) – создвние вертикальных связей между двумя строками лабиринта. Используется в EllerGenerate();

private void CheckLastVertical(int rowNum, bool isAddedVertical) – проверка наличия прохода из текущей строки в следующую. Используется в EllerGenerate();

private void CreateLastRow() – генерация последней строки лабиринта. Используется в EllerGenerate();

## Класс HintRenderer

Класс HintRenderer : MonoBehaviour – класс, отвечающий за поиск и визуализвацию пути.

Атрибуты:

public MazeGenerator MazeSpawner – объект класса MazeGenerator;

private LineRenderer componentLineRenderer – объект класса LineRenderer;

public Floor FloorPrefab – объект класса Floor;

public List<List<Floor>> floors – двумерный список объектов класса Floor;

public static float HintRenderTimeout = 0.1f – константа, отвечающая за время паузы при исполнении кода (для наглядности визуализации).

Методы:

private void Start() – инициализация объекта класса LineRenderer. Вызывается автоматически при объявлении объекта данного класса;

public void Clear() – удаление ранее отрисованных путей с экрана;

public IEnumerator<object> DrawPath(int x, int y) – визуализация пути (не является алгоритмом поиска);

public IEnumerator<object> Lee0(int bx, int by) – алгоритм Ли (волновой алгоритм) поиска пути и последующая визуализация;

public IEnumerator<object> WallPath(int bx, int by) – алгоритм следования вдоль стены и последующая визуализация пути.

## Класс PlayerControls

Класс PlayerControls : MonoBehaviour – класс, отвечающий за управление пользователем курсором (игроком).

Атрибуты:

public float Speed – скорость перемещения игрока;

private RigidBody2D componentRigidBody – физическое «тело» игрока.

Методы:

private void Start() – инициализация объекта класса PlayerControls. Вызывается автоматически при объявлении объекта данного класса. Объект обращается к компоненте RigidBody2D игрового объекта «Player»;

private void Update() – обновление объекта класса PlayerControls. Вызывается автоматически при каждом обновлении фрейма. Объект обновляет данные о передвижении игрока. При нажатии на клавиши WASD или клавиши стрелок происходит движение игрока по лабиринту.

## Класс CameraFollows

Класс CameraFollows : MonoBehaviour – класс, отвечающий за одновременное движение камеры и игрока.

Атрибуты:

private Transform player – игровой объект игрока.

Методы:

void Start() – инициализация объекта класса CameraFollows. Вызывается автоматически при объявлении объекта данного класса. Объект находит игрока;

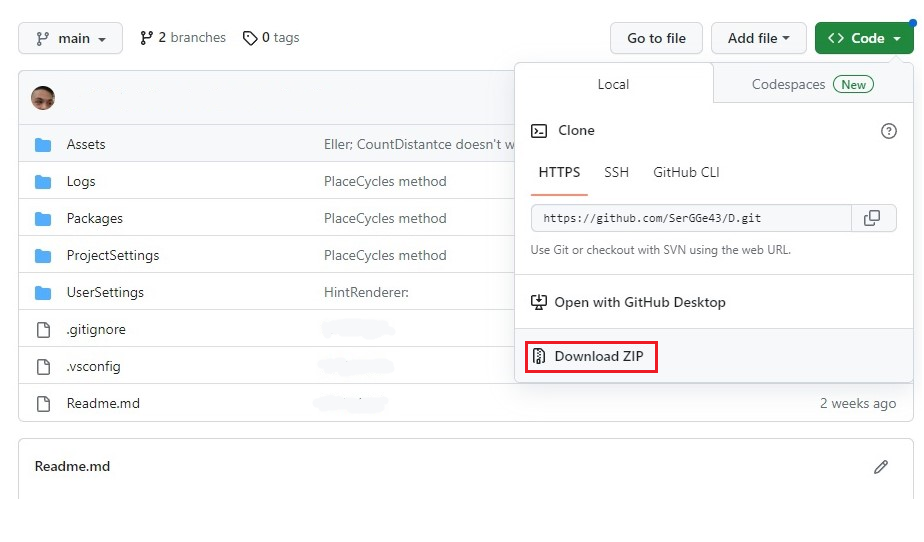
void Update() – обновление объекта класса CameraFollows. Вызывается автоматически при каждом обновлении фрейма. Объект считывает координаты игрока и перемещается в это же место.

# 2.6. Руководство пользователя

### Установка и запуск проекта:

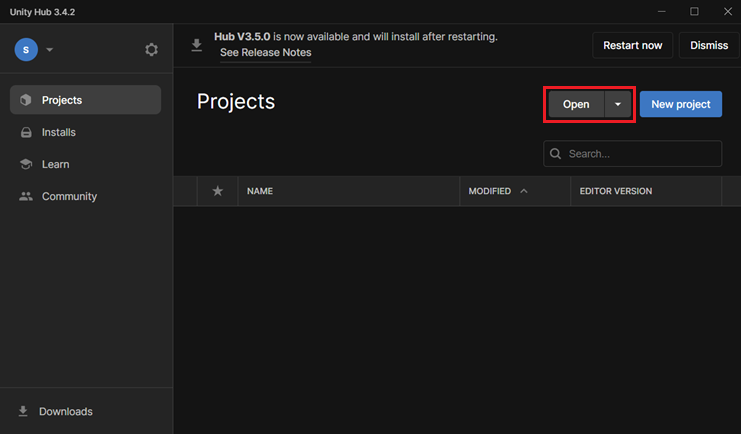
Перед работой с проектом необходимо установить среду разработки Unity. Сделать это можно на официальном сайте, в разделе загрузки (https://unity.com/ru/download). Присутствуют версии для операционных систем Windows, Mac и Linux.

После установки среды разработки необходимо скачать архив, содержащий проект, с платформы GitHub и распаковать его в любое удобное расположение.



*Рисунок 14 – Вид репозитория при скачивании проекта*

Далее необходимо открыть среду Unity и в разделе projects (проекты) выбрать опцию open (открыть).

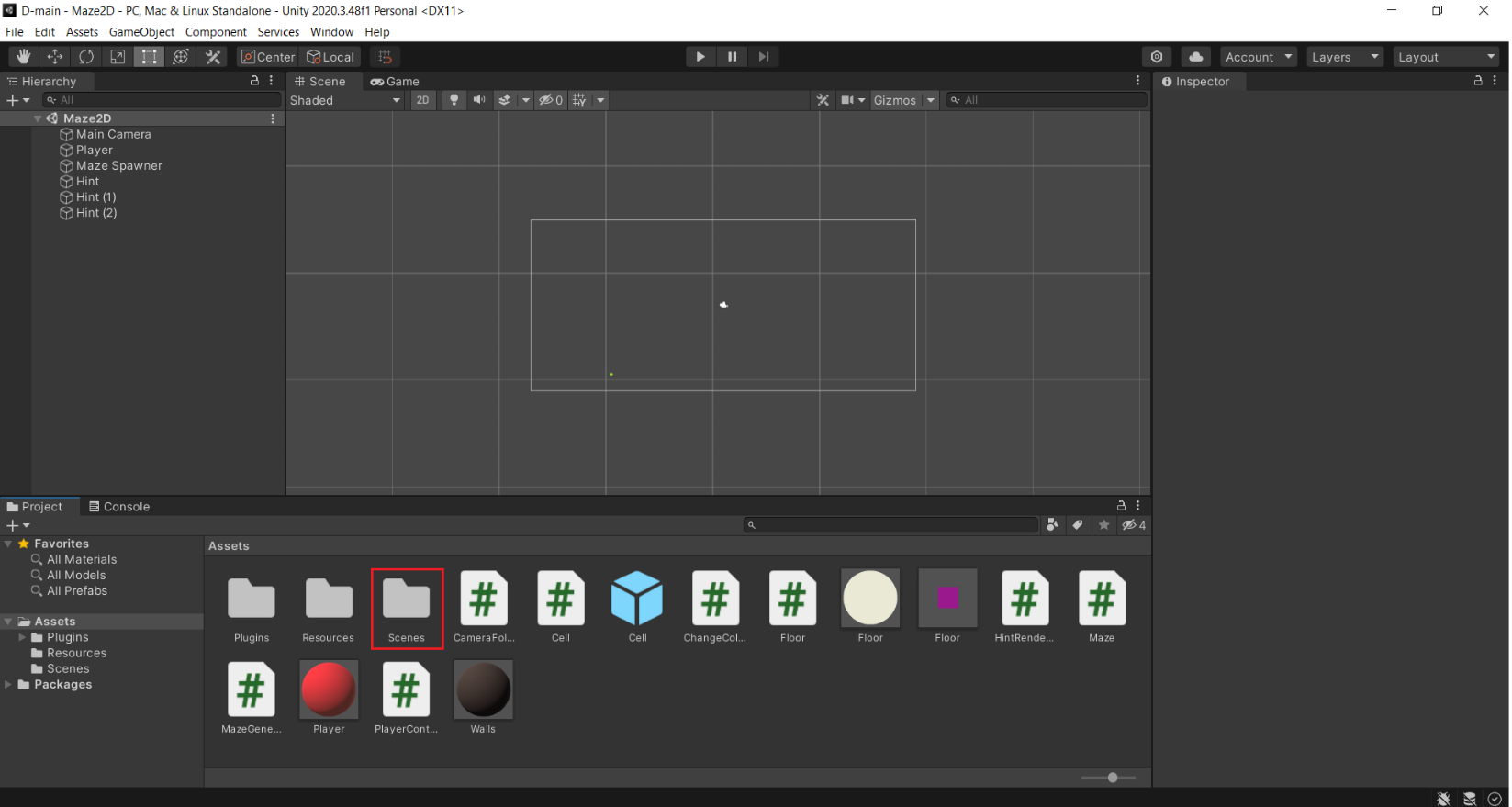


*Рисунок 15 – Окно управления проектами среды Unity*

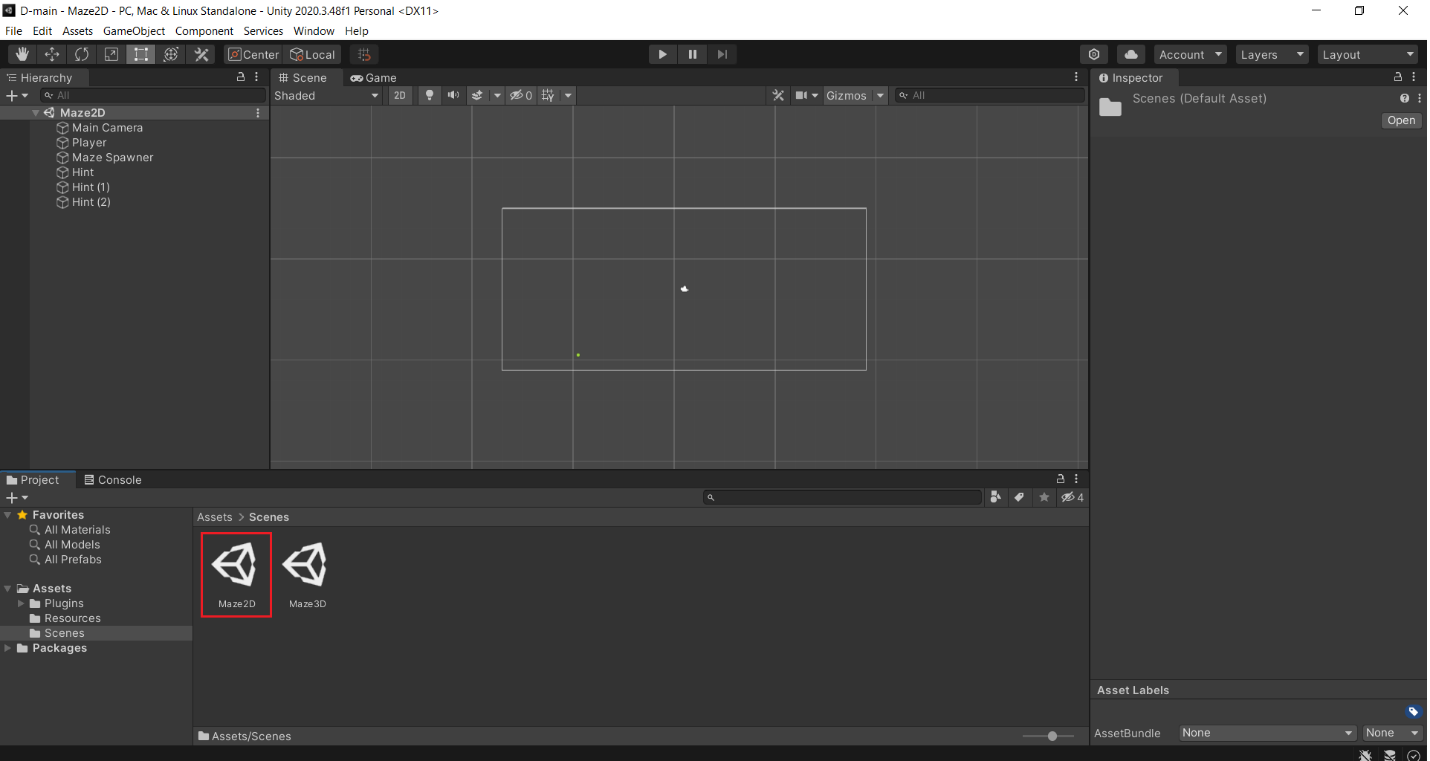
Далее во всплывающем окне необходимо выбрать папку проекта, распакованную из архива и подтвердить действие кнопкой open (открыть).

### Инструкция пользователя:

После запуска проекта необходимо запустить сцену. Для этого переходим папку scenes (сцены) и двойным нажатием открываем сцену Maze2D.

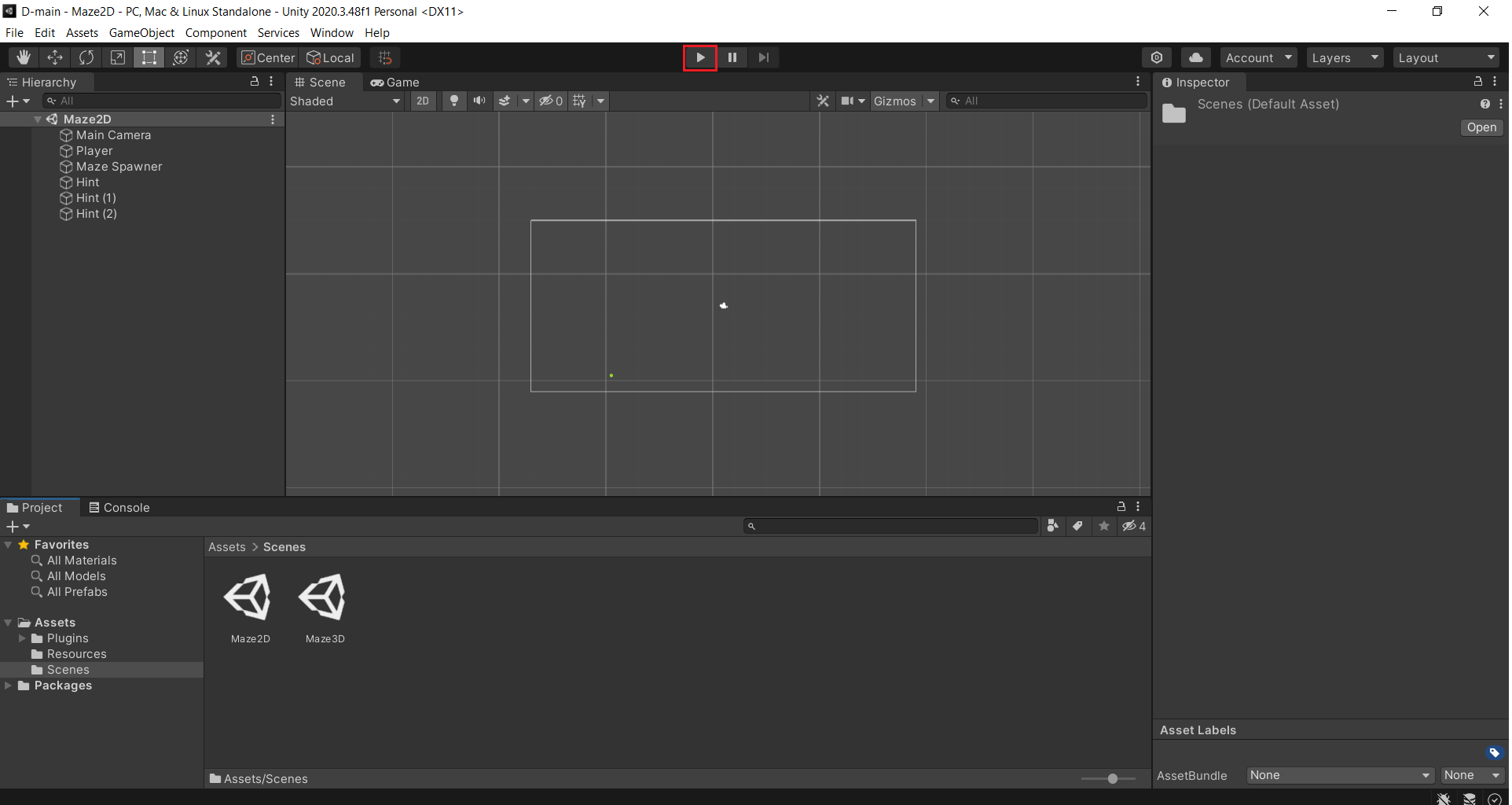


*Рисунок 16 – Выбор папки*

**

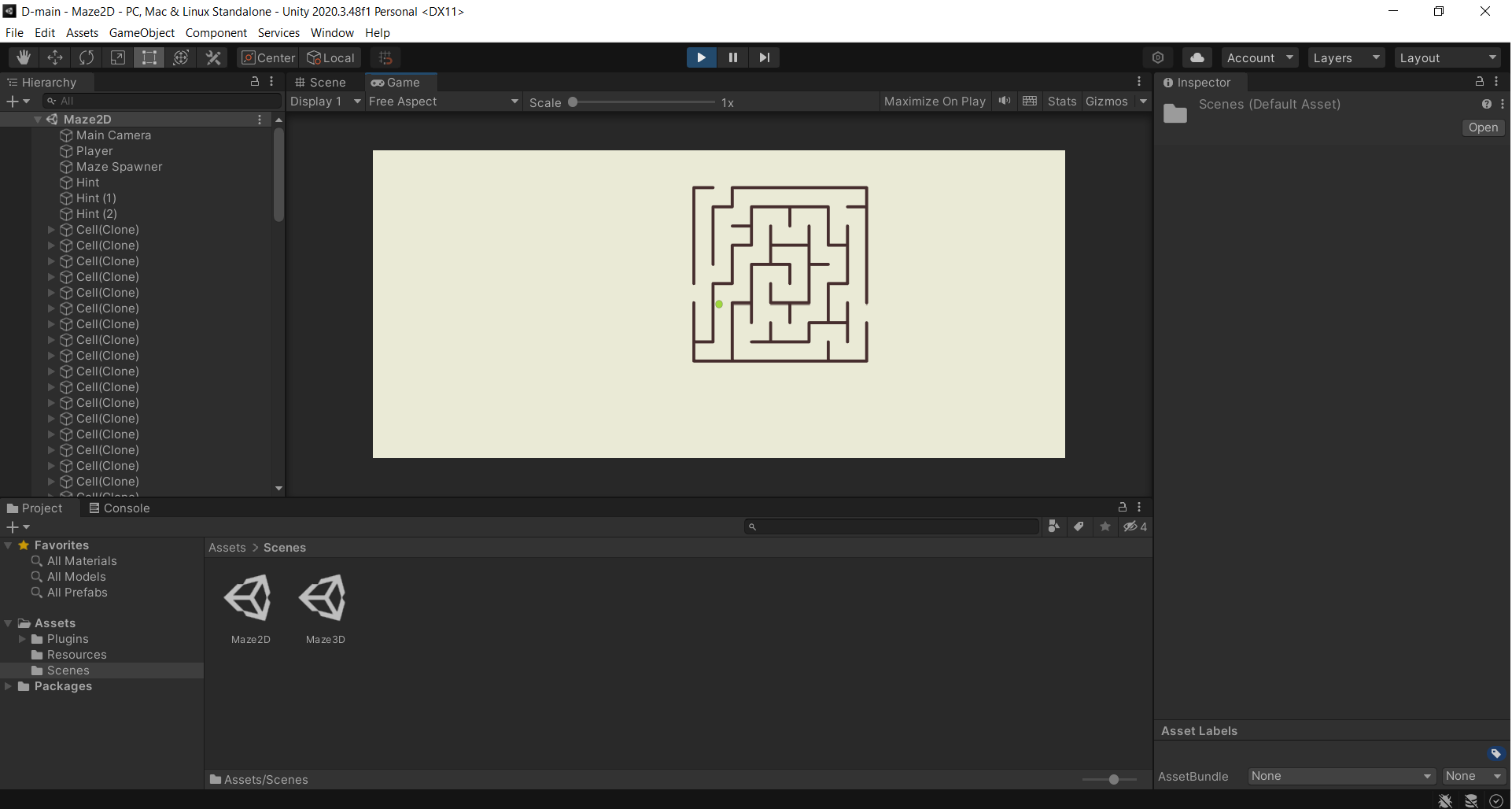
*Рисунок 17 – Выбор сцены*

Далее необходимо запустить выбранную сцену с помощью кнопки в верхней части экрана.



*Рисунок 18 – Запуск сцены*

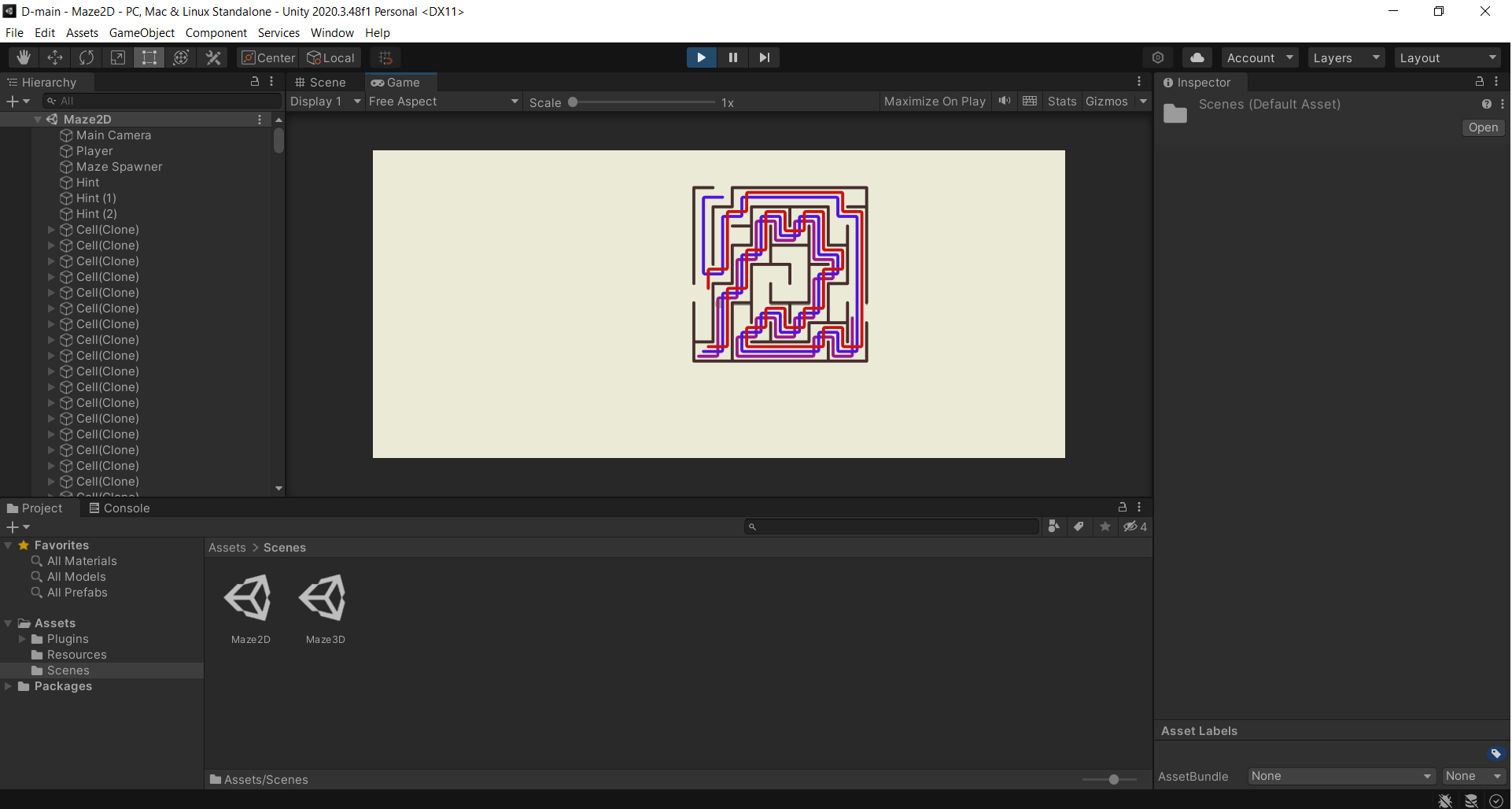
После этого на экране в реальном времени отрисуется лабиринт и появится точка-пользователь.



*Рисунок 19 – Отрисованный лабиринт*

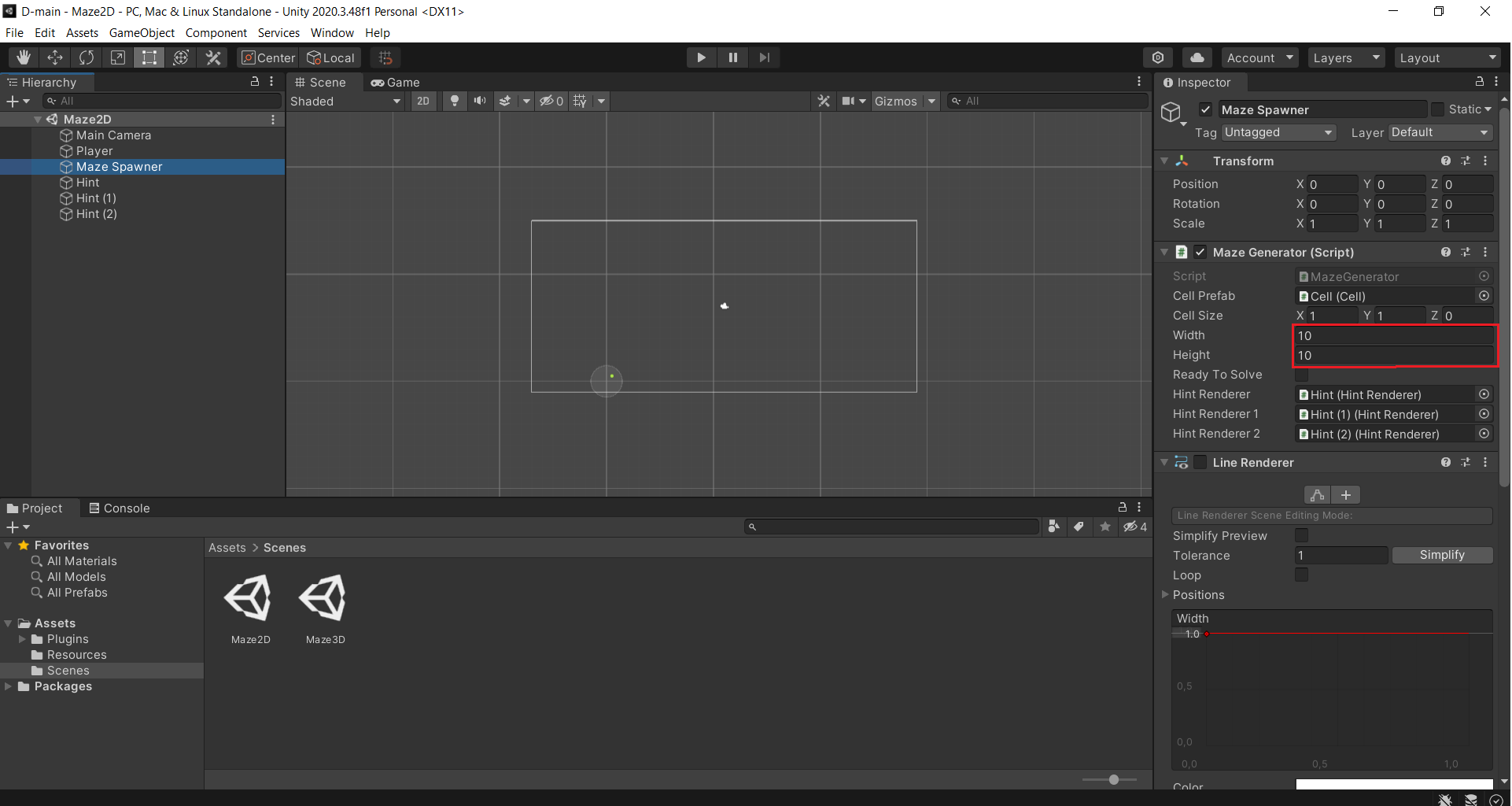
При этом пользователь может перемещаться по лабиринту при помощи клавиш-«стрелочек» на клавиатуре.

Для запуска поиска выхода из лабиринта необходимо нажать клавишу L.



*Рисунок 20 – Лабиринт со сгенерированными маршрутами выхода*

Генерация лабиринтов большего размера возможна путем изменения параметров width и height у объекта Maze Spawner



*Рисунок 21 – Изменение размера лабиринта*

# Выводы по проделанной работе

В ходе работы над данным проектом были получены следующие результаты:

1. Изучены и реализованы алгоритмы генерации лабиринтов
2. Освоены методы нахождения выхода из лабиринта
3. Изучены способы работы со средой разработки Unity в сфере визуализации работы алгоритмов
4. Реализованы механизмы многопоточности среды Unity применительно к процедуре поиска выхода из лабиринта
5. Получена оценка «отлично» автоматом за экзамен по дисциплине АиСД