**Лабораторные работы.**

**Процедурная генерация уровней и лабиринтов в Unity**

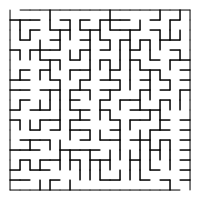
**Примечание.** Это руководство было написано с использованием Unity версии 1.0 от 2017 года.

В этих лабораторных работах вы узнаете, как:

* Процедурно генерировать уровни, и попробуете создать свою игру-лабиринт.
* Генерировать данные относительно лабиринтов.
* Использовать информацию о лабиринте для построения сетки или же меша.

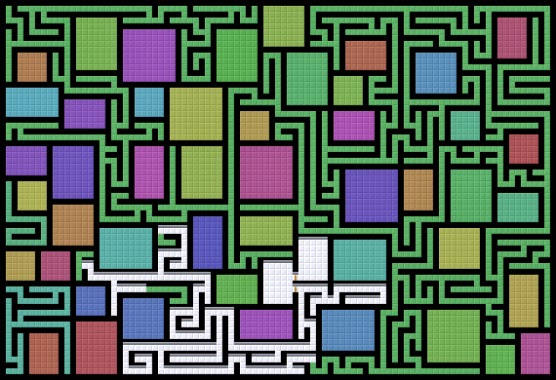
**Лабораторные работа 1. Создание проекта и архитектуры кода.**

Большинство алгоритмов, которые вы можете найти (например, [тут](http://www.astrolog.org/labyrnth/algrithm.htm) и [там](http://www.jamisbuck.org/mazes/)), предусматривают создание «идеальных», плотных лабиринтов; то есть тех, которые имеют только один правильный путь и не имеют петель. Они очень похожи на те, которые вы найдете в разделе головоломки из газеты.



Тем не менее, большинство игр интереснее проходить с лабиринтами, которые являются несовершенными, с петлевыми дорожками, и разреженными, сделанными из открытых пространств вместо узких извилистых коридоров.

Это особенно верно для жанра, похожего на ролевые игры, где процедурные уровни не столько «лабиринты», сколько подземелья. Например **Terraria** или **Don’t Starve**

****

В этой лабораторной работе вы собираетесь реализовать один из самых простых алгоритмов лабиринта, описанных [здесь](http://www.robotron2084guidebook.com/home/games/berzerk/mazegenerator/). Причиной такого выбора является простое добавление лабиринтов в игру с наименьшими усилиями. Этот простой и проверенный подход, который отлично подойдёт для классических игр по вышеуказанной ссылке, поэтому вы будете использовать тот же алгоритм для создания лабиринтов в игре под названием **Speedy Treasure Thief**.

В этой игре каждый уровень представляет собой новый лабиринт, который включает в себя сундук с сокровищами где-то на уровне. Тем не менее, у вас не так много времени, чтобы найти его и уйти, прежде чем охранники вернутся! Каждый уровень имеет ограничение по времени, и вы можете продолжать играть, пока вас не поймают. Ваша оценка основана на том, сколько сокровищ вы собрали за определённый период времени.



Для начала создайте новый пустой проект в Unity.

Теперь загрузите [стартовый пакет](https://koenig-media.raywenderlich.com/uploads/2017/11/proc-mazes-starter.zip), разархивируйте его и импортируйте \*\* **proc-mazes-starter.unitypackage** \*\* в ваш новый проект.

Стартовый пакет включает в себя следующее:

1. Папка **Graphics**, которая содержит все изображения, необходимые для игры.
2. Сама **сцена**, которая является начальной сценой для этой лабораторной работы и содержит плеер и пользовательский интерфейс.
3. Папка с именем **Scripts**, которая содержит два вспомогательных сценария. Остальное вы напишете в этой лабораторной работе.

Этого достаточно, чтобы начать. Вы подробно остановитесь на каждой из этих областей чуть позже.

**Создаем архитектуру нашего кода**

Начните с добавления пустого объекта на сцену. Выберите **GameObject ▸ Create Empty**, дайте ему имя **Controller** и поместите его в **(X: 0, Y: 0, Z: 0)**. Этот объект является просто точкой привязки для скриптов, управляющих игрой.

В каталоге **Scripts** проекта создайте сценарий **C#** с именем **GameController**, затем создайте другой сценарий и назовите его **MazeConstructor**. Первый сценарий будет управлять всей игрой, а второй — специально для создания лабиринта.

Замените все в **GameController** следующим кодом:

using System;

using UnityEngine;

[RequireComponent(typeof(MazeConstructor))] // 1

public class GameController : MonoBehaviour

{

private MazeConstructor generator;

void Start()

{

generator = GetComponent<MazeConstructor>(); // 2

}

}

Вот краткое изложение того, что вы только что создали:

1. Атрибут **RequireComponent** гарантирует, что компонент **MazeConstructor** также будет добавлен при добавлении этого сценария в **GameObject**.
2. Закрытая переменная, в которой хранится ссылка, возвращаемая функцией **GetComponent** ().

Добавьте этот сценарий в сцену: перетащите сценарий **GameController** из окна проекта и поместите его в **GameObject** контроллера в окне иерархии.

Обратите внимание, что **MazeConstructor** также был добавлен в контроллер; это произошло автоматически из-за атрибута **RequireComponent**.

Теперь в **MazeConstructor** замените все на следующий код:

using UnityEngine;

public class MazeConstructor : MonoBehaviour

{

//1

public bool showDebug;

[SerializeField] private Material mazeMat1;

[SerializeField] private Material mazeMat2;

[SerializeField] private Material startMat;

[SerializeField] private Material treasureMat;

//2

public int[,] data

{

get; private set;

}

//3

void Awake()

{

// default to walls surrounding a single empty cell

data = new int[,]

{

{1, 1, 1},

{1, 0, 1},

{1, 1, 1}

};

}

public void GenerateNewMaze(int sizeRows, int sizeCols)

{

// stub to fill in

}

}

Вот что дал нам код, который вы видите выше:

1. Все эти поля доступны вам в Инспекторе. **showDebug** будет использоваться для отображения окна отладки, в то время как различные ссылки на материалы являются материалами для генерированных моделей. Между прочим, атрибут **SerializeField** отображает поле в Инспекторе, даже если переменная является закрытой для доступа к коду.
2. Далее идет свойство данных. Декларации доступа (то есть объявление свойства как открытого, но затем назначение частного набора) делает его доступным только для чтения вне этого класса. Таким образом, данные лабиринта не могут быть изменены извне.

Что касается типа данных, лабиринт сводится к сетке ячеек. Таким образом, данные лабиринта — это просто двумерный массив, который равен **0** или **1** (для представления открытого или заблокированного) для каждого пространства.

Последний интересный код в **Awake** () – инициализирует данные с массивом 3 на 3, который окружает ноль. 1 значит что это «стена», в то время как 0 означает «пусто», поэтому эта сетка по умолчанию является просто замурованной комнатой.

Это достаточный фундамент для кода, но пока мы ничего не видим!

Чтобы отобразить данные лабиринта и проверить, как они выглядят, добавьте следующий метод в **MazeConstructor**:

void OnGUI()

{

//1

if (!showDebug)

{

return;

}

//2

int[,] maze = data;

int rMax = maze.GetUpperBound(0);

int cMax = maze.GetUpperBound(1);

string msg = "";

//3

for (int i = rMax; i >= 0; i--)

{

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

{

if (maze[i, j] == 0)

{

msg += "....";

}

else

{

msg += "==";

}

}

msg += "\n";

}

//4

GUI.Label(new Rect(20, 20, 500, 500), msg);

}

**Разберемся с каждой строкой кода по очереди**

1. Этот код проверяет, включены ли отладочные дисплеи.
2. Вы сможете инициализировать несколько событий локальных переменных: копию готового лабиринта, максимальную строку, столбец а также строку для построения.
3. Два вложенных цикла перебирают строки и столбцы двумерного массива. Для каждой из строки или столбца данного массива, код будет проверяет сохраненное значение и добавляет либо «….», либо «==», это зависит от того, равняться ли значение нулю. Кроме того, код добавляет новую строку после итерации по всем столбцам в строке, так что каждая строка является новой строчкой.
4. Наконец то добрались до **Label** (), который распечатывает встроенную строчку. Этот лейбел использует совершенно новую систему графического интерфейса для дисплеев, видимых игроком, но более старая система используется для создания дисплеев быстрой отладки без особого труда.

Обязательно включите **Show Debug** для значения **MazeConstructor**. Жмите **Play**, и сохраняйте данные лабиринта (который на данный момент является лабиринтом по умолчанию)  
вот что вы увидите:



Отображение сохраненных данных – это хороший результат для начала! Тем не менее, код на самом деле еще не генерирует лабиринт. В следующем лабораторной работе вы поймёте как справиться с этой задачей.

**Лабораторные работа 2. Генерация данных по лабиринту**

Обратите внимание, что **MazeConstructor**.**GenerateNewMaze** () в настоящее время пуст; сейчас это просто поле, которое нужно заполнить позже. В сценарии **GameController** добавьте следующую строку в конец метода **Start** (). Данный код вызовет этот метод для данного параметра:

generator.GenerateNewMaze(13, 15);

Числа 13 и 15 в значениях метода определяют, насколько велик лабиринт. Хотя они еще не используются, эти параметры размера уже определяют количество строк и столбцов в меше соответственно.

С этого момента вы можете начать генерировать данные для лабиринта. Создайте новый скрипт с именем MazeDataGenerator; этот класс будет инкапсулировать логику генерации данных, в этом нам поможет MazeConstructor. Откройте новый скрипт и замените все на:

using System.Collections.Generic;

using UnityEngine;

public class MazeDataGenerator

{

public float placementThreshold; // chance of empty space

public MazeDataGenerator()

{

placementThreshold = .1f; // 1

}

public int[,] FromDimensions(int sizeRows, int sizeCols) // 2

{

int[,] maze = new int[sizeRows, sizeCols];

// stub to fill in

return maze;

}

}

Обратите внимание, что этот класс не наследуется от **MonoBehaviour**. Он не будет напрямую использоваться в качестве компонента, только изнутри **MazeConstructor**, поэтому он не нуждается в функциональности **MonoBehaviour**..

Тем временем:

1. **placementThreshold** будет использоваться алгоритмом генерации данных, чтобы определить, является ли пространство пустым. Этой переменной присваивается значение по умолчанию в конструкторе класса, но она становится общедоступной, чтобы другой код мог настраивать генерированный лабиринт.
2. Еще раз, один метод (в данном случае **FromDimensions** ()) является просто заглушкой для вызова и вскоре будет заполнен.

Затем добавьте несколько разделов кода в **MazeConstructor,** чтобы он мог вызывать метод-заглушку. Сначала добавьте приватную переменную для хранения генератора данных:

private MazeDataGenerator dataGenerator;

Затем создайте его экземпляр в **Awake** (), сохранив генератор в новой переменной, добавив следующую строку в начало метода **Awake** ().

dataGenerator = new MazeDataGenerator();

Наконец, вызовите **FromDimensions** () в **GenerateNewMaze** (), передав размер сетки и сохраняя полученные данные. Найдите строку, содержащую // заглушку для заполнения в **GenerateNewMaze** (), и замените ее следующим:

if (sizeRows % 2 == 0 && sizeCols % 2 == 0)

{

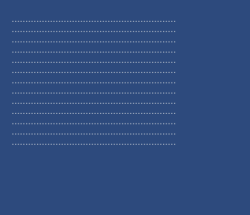
Debug.LogError("Odd numbers work better for dungeon size.");

}

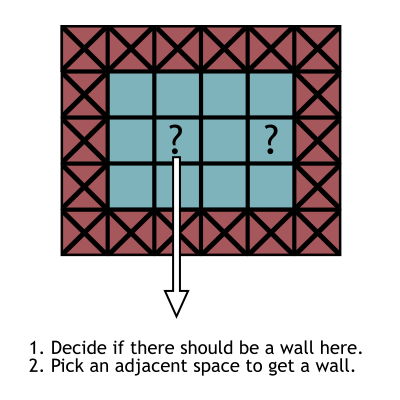
data = dataGenerator.FromDimensions(sizeRows, sizeCols);

Обратите внимание на предупреждение о нечетных числах, это происходит потому, что генерированный лабиринт будет окружен стенами.

Запустите игру, чтобы увидеть правильные размеры, но в остальном вы увидите пустые данные лабиринта:



Отлично! Все готово для хранения и отображения данных лабиринта! Время реализовать алгоритм генерации лабиринта внутри **FromDimensions** ().



Алгоритм, описанный ранее, выполняет итерации по всем другим пространствам в сетке (но не по каждому отдельному пространству!), Чтобы одновременно разместить стену и выбрать соседнее пространство для блокировки. Алгоритм, запрограммированный здесь, является небольшой модификацией, которая также решает, следует ли просто пропускать пространство, что приводит к появлению открытых местностей для изменения лабиринта. Поскольку алгоритму не нужно много хранить или знать что-либо об остальной части лабиринта, например, в списке точек ветвления для перебора, код становится очень простым.

Чтобы реализовать этот алгоритм создания лабиринта, добавьте следующий код в **FromDimensions** () в **MazeDataGenerator**, заменив строку, которая читает // заготовку для заполнения.

int rMax = maze.GetUpperBound(0);

int cMax = maze.GetUpperBound(1);

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

{

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

{

//1

if (i == 0 || j == 0 || i == rMax || j == cMax)

{

maze[i, j] = 1;

}

//2

else if (i % 2 == 0 && j % 2 == 0)

{

if (Random.value > placementThreshold)

{

//3

maze[i, j] = 1;

int a = Random.value < .5 ? 0 : (Random.value < .5 ? -1 : 1);

int b = a != 0 ? 0 : (Random.value < .5 ? -1 : 1);

maze[i+a, j+b] = 1;

}

}

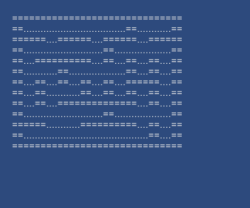
}

}

Приведённый код получает границы двухмерного 2D-массива, после чего проходит через него:

1. Для определенной ячейки меша, код сперва проверит, находится ли текущая ячейка снаружи меша (то есть, существует ли какой-либо индекс внутри границ конкретного массива). Если это так, назначьте 1 для этой стены.
2. Затем код должен проверить, делятся ли координаты равномерно на 2, для дальнейшей работы с любой другой ячейкой. Существует дополнительная проверка по значению **placeThreshold,** описанному ранее, чтобы случайно не пропустить эту ячейку и продолжить итерацию по массиву.
3. Также, код вбивает значение 1 как текущей ячейке, так и рандомной соседней ячейке. Код использует серию троичных операторов для случайного добавления 0, 1 или -1 к индексу массива, таким образом он получит индекс напрямую от соседней ячейки.

Снова просмотрите данные по лабиринту, чтобы проверить, как выглядит случайно генерированный лабиринт:



Перезапустив игру, вы увидите, что каждый раз данные лабиринта обновляются.

Следующая большая задача — генерировать трехмерную сетку из данных 2D лабиринта.

**Лабораторные работа 3. Генерация сетки лабиринта – или же меша**

Теперь, когда базовые данные для лабиринта генерируются правильно, вы можете построить сетку на основе этих данных.

Создайте еще один новый скрипт с именем **MazeMeshGenerator**. Подобно тому, как **MazeDataGenerator** инкапсулировал логику генерации данных, **MazeMeshGenerator** будет содержать логику генерации сетки и для этого будет использоваться **MazeConstructor** чтобы обработать этот шаг в создании лабиринта.

Или, скорее, он в конечном итоге будет содержать логику генерации сетки. Во-первых, вы просто создадите текстурированный квад для демонстрационных целей, а затем измените этот код для создания всего лабиринта. Чтобы сделать это, вам нужно добавить несколько мелких правок в редакторе Unity, прежде чем углубляться в код.

Сначала необходимо связать материалы, которые будут применены к генерированной сетке.

Выберите папку «**Graphics**» в окне «**Project**», затем выберите «**Controller up**» в окне «**Hierarchy**», чтобы увидеть его компонент «**Maze Constructor**» в **Инспекторе**.

Перекидывайте материалы из директории где лежат картинки «**Graphics**» в слоты для материалов в конструкторе лабиринта. Используйте **floor-mat** для Материала 1 и **wall-mat** для Материала 2, пока что сокровища и клады будут в соответствующих слотах.

Поскольку вы уже работаете в Инспекторе, также добавьте тег с именем **Generated**: щелкните меню «**Tag**» в верхней части Инспектора и выберите «**Add** **Tag**». Когда вы генерируете сетки, вы назначаете этот тег для их идентификации.

Теперь, когда все необходимые изменения внесены в редактор Unity, откройте свежий скрипт и пропишите там следующий код:

using System.Collections.Generic;

using UnityEngine;

public class MazeMeshGenerator

{

// generator params

public float width; // how wide are hallways

public float height; // how tall are hallways

public MazeMeshGenerator()

{

width = 3.75f;

height = 3.5f;

}

public Mesh FromData(int[,] data)

{

Mesh maze = new Mesh();

//1

List<Vector3> newVertices = new List<Vector3>();

List<Vector2> newUVs = new List<Vector2>();

List<int> newTriangles = new List<int>();

// corners of quad

Vector3 vert1 = new Vector3(-.5f, -.5f, 0);

Vector3 vert2 = new Vector3(-.5f, .5f, 0);

Vector3 vert3 = new Vector3(.5f, .5f, 0);

Vector3 vert4 = new Vector3(.5f, -.5f, 0);

//2

newVertices.Add(vert1);

newVertices.Add(vert2);

newVertices.Add(vert3);

newVertices.Add(vert4);

//3

newUVs.Add(new Vector2(1, 0));

newUVs.Add(new Vector2(1, 1));

newUVs.Add(new Vector2(0, 1));

newUVs.Add(new Vector2(0, 0));

//4

newTriangles.Add(2);

newTriangles.Add(1);

newTriangles.Add(0);

//5

newTriangles.Add(3);

newTriangles.Add(2);

newTriangles.Add(0);

maze.vertices = newVertices.ToArray();

maze.uv = newUVs.ToArray();

maze.triangles = newTriangles.ToArray();

return maze;

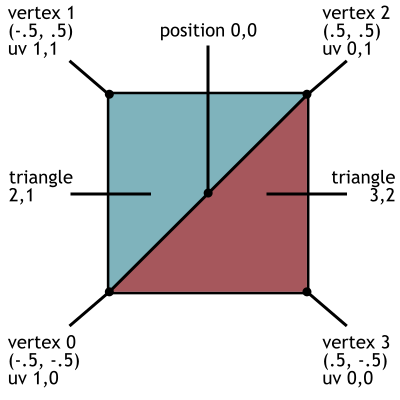
}

}

Два поля в верхней части класса, ширина и высота, аналогичны **locationThreshold** от **MazeDataGenerator**: значения, установленные в конструкторе по умолчанию, которые используются кодом генерации сетки.

Большая часть нужного нам кода лежит внутри **FromData** (); это метод, который нужен **MazeConstructor** для создания сетки. На данный момент код просто создает один квад. Вы увидите как это работает чуть позже. Вы растяните это на весь уровень в скором времени.

На этой иллюстрации показано, из чего сделан квад:



Этот код немного длинный, но довольно повторяющийся, с небольшими изменениями:

1. Для настройки сетки нужно задать параметры: вершины, координаты UV а также треугольники.
2. В каждом списке вершин, находиться положение каждой из них.
3. UV-координаты в списке идут с соответстветствии с самой вершиной в этом списке.
4. Треугольники имеют своё значение индекса в списке вершин (например «этот треугольник составлен из вершин с координатами 0, 1 и 2»).
5. Обратите внимание, что вы создали 2 треугольника; а четырехугольник будет состоять из двух треугольников. Также обратите внимание, что были использованы типы данных **List** (для добавления в список), но в конечном итоге это массивы **Meshneeds**.

**MazeConstructor** должен создать экземпляр **MazeMeshGenerator**, а затем вызвать метод генерации сетки. Между тем, он также должен отображать сетку, поэтому вот кусочки кода, которые нужно добавить:

Сначала создайте приватное поле для хранения генератора сетки.

private MazeMeshGenerator meshGenerator;

Создайте его в **Awake** (), сохранив генератор сетки в новом поле, добавив следующую строку в начало метода **Awake** ():

meshGenerator = new MazeMeshGenerator();

Затем добавьте метод DisplayMaze ():

private void DisplayMaze()

{

GameObject go = new GameObject();

go.transform.position = Vector3.zero;

go.name = "Procedural Maze";

go.tag = "Generated";

MeshFilter mf = go.AddComponent<MeshFilter>();

mf.mesh = meshGenerator.FromData(data);

MeshCollider mc = go.AddComponent<MeshCollider>();

mc.sharedMesh = mf.mesh;

MeshRenderer mr = go.AddComponent<MeshRenderer>();

mr.materials = new Material[2] {mazeMat1, mazeMat2};

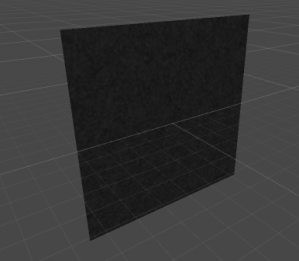
}

Отлично, чтобы вызвать **DisplayMaze** (), добавьте следующую строку в конец **GenerateNewMaze** ():

DisplayMaze();

Сама по себе сетка — это просто данные. Меш не виден, пока не назначен объекту (точнее, объекту **MeshFilter**) в сцене. Таким образом, **DisplayMaze** () не только вызывает **MazeMeshGenerator.FromData** (), но и вставляет этот вызов в середине создания нового **GameObject**, устанавливая тег **Generated**, добавляя **MeshFilter** и генерированный меш, добавляя **MeshCollider** для столкновения с лабиринтом, ну и конечно добавление **MeshRenderer** и материалов для него.

Запрограммировав класс **MazeMeshGenerator** и создав его в **MazeConstructor**, нажмите «**Play**»:



Вы самостоятельно построили текстурированный квад полностью из кода! Это захватывающее и важное начало, поэтому остановитесь здесь, чтобы рассмотреть свою работу до этого момента, если вы не до конца понимаете, как всё работает. Почитайте код несколько раз.

Далее следует довольно значительный рефакторинг **FromData** (); нам нужно полностью его заменить:

public Mesh FromData(int[,] data)

{

Mesh maze = new Mesh();

//3

List<Vector3> newVertices = new List<Vector3>();

List<Vector2> newUVs = new List<Vector2>();

maze.subMeshCount = 2;

List<int> floorTriangles = new List<int>();

List<int> wallTriangles = new List<int>();

int rMax = data.GetUpperBound(0);

int cMax = data.GetUpperBound(1);

float halfH = height \* .5f;

//4

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

{

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

{

if (data[i, j] != 1)

{

// floor

AddQuad(Matrix4x4.TRS(

new Vector3(j \* width, 0, i \* width),

Quaternion.LookRotation(Vector3.up),

new Vector3(width, width, 1)

), ref newVertices, ref newUVs, ref floorTriangles);

// ceiling

AddQuad(Matrix4x4.TRS(

new Vector3(j \* width, height, i \* width),

Quaternion.LookRotation(Vector3.down),

new Vector3(width, width, 1)

), ref newVertices, ref newUVs, ref floorTriangles);

// walls on sides next to blocked grid cells

if (i - 1 < 0 || data[i-1, j] == 1)

{

AddQuad(Matrix4x4.TRS(

new Vector3(j \* width, halfH, (i-.5f) \* width),

Quaternion.LookRotation(Vector3.forward),

new Vector3(width, height, 1)

), ref newVertices, ref newUVs, ref wallTriangles);

}

if (j + 1 > cMax || data[i, j+1] == 1)

{

AddQuad(Matrix4x4.TRS(

new Vector3((j+.5f) \* width, halfH, i \* width),

Quaternion.LookRotation(Vector3.left),

new Vector3(width, height, 1)

), ref newVertices, ref newUVs, ref wallTriangles);

}

if (j - 1 < 0 || data[i, j-1] == 1)

{

AddQuad(Matrix4x4.TRS(

new Vector3((j-.5f) \* width, halfH, i \* width),

Quaternion.LookRotation(Vector3.right),

new Vector3(width, height, 1)

), ref newVertices, ref newUVs, ref wallTriangles);

}

if (i + 1 > rMax || data[i+1, j] == 1)

{

AddQuad(Matrix4x4.TRS(

new Vector3(j \* width, halfH, (i+.5f) \* width),

Quaternion.LookRotation(Vector3.back),

new Vector3(width, height, 1)

), ref newVertices, ref newUVs, ref wallTriangles);

}

}

}

}

maze.vertices = newVertices.ToArray();

maze.uv = newUVs.ToArray();

maze.SetTriangles(floorTriangles.ToArray(), 0);

maze.SetTriangles(wallTriangles.ToArray(), 1);

//5

maze.RecalculateNormals();

return maze;

}

//1, 2

private void AddQuad(Matrix4x4 matrix, ref List<Vector3> newVertices,

ref List<Vector2> newUVs, ref List<int> newTriangles)

{

int index = newVertices.Count;

// corners before transforming

Vector3 vert1 = new Vector3(-.5f, -.5f, 0);

Vector3 vert2 = new Vector3(-.5f, .5f, 0);

Vector3 vert3 = new Vector3(.5f, .5f, 0);

Vector3 vert4 = new Vector3(.5f, -.5f, 0);

newVertices.Add(matrix.MultiplyPoint3x4(vert1));

newVertices.Add(matrix.MultiplyPoint3x4(vert2));

newVertices.Add(matrix.MultiplyPoint3x4(vert3));

newVertices.Add(matrix.MultiplyPoint3x4(vert4));

newUVs.Add(new Vector2(1, 0));

newUVs.Add(new Vector2(1, 1));

newUVs.Add(new Vector2(0, 1));

newUVs.Add(new Vector2(0, 0));

newTriangles.Add(index+2);

newTriangles.Add(index+1);

newTriangles.Add(index);

newTriangles.Add(index+3);

newTriangles.Add(index+2);

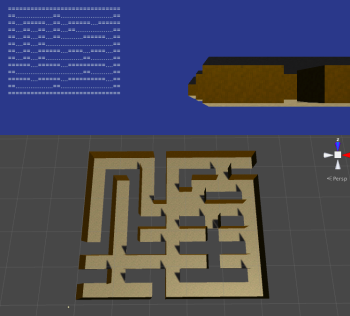
newTriangles.Add(index);

}

Последняя часть кода была довольно длинной! Но опять же, код повторяется, только некоторые числа в нём изменены. В частности, код для создания квада был перемещен в отдельный метод **AddQuad** (), он служит для повторного вызова квада для пола, потолка и стен из каждой ячейки меша.

1. Остальные 3 параметра **AddQuad** () — это тот самый список вершин, UV и треугольников, к которому нужно добавить значения. Для справки, первая строка метода получает индекс для начала; По мере добавления четырех квадратов, их индекс будет расти.
2. Важно понимать, что первый параметр **AddQuad** () является матрицей преобразования, и эта часть может сбить вас с толку. По существу, параметры положение / вращение / масштаб могут быть сохранены в матрице, а затем применены к вершинам. Это то, что делают вызовы **MultiplyPoint3x4** (). Таким образом, можно использовать один и тот же код для создания четырехугольника, полов, стен и т. д. Вам нужно только изменить используемую матрицу преобразования.
3. Возвращаясь к **FromData** (), он отвечает за списки вершин, UV и треугольников, которые создаются вверху. На этот раз есть два списка треугольников. Объект Unity **Mesh** может хранить в себе несколько подсетей с разным материалом на каждом меше, в итоге каждый список треугольников будет определяться как отдельная подсеть. Вы объявляете две подсетки, чтобы можно было назначать различные материалы, один для пола а другой для стен.
4. После этого вы перебираете 2D-массив и строите квадраты для пола, стенок лабиринта и потолка в каждой ячейке . В то время как каждая ячейка нуждается в полу и потолке, существуют проверки соседних ячеек, чтобы увидеть, какие стены необходимы. Обратите внимание, как **AddQuad** () вызывается неоднократно, но всегда будет с другой матрицей преобразования и с совершенно другими списками треугольников, которые используются для стенок и полов. Также обратите внимание, что ширина и высота используются для определения расположения квадратов и их размера.
5. Последний параметр: **RecalculateNormals** () подготавливает сетку для освещения.

Нажмите **Play**, чтобы увидеть полную созданную сетку лабиринта:



Наши поздравления; это было большинство лабиринтов и программ, необходимых для игры **Speedy Treasure Thief**. Предлагаем вам завершить остальную часть игры в следующем лабораторной работе.

**Лабораторные работа 4. Завершающая настройка созданной игры**

Вам нужно сделать еще несколько дополнений и изменений в коде, но сначала давайте рассмотрим, что было предоставлено стартовым пакетом. Как упоминалось во введении, стартовый пакет включал в себя два сценария, сцену с плеером и пользовательским интерфейсом а также всю графику для игры-лабиринта. Скрипт **FpsMovement** — это просто версия скрипта контроллера персонажей из [книги](https://www.amazon.com/gp/product/1617294969), а **TriggerEventRouter** — это очень удобная утилита, для запуска внутри игры.

В сцене уже настроен проигрыватель, включая компонент **FpsMovement** и прожектор, прикрепленный к камере. Скайбокс и освещение окружающей среды также отключены в окне «**Lightning** **setting**». Наконец, сцена имеет холст пользовательского интерфейса с уже размещенными метками для счета и времени.

Это итог того, что предоставляет стартовый пакет. Теперь вы можете самостоятельно написать оставшийся код для этой игры.

Начнём с **MazeConstructor**. Сначала добавьте следующие свойства для хранения размеров и координат:

public float hallWidth

{

get; private set;

}

public float hallHeight

{

get; private set;

}

public int startRow

{

get; private set;

}

public int startCol

{

get; private set;

}

public int goalRow

{

get; private set;

}

public int goalCol

{

get; private set;

}

Дальше будем добавлять несколько свежих методов. Первый из них — это **DisposeOldMaze** (); как следует из названия,он удаляет любой существующий, или старый лабиринт. Он попросту найдёт все объекты с тегом **Generated** и уничтожит их.

public void DisposeOldMaze()

{

GameObject[] objects = GameObject.FindGameObjectsWithTag("Generated");

foreach (GameObject go in objects) {

Destroy(go);

}

}

Следующий метод который мы добавим — **FindStartPosition** (). Этот код начинается с 0,0 и перебирает данные лабиринта, пока не найдет валидное пространство. После чего эти координаты будут сохранены в качестве начальной позиции лабиринта.

private void FindStartPosition()

{

int[,] maze = data;

int rMax = maze.GetUpperBound(0);

int cMax = maze.GetUpperBound(1);

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

{

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

{

if (maze[i, j] == 0)

{

startRow = i;

startCol = j;

return;

}

}

}

}

Точно так же **FindGoalPosition** () делает то же самое, только начиная с максимальных значений и заканчивая обратным отсчетом. Добавьте и этот метод:

private void FindGoalPosition()

{

int[,] maze = data;

int rMax = maze.GetUpperBound(0);

int cMax = maze.GetUpperBound(1);

// loop top to bottom, right to left

for (int i = rMax; i >= 0; i--)

{

for (int j = cMax; j >= 0; j--)

{

if (maze[i, j] == 0)

{

goalRow = i;

goalCol = j;

return;

}

}

}

}

Параметры **PlaceStartTrigger**() и **PlaceGoalTrigger**() служат для размещения объектов на сцене в начальной и конечной позициях. Их коллайдер установлен как триггер, сперва будет применен соответствующий материал, а затем добавится событие **TriggerEventRouter** (из стартового пакета). Этот компонент принимает функцию обратного вызова, когда что-то входит в значение триггера. Добавьте также эти два метода:

private void PlaceStartTrigger(TriggerEventHandler callback)

{

GameObject go = GameObject.CreatePrimitive(PrimitiveType.Cube);

go.transform.position = new Vector3(startCol \* hallWidth, .5f, startRow \* hallWidth);

go.name = "Start Trigger";

go.tag = "Generated";

go.GetComponent<BoxCollider>().isTrigger = true;

go.GetComponent<MeshRenderer>().sharedMaterial = startMat;

TriggerEventRouter tc = go.AddComponent<TriggerEventRouter>();

tc.callback = callback;

}

private void PlaceGoalTrigger(TriggerEventHandler callback)

{

GameObject go = GameObject.CreatePrimitive(PrimitiveType.Cube);

go.transform.position = new Vector3(goalCol \* hallWidth, .5f, goalRow \* hallWidth);

go.name = "Treasure";

go.tag = "Generated";

go.GetComponent<BoxCollider>().isTrigger = true;

go.GetComponent<MeshRenderer>().sharedMaterial = treasureMat;

TriggerEventRouter tc = go.AddComponent<TriggerEventRouter>();

tc.callback = callback;

}

Осталось всего лишь заменить весь метод **GenerateNewMaze** () следующим кодом:

public void GenerateNewMaze(int sizeRows, int sizeCols,

TriggerEventHandler startCallback=null, TriggerEventHandler goalCallback=null)

{

if (sizeRows % 2 == 0 && sizeCols % 2 == 0)

{

Debug.LogError("Odd numbers work better for dungeon size.");

}

DisposeOldMaze();

data = dataGenerator.FromDimensions(sizeRows, sizeCols);

FindStartPosition();

FindGoalPosition();

// store values used to generate this mesh

hallWidth = meshGenerator.width;

hallHeight = meshGenerator.height;

DisplayMaze();

PlaceStartTrigger(startCallback);

PlaceGoalTrigger(goalCallback);

}

Переписанный **GenerateNewMaze** () вызовет новые методы, которые вы только что прописали, для таких вещей, как удаление старой сетки и размещение триггеров.

Вы добавили много кода в **MazeConstructor**. Великолепно. К счастью, вы уже закончили с этим классом на некоторое время. Осталось добавить ещё один небольшой код.

Теперь добавьте дополнительный код в **GameController**. Замените все содержимое этого файла следующими строками:

using System;

using UnityEngine;

using UnityEngine.UI;

[RequireComponent(typeof(MazeConstructor))]

public class GameController : MonoBehaviour

{

//1

[SerializeField] private FpsMovement player;

[SerializeField] private Text timeLabel;

[SerializeField] private Text scoreLabel;

private MazeConstructor generator;

//2

private DateTime startTime;

private int timeLimit;

private int reduceLimitBy;

private int score;

private bool goalReached;

//3

void Start() {

generator = GetComponent<MazeConstructor>();

StartNewGame();

}

//4

private void StartNewGame()

{

timeLimit = 80;

reduceLimitBy = 5;

startTime = DateTime.Now;

score = 0;

scoreLabel.text = score.ToString();

StartNewMaze();

}

//5

private void StartNewMaze()

{

generator.GenerateNewMaze(13, 15, OnStartTrigger, OnGoalTrigger);

float x = generator.startCol \* generator.hallWidth;

float y = 1;

float z = generator.startRow \* generator.hallWidth;

player.transform.position = new Vector3(x, y, z);

goalReached = false;

player.enabled = true;

// restart timer

timeLimit -= reduceLimitBy;

startTime = DateTime.Now;

}

//6

void Update()

{

if (!player.enabled)

{

return;

}

int timeUsed = (int)(DateTime.Now - startTime).TotalSeconds;

int timeLeft = timeLimit - timeUsed;

if (timeLeft > 0)

{

timeLabel.text = timeLeft.ToString();

}

else

{

timeLabel.text = "TIME UP";

player.enabled = false;

Invoke("StartNewGame", 4);

}

}

//7

private void OnGoalTrigger(GameObject trigger, GameObject other)

{

Debug.Log("Goal!");

goalReached = true;

score += 1;

scoreLabel.text = score.ToString();

Destroy(trigger);

}

private void OnStartTrigger(GameObject trigger, GameObject other)

{

if (goalReached)

{

Debug.Log("Finish!");

player.enabled = false;

Invoke("StartNewMaze", 4);

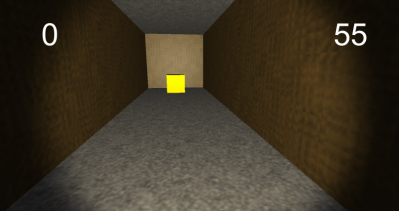
}

}

}

1. Сначала вы внедрили сериализованные поля для каждого объекта в сцене.
2. Несколько личных переменных были добавлены, чтобы отследить таймер игры, счет, и узнать была ли найдена цель лабиринта.
3. **MazeConstructor**активируется так же, как и раньше, но сейчас параметр **Start**() использует новые методы, которые делают больше, чем просто вызов **GenerateNewMaze**().
4. **StartNewGame()**используется для запуска всей игры с самого начала, в отличие от переключения между уровнями в игре. Таймер устанавливается на начальные значения, счет сбрасывается, а затем создается лабиринт.
5. **StartNewMaze()**отвечает за переход на следующий уровень без начала всей игры. Кроме создания нового лабиринта, этот метод отправит игрока в начало уровня, сбросит цель и сократит время.
6. **Update()**смотрит за активностью игрока, после чего обновляет оставшееся минуты до завершения уровня. Когда время истекло, игрока выкинет и запустится новая игра.
7. **OnGoalTrigger()**и **OnStartTrigger()** это обратные вызовы, передаваемые в **TriggerEventRouterin MazeConstructor.** Событие **OnGoalTrigger**() этот триггер проверяет, что цель была найдена, а затем увеличивает счет. Другое событие **OnStartTrigger** () проверяет, была ли цель достигнута, а затем выбрасывает игрока и запускает новый лабиринт.

Вот и весь код. Обратите свое внимание на сцену в Unity. Сначала выберите **Canvas** в окне **Hierarchy** и включите его в **Инспекторе**. **Canvas** был отключен, чтобы не мешать отображению отладки при создании кода лабиринта. Помните, что сериализованные поля были добавлены, поэтому перетащите эти объекты сцены (**Player**, метку времени на холсте и метку счета) в слоты в **Инспекторе**. Отключайте окно исправления ошибок **Show Debug** и запускайте игру нажав на Play:



Процедурно генерировать лабиринты, в первое время может быть очень сложно, но они приводят к увлекательному и динамичному игровому процессу.

**Что дальше?**

Если вы выполняли все шаги по-порядку, у вас должна была получиться готовая игра.

Вы можете скачать [готовый проект](https://koenig-media.raywenderlich.com/uploads/2017/11/proc-maze-tutorial.zip).

Забегая вперед, вы можете попробовать другие алгоритмы генерации, заменив код в FromDimensions().

Вы можете попробовать сгенерировать другое окружение.

Случайно сгенерированные на карте предметы и враги могут разнообразить игру.