УДК 004

Процедурная генерация игровых локаций на Unity

Фатеенков Данила Витальевич
Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация
В статье описывается реализация на языке программирования C# алгоритма генерации игровой локации. Рассматриваются основные функции Unity API для генерации объектов на игровом пространстве, а также проверки коллизии для поиска ошибок в генерации.
Ключевые слова: Unity, C#, процедурная генерация

Procedural generation of game locations on Unity

Fateenkov Danila Vitalievich
Sholom-Aleichem Priamursky State University
Student

Abstract
The article describes implementation of the game location generation algorithm in C# programming language. The main functions of Unity API for generating objects on the game space, as well as collision checks for finding errors in the generation are considered.
Keywords: Unity, C#, procedural generation

1. Введение

1.1 Актуальность

Процедурная генерация – метод создания определённых объектов или данных алгоритмически. В основном данные создаются на основе раннее указанных значений, а также генерируемой компьютером случайности и вычислительной мощности.

Процедурные генерации используются во многих сферах, где имеет место создание объектов и данных, которые вручную разработать будет тяжело: в компьютерных играх на основе данного метода строятся игровые локации, в графике – текстуры и 3D-модели, в киноиндустрии – спецэффекты.

Данный метод разработки актуален и его применение при решении задачи генерации большого количества данных часто представлены уникальными алгоритмами, которые созданы для решения одной или нескольких задач.

1.2 Обзор исследований

И.А Демидов в своей работе рассмотрел такие алгоритмы процедурной генерации игровой локации как: клеточный автомат, двоичное разбиение пространства, алгоритм случайного блуждания и применение логистической решётки [1]. Ч.Т Куулар, Т.В Монгуш и В.А Коровкин разработали технологию процедурной генерации игровых локаций с применением графовых алгоритмов, языка программирования C++ и Unreal Engine 4 [2]. М.Г Меженин провёл обзор исследований в области процедурной генерации игровых правил и игр [3]. С.В Николаев и А.Г Демянчук разработали алгоритм процедурной генерации игровой локации для сетевого шутера, созданного на Unreal Engine 4 [4].

1.3 Цель исследования

Цель – реализовать алгоритм случайной генерации локаций в среде разработки компьютерных игр Unity, используя язык программирования C#. Алгоритм должен генерировать цепочку помещений в четырёх направлениях в зависимости от расположения выхода из ранее созданной локации.

2. Материалы и методы

Для реализации алгоритма будет использован язык программирования C# и среда разработки Unity.

3. Результаты и обсуждения

Игровая локация будет представлять набор комнат, которые необходимо создать заранее. Набор комнат представлен в виде массива объектов (рисунок 1). Комната может состоять из любых объектов, но один объект должен присутствовать обязательно – триггер в переходе в другую комнату. Данный объект является точкой генерации комнаты и должен определять, какая комната генерируется следующей.

Рисунок 1. Набор комнат, используемый для генерации

Задаются следующие переменные в скрипте, который отвечает за генерацию новой комнаты:

```
    public GameObject[] rooms; //массив комнат
    private GameObject roomGenerate; //выбранная комната для
генерации

    private Vector3 triggerPosition; //позиция триггера
    private Vector3 roomCoordinates; //координаты новой комнаты
```

На игровом поле задаётся стартовая комната – координаты в пространстве могут быть любые, но чаще всего стартовой точкой определяются координаты (0,0,0).

Также создан класс “RoomCounter” и глобальная переменная “roomCounter”, которая является целочисленной переменной – количество комнат на игровой локации.

В Unity API задана функция “Instantiate”, создающая клон уже существующего объекта. Перед созданием комнаты необходимо определить её расположение, а также поворот. Изначально выбирается одна из комнат из массива случайным образом. Длина массива изменяется при добавлении нового элемента в неё. Также определяется позиция триггера, который является точкой создания новой комнаты. Скрипт генерации следующей комнаты устанавливается на триггер:

```
void PickNextRoom(){
     roomGenerate = rooms[Random.Range(0, rooms.Length)];
     triggerPosition = this.transform.position;
     DetectDirection();
}
```

После выбора комнаты и определения позиция триггера в функции “PickNextDoor” запускается функция “DetectDirection”, которая определяет поворот комнаты. Скрипт работает только с двумя координатными осями: X и Z, и для оси Y требуется незначительное изменение в коде функции (так как данная ось только определяет высоты триггера, то достаточно заменить 0 на “triggerPosition.y”).

Поворот триггера в пространстве определяется с использованием метода “eulerAngles”, который возвращает трёхмерный вектор поворота объекта в пространстве. Так как не предполагается, что комнаты могут быть расположены под углом, то достаточно проверять значение оси Y. Для определения поворота берётся значение оси Y поворот триггера, на который установлен скрипт.

Также функция “DetectDirection” определяет координаты объекта в пространстве. Так как триггер, на который установлен скрипт, является центром при генерации новой комнаты, то необходимо задать смещение от центра, чтобы не было коллизий. Данное смещение составляет половину выбранной комнаты для генерации (рис. 2).

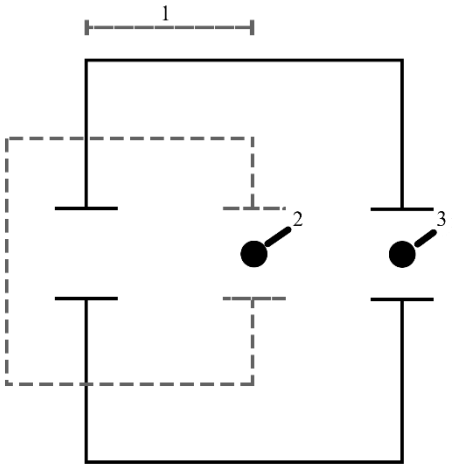

Рисунок 2. Определение координат для новой комнаты

1 – расстояние, на которое сдвигается новая комната.

2 – триггер генерации комнаты, который привязан к уже созданному префабу. Является центром для новой комнаты при исходных координатах.

3 – триггер генерации комнаты, привязанный к новой комнате. Не начинает свою работу, пока комната не расположена в правильных координатах.

Пунктирной линией обозначена комната, которая уже создана и триггер перехода которой начал свою работу, а непрерывной линией – комната, которая генерируется на данном шаге.

Для определения размера комнаты, необходимо получить значения размера коллайдера пола. Для получения данных значения нужно обратиться к компоненту “BoxCollider” (или другой коллайдер, который используется). Размер получить можно как вектор через функцию “size”.

```
void DetectDirection(){
     switch ((int)this.gameObject.transform.eulerAngles.y){
          case 0:
               roomCoordinates.x = triggerPosition.x -
          (roomGenerate.GetComponent<BoxCollider>().size.x *
          roomGenerate.transform.localScale.x) / 2;
               roomCoordinates.y = 0;
               roomCoordinates.z = triggerPosition.z;
               break;
          case 90:
               roomCoordinates.x = triggerPosition.x;
               roomCoordinates.y = 0;
               roomCoordinates.z = triggerPosition.z +
          (roomGenerate.GetComponent<BoxCollider>().size.x *
          roomGenerate.transform.localScale.x) / 2;
               break;
          case 180:
               roomCoordinates.x = triggerPosition.x +
          (roomGenerate.GetComponent<BoxCollider>().size.x *
          roomGenerate.transform.localScale.x) / 2;
```

```
                roomCoordinates.y = 0;
                roomCoordinates.z = triggerPosition.z;
                break;
            case 270:
                roomCoordinates.x = triggerPosition.x;
                roomCoordinates.y = 0;
                roomCoordinates.z = triggerPosition.z -
            (roomGenerate.GetComponent<BoxCollider>().size.x *
            roomGenerate.transform.localScale.x) / 2;
                break;
            default:
                Debug.Log((int)this.gameObject.transform.eule
            rAngles.y);
                break;
        }
    }
```

После определения координат и поворота комнаты, она создаётся с применением функции “Instantiate”, основными параметрами которой являются:

1. Объект, который необходимо создать (представлен как GameObject).
2. Расположение объекта в игровом пространстве (представлен как Vector3).
3. Поворот объекта (представлен как Quaternion).
4. Родительский объект, к которому будет привязан созданный (представлен как Transform).

Созданной комнате задаётся поворот через функцию “transform.Rotate()”, который был определён в “DetectDirection()”. Также уменьшается счётчик “roomCounter” на 1 и скрипт прекращает свою работу:

```
if (RoomCounter.roomCounter > 0){
     PickNextRoom();
     var roomNew = Instantiate(roomGenerate, new
     Vector3(roomCoordinates.x, roomCoordinates.y,
     roomCoordinates.z), Quaternion.identity);
     roomNew.gameObject.transform.Rotate(0,
     this.gameObject.transform.eulerAngles.y, 0,
     Space.Self);
     RoomCounter.roomCounter--;
     this.GetComponent<NewRoomGeneration>().enabled = false;
}
if (RoomCounter.roomCounter == 0)
{
     this.GetComponent<NewRoomGeneration>().enabled = false;
}
```

Получившийся скрипт будет генерировать локации случайным образом с заданным количеством комнат (переменная roomCounter), но отсутствует проверка коллизий, что может вызвать сбои в работе алгоритма.

В первую очередь создаётся новый тег для комнат, так как при проверке коллизий, скрипт будет получать тег объектов, а не их название или другой параметр (рис. 3).

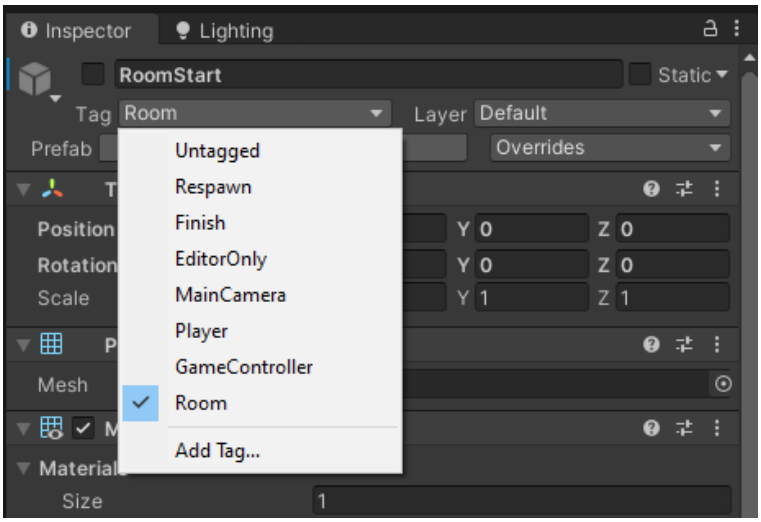

Рисунок 3. Новый тег, созданный для комнат

Существует 2 способа проверки коллизии у объектов:

1. Добавление функции “OnCollisionEnter”. Данная функция будет выполняться только если произошло столкновение с каким-либо объектом.

2. Применение класса “Physics”, в котором есть функции, собирающие все объекты с коллизиями в определённой области.

Первый способ легче в реализации и не требует работы с тегами. Достаточно проверять имена объектов, которые столкнулись и если они не равны, а также наличие слова “Room” у объекта, с которым произошло взаимодействие (так как данное слово присутствует во всех префабах, но отсутствует в элементах комнат), то удалять созданный объект.

Удалить объект можно, используя функцию “Destroy()”, единственным параметром которого является игровой объект. Также нужно сравнивать имена не элементов комнат с самой комнатой, в которой происходит сравнение, а обращаться к родительскому объекту элементов префаба и получать его название:

```
collision.transform.parent.gameObject.name
```

Определить наличие подстроки в исходной строке можно, применив функцию “Contains”, которая является функцией стандартной библиотеки C#. Перед удалением комнаты также необходимо увеличить roomCounter на 1.

Второй способ подразумевает использование созданного тега. Создаётся коллайдер, который находит все коллизии внутри себя. Недостаток данного способа в контексте решаемой задачи состоит в том, что формы

коллайдера ограничены и создать можно только куб, капсулу или сферу, то есть форма комнаты не всегда совпадает с коллайдером.

Для работы скрипта задаётся массив объектов, с которыми произошло взаимодействие внутри созданного коллайдера:

```
private Collider[] hitColliders;
```

После создания необходимого объекта класса “Physics” подсчитывается количество вошедших в массив объектов с тегом “Rooms”. Для выполнения данной задачи можно использовать библиотеку LINQ.

LINQ – язык запросов, позволяющий выполнять операции и функции по выборке данных из объектов, массивов, XML-документов, баз данных и дата сетов. В C# он подключается, как и другие библиотеки: using System.Linq. Запрос по выборке данных с определённым параметров задаётся следующим образом:

```
var selectedItems = arr_items.Where(t => t.StartsWith("P"));
```

После подсчёта объектов с тегом “Rooms” необходимо удалить созданную комнату, если счётчик больше 1 (исходный объект также будет добавлен в массив), то объект удаляется:

```
hitColliders = Physics.OverlapBox(this.transform.position,
this.transform.localScale * 4.75f, Quaternion.identity);
int roomCollidedCount = hitColliders.Count(t => t.tag ==
"Room");
if (roomCollidedCount > 1)
{
     RoomCounter.roomCounter++;
     Destroy(this.gameObject);
}
```

Реализация проверки коллизий у объекта позволит избежать ошибок в генерации игровой локации и сделает её проходимой. Данный код можно реализовать в отдельном скрипте, который будет привязан к префабу – данный вариант легче при работе с координатами префаба, так как нет необходимости определять их через функции класса “transform.parent”. Но у второго способа проверки коллизий есть недостаток – если объектов будет слишком много, то производительность может снизиться, поэтому нужно учитывать количество объектов в префабе при создании родительских объектов для процедурной генерации.

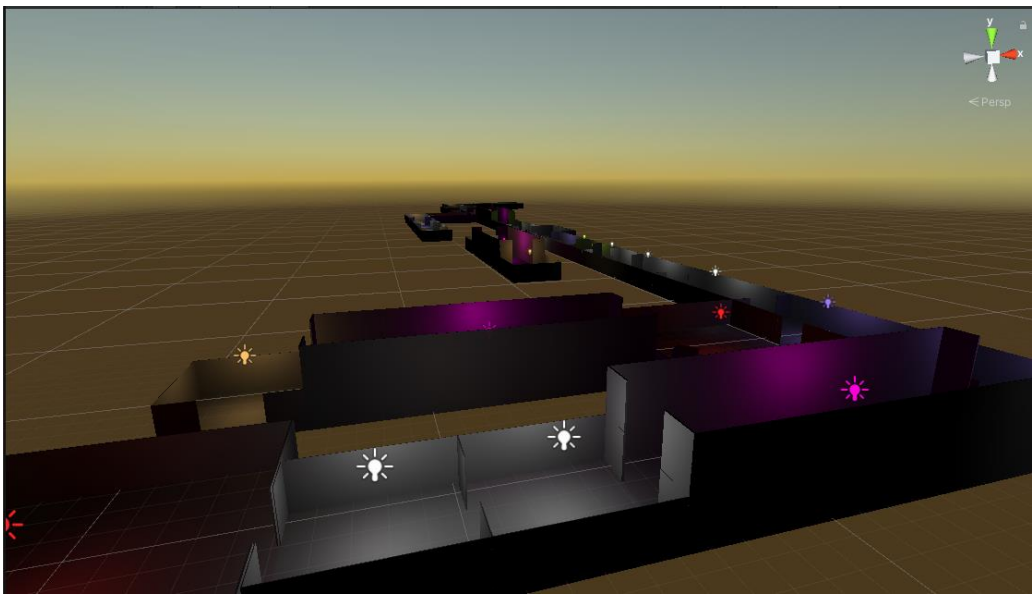

Рисунок 4. Пример работы алгоритма генерации

У алгоритма есть недостаток – генерация может завершиться, но количество комнат не достигнет нуля. Это может произойти в случае, если все пути при генерации замкнутся или окажутся в тупике. Данную проблему можно исправить добавлением ещё одного этажа, на который будет переходить алгоритм и продолжать генерацию локации. Узнать поворот комнаты-перехода на верхний этаж не сложно – достаточно взять угол поворота с комнаты, на место которой ставится переход. Главная проблема – определение расположения объекта в пространстве.

Так как переход на верхний этаж создаётся из родительского объекта, а не триггера, то нужно сдвигать данный переход на половину от длины комнаты либо по оси X, либо по оси Z (зависит от поворота комнаты) и сделать ещё один сдвиг вперёд на половину длины комнаты-перехода. Но удобнее сдвигать проблемную комнату на исходную позицию и создавать переход с последующим удалением ненужной комнаты.

Для определения координат задана новая функция “MoveRoom()”, которая возвращает проблемную комнату на исходную позицию и делает сдвиг на половину длины комнаты-перехода:

```
void MoveRoom() {
     switch ((int)this.gameObject.transform.eulerAngles.y) {
     case 0:
          this.transform.position = new
          Vector3(this.transform.position.x +
          ((this.GetComponent<BoxCollider>().size.x *
          this.transform.localScale.x) / 2) -
          ((stairObject.GetComponent<BoxCollider>().size.x *
          stairObject.transform.localScale.x) / 2),
          this.transform.position.y, this.transform.position.z);
          break;
     case 90:
```

```
            this.transform.position = new
            Vector3(this.transform.position.x,
            this.transform.position.y, this.transform.position.z -
            ((this.GetComponent<BoxCollider>().size.x *
            this.transform.localScale.x) / 2) +
            ((stairObject.GetComponent<BoxCollider>().size.x *
            stairObject.transform.localScale.x) / 2));
            break;
      case 180:
            this.transform.position = new
            Vector3(this.transform.position.x -
            ((this.GetComponent<BoxCollider>().size.x *
            this.transform.localScale.x) / 2) +
            ((stairObject.GetComponent<BoxCollider>().size.x *
            stairObject.transform.localScale.x) / 2),
            this.transform.position.y, this.transform.position.z);
            break;
      case 270:
            this.transform.position = new
            Vector3(this.transform.position.x,
            this.transform.position.y, this.transform.position.z +
            ((this.GetComponent<BoxCollider>().size.x *
            this.transform.localScale.x) / 2) -
            ((stairObject.GetComponent<BoxCollider>().size.x *
            stairObject.transform.localScale.x) / 2));
            break;
default:
            break;
      }
}
```

Но также можно запоминать расположение триггера, который был использован для создания проблемной комнаты, чтобы не вычислять расстояние, на которое необходимо сдвинуть комнату назад.

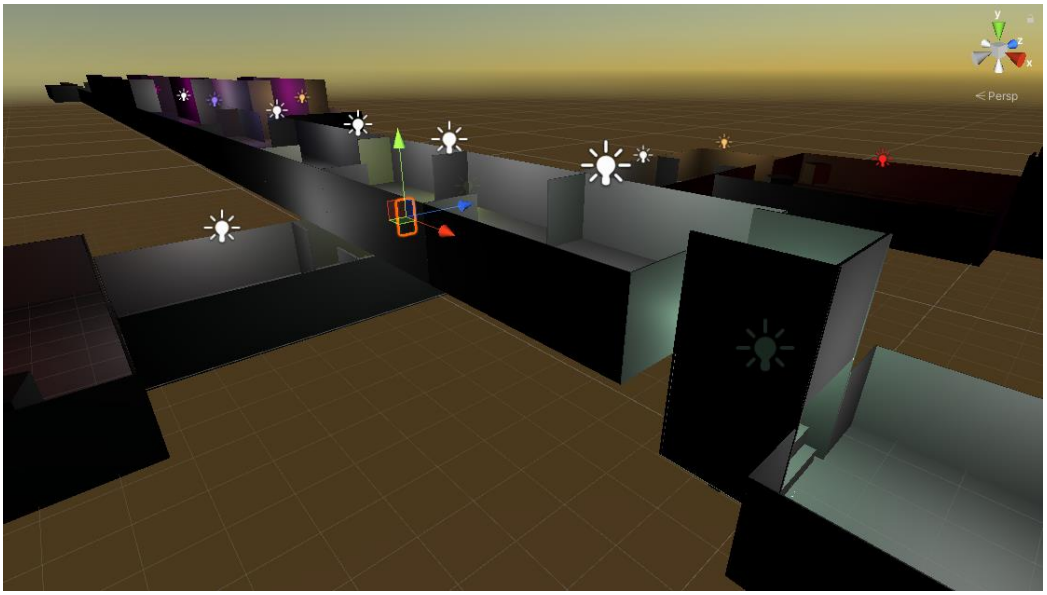

Рисунок 5. Пример работы алгоритма генерации с переходами по оси Y

Таким образом, на выходе получился алгоритм процедурной генерации локаций для игры, созданной в Unity. Алгоритм протестирован на генерациях больших локаций (500 и больше комнат) и способен показать хороший результат в скорости работы.

Библиографический список

1. Демидов И.А Генерация игровых пространств с помощью логистической решетки. СПб.: ООО "Издательство ВВМ", 2019. 217 с.
2. Куулар Ч.Т, Монгуш Т.В, Коровкин В.А Разработка технологии для процедурной генерации виртуальных пространств на Unreal Engine 4. Томск: Национальный исследовательский Томский политехнический университет, 2021. 320-322 с.
3. Меженин М.Г Обзор систем процедурной генерации игр // Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. 2015. Т. 4. № 1. С. 5-20
4. Николаев С.В, Демянчук А.Г. Алгоритм процедурной генерации города для сетевого шутера. Санкт-Петербург: ЕНМЦ «Мультидисциплинарные исследования», 2021. С. 12-19.
5. Unity - Manual: Unity User Manual 2020.3 (LTS) URL: https://docs.unity3d.com/Manual/index.html. – Дата доступа: 29.01.2022.