3 Анализ задачи

3.1 Выбор и обоснование алгоритмов

3.1.1 Алгоритмы обработки исходных данных

Учитывая существующую на данный период огромную библиотеку музыкальных произведений и большую вариативность параметров, которые можно достать из одного произведения, было принято решение использовать музыку в качестве входных данных для генератора. Лишь несколько существующих программ используют музыкальные произведения в качестве входных данных для своих генераторов, из-за чего новые работы будут востребованными.

Музыкальные данные представляют собой волну звукового сигнала. После оцифровки сигнала, музыкальные данные являются набором значений, отражающим аналоговую цифровую волну, прошедшую дискретизацию и квантование.

*Дискретизация* – это разбиение непрерывного сигнала на набор дискретных значений, обычно взятых через разные промежутки времени. В таком случае, дискретизация имеет частоту, измеряемую в герцах, и означает количество измерений волны в секунду. Согласно теореме Котельникова (в англоязычной литературе — теореме Найквиста-Шеннона), для того, чтобы точно передать сигнал, состоящий из частот с максимумом в *N*, нужно использовать дискретизацию с частотой в *2N*. Потому, чтобы передать песни во всём слышимом диапазоне (от 20 Гц до 20 кГЦ), самый распространённый музыкальный формат имеют частоту дискретизации в 44,1 кГЦ.

*Квантование* – это разбиение значений сигнала на набор уровней и округление значения любого значения к ближайшему к нему уровню. Так как в цифровой обработке используются двоичные числа, то уровни квантования являются степенью двойки, а сама степень называется *глубиной квантования* [8].

Разница между дискретизацией, квантованием и оцифровкой приведена на рисунке 3.1.

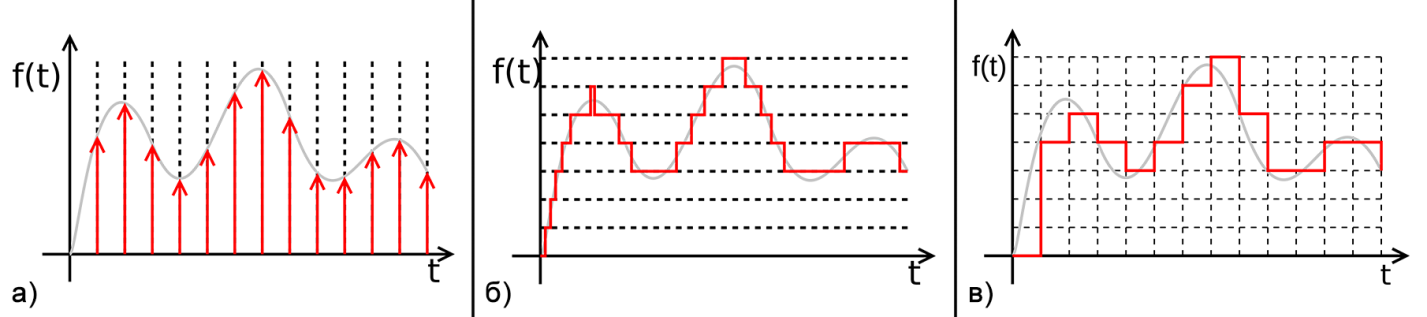


Рисунок 3.1 – Разница между обработкой звуковой волны: а) дискретизация;

б) квантование; в) оцифровка (дискретизация и квантование)

Полученная информация не обладает достаточной глубиной для определения темпа песни и получения вариативных параметров. Для получения нужных данных, следует преобразовать сигнал из амплитудно-временной плоскости в амплитудно-частотную. Для этого следует использовать оконное преобразование Фурье, разновидность преобразования Фурье, для разложения сигнала на частоты на конечном интервале. На входе вместо сигнала в данном преобразовании будет произведение сигнала на оконную функцию. Из-за этого возникает эффект под названием *«растекание спектра сигнала»*. Чтобы его ослабить, используют разные функции окна [9]. Разные функции окон и утечки сигнала после их применения показаны на рисунке 3.2.

Оптимальным для снятия замеров звуковой волны песни будет окно в 1/40 секунды, что для частоты дискретизации песни в 44,1 кГц будет составлять 1024 замера, а для песен с частотой 96 кГЦ – 2048 замеров для окна. Такое количество замеров обусловлено тем, что для ускорения процесса подсчёта будет использоваться не дискретное, а быстрое преобразование Фурье, имеющий сложность , которому на вход нужно подавать количество замеров, равное степени двойки. Выходом будет спектр частот, состоящий из замеров положительных и отрицательных значений одной частоты. Это означает, для дальнейшей обработки понадобится только половина полученных данных. Однако даже 512 массивов значений – это слишком много. Для работы достаточно будет 128.

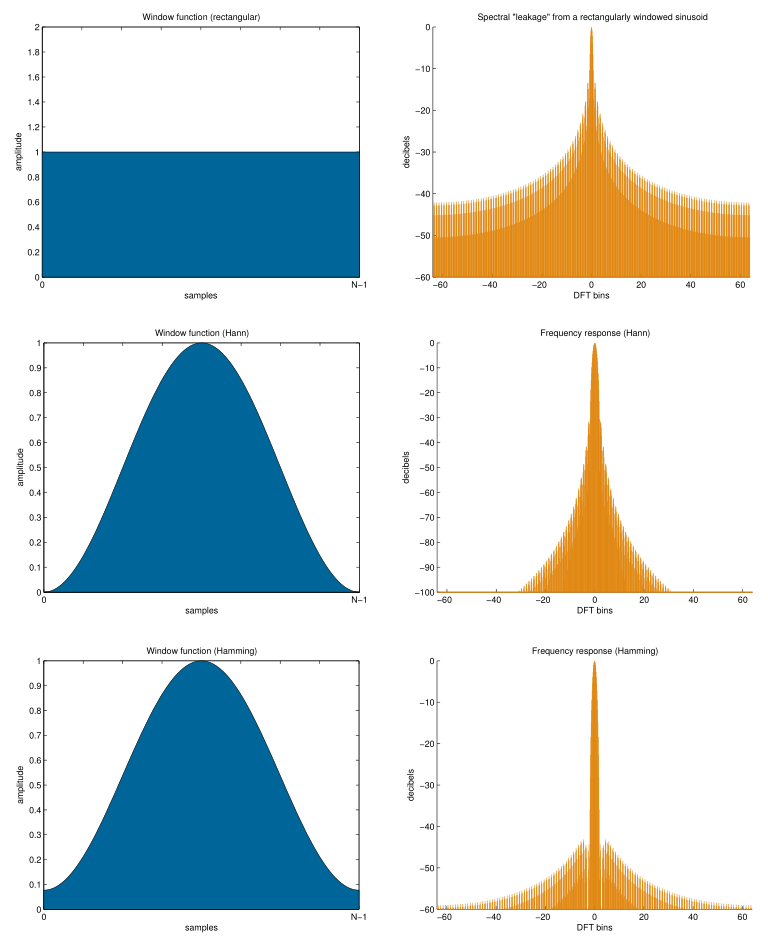


Рисунок 3.2 – Разные окна и соответствующие им утечки сигнала

Чтобы создать из 512 или 1024 массивов 128, можно создать новый ряд массивов, используя усреднённые значения энергий нескольких частотных диапазонов. Исходя из того, что человеческий слух имеет логарифмическую шкалу слышимости, то при сокращении данных следует воспользоваться этой особенностью, и использовать для каждого усреднённого массива не константное значение частотных диапазонов, а линейно повышать их для каждого последующего массива [10]. Тогда данные в каждом массиве будут усреднённым значением из исходных массивов, где вычисляется по формулам (3.1) и (3.2).

, для 1024 замеров (3.1)

, для 2048 замеров (3.2)

Для расчётов темпа песни, был выбран способ подсчётов музыкальных долей с помощью нахождения пиков спектрального потока, измеряющего, насколько быстро изменяется мощность спектра сигнала [11], вычисляемого для потоков и значений энергии потока в точке по формуле (3.3).

(3.3)

*G* – это любая функция преобразования, которая может быть как логарифмической (для улучшения работы в области, близкой к нулю), так и линейной, вычисляемая для каждого [11][12]. Для генератора подойдёт и простое отображение , так как это увеличит быстродействие программы.

После подсчёта спектрального потока, следует отсечь отрицательную часть, так как для подсчёта пиков нужна только информация об увеличении энергии, и для каждого значения данного потока подсчитать среднее значение в окне [13]. Оптимальный размер окна составляет 10 значение вперёд и назад. Наконец, чтобы выделить пики из потока, следует увеличить среднее потока в некоторое количество раз и отсечь от исходных значений всё, что будет меньше полученного модифицированного среднего значения, как показано на рисунке 3.3. Для проекта было решено увеличивать среднее значение в 4 раза, после чего вычислять пики в полученных значениях, так как дальнейшие вычисления темпа песни будут слишком времязатратными, а влияние на генерируемую карту, в конечном итоге, окажется не столь критичным.

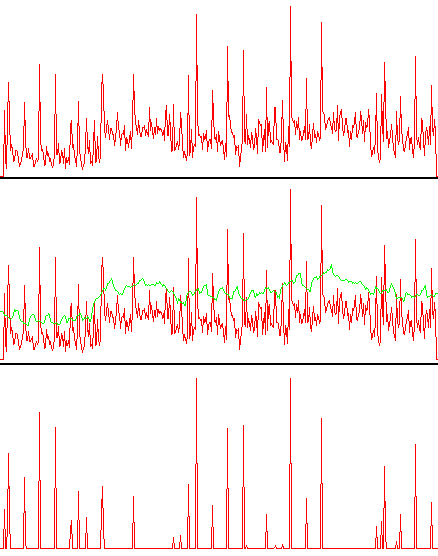


Рисунок 3.3 – Нахождение среднего значения спектрального потока

и отсечение нижних значений

Пик – это значение, которое больше предыдущего и последующего значений потока [11][12]. Так как среднее значение энергии было увеличено в 4 раза, то в отсечённом графике будет содержаться минимальное количество шума и слабых долей песни, потому можно приравнять каждый пик отсечённой области к одной доле.

Темп песни на каждом интервале определяется подсчётом количества долей на данном интервале, и умножением полученного количества на некое значение, чтобы привести темп к общепринятым *«долям (или ударам) в минуту»*. Для подсчёта было выбрано окно в 4 секунды, которое будет меньше у крайних значений, а полученное количество долей будет увеличено в 15 раз, и для крайних значений – в большее количество раз.

Чтобы получить большее количество данных, было решено найти пики в каждом частотном интервале по отдельности по тому же алгоритму, и подсчитать их количество.

3.1.2 Алгоритмы генерации карты

Генерация карты состоит из нескольких этапов:

1. Создание поверхности земли;
2. Разделение области на районы;
3. Размещение строений в полученных районах.

Поверхность земли было принято генерировать с помощью шума Перлина. Шумом Перлина называется градиентный шум [14], который можно представить геометрически с помощью бесконечной плоскости и случайных векторов. Из каждого узла сетки, равного целому числу, исходит случайный единичный вектор, который определяет направление наклона плоскости. Все значения точек на плоскости вычисляются исходя из направления ближайших четырёх векторов [15]. На рисунке 3.4 показано, как направление соседних векторов влияет на высоту каждой точки.

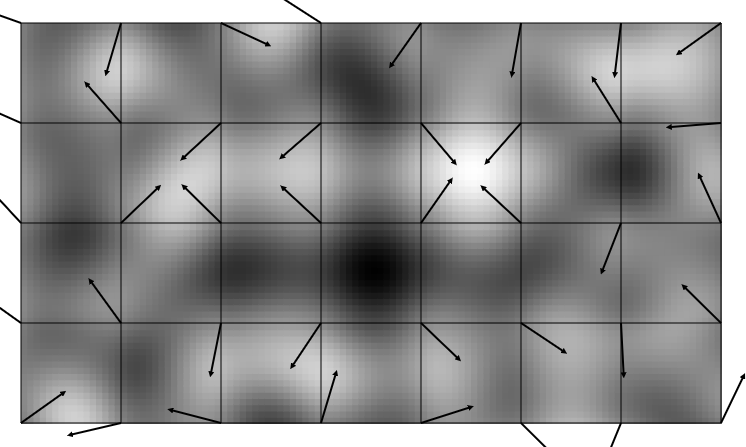


Рисунок 3.4 – Шум Перлина

Для сглаживания шума используются октавы – значения шума с частотой вдвое большей исходной и меньшей амплитудой [15]. Данные складываются, и на выходе получается многомерный шум, который показан на рисунке 3.5.

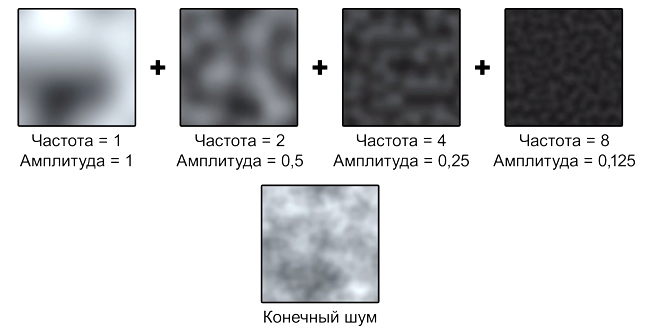


Рисунок 3.5 – Шум Перлина после сложения с октавами

Для получения разных значений, можно использовать случайный сдвиг октав и сдвиг самого шума Перлина полностью. Случайный сдвиг октав будет подсчитываться с использованием генератора случайных чисел, зерном которого будет общее количество замеров выбранной для генерации песни, а сдвигом для карты будет служить количество пиков в 122 и 123 частотном интервале.

Размер квадратной карты в точках зависит от длины песни. Сторона квадрата равна половине количества замеров темпа песни, округлённой вниз. Замеры темпа служат для выявления расстояния между соседними точками. При значении темпа 50 и ниже ударов в минуту, расстояние между данной точкой и следующей будет равным 5 условных единиц. При темпе в 300 и больше ударов в минуту, расстояние сужается до 0,5 условных единиц. Остальные значения линейно интерполируются между этими двумя точками. Первая половина замеров служит для расчёта значений по оси x, вторая половина – для значений по оси z. Значение по оси y берётся из шума Перлина и умножается на коэффициент крутизны уклона местности. Результат создания карты поверхности показан на рисунке 3.6.

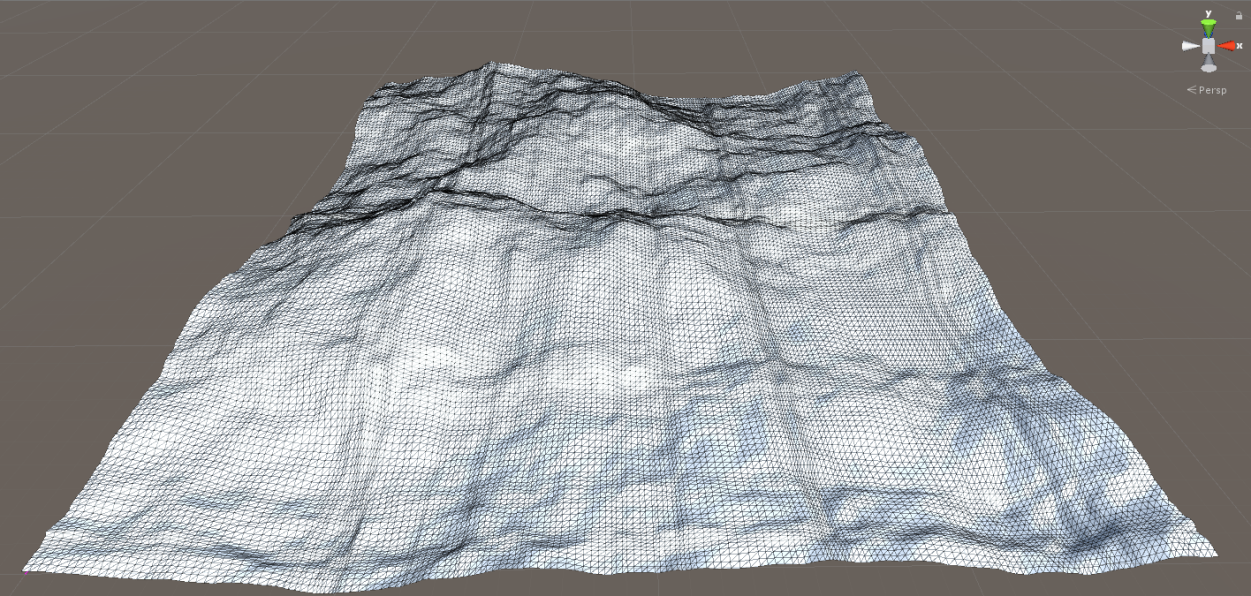


Рисунок 3.6 – Карта поверхности

После создания карты поверхности, её площадь разбивается на районы с помощью диаграммы Вороного, сглаженной алгоритмом Ллойда, и заполняется первоначальными точками для размещения зданий [16].

*Диаграмма Вороного* множества точек представляет собой разбиение исходной плоскости на несколько областей, где все точки каждой области находят ближе к одному элементу множества , чем ко всем остальным [17].

С помощью итераций алгоритма Ллойда появляется возможность превратить области случайного размера диаграммы Вороного на области схожего размера. Каждая итерация представляет собой следующие действия: после построения диаграммы, необходимо найти центр распределения для каждой области и переместить элемент множества , принадлежащий этой области, на позицию центра распределения, после чего заново построить диаграмму Вороного для нового множества точек. После нескольких итераций, все области диаграммы станут схожего размера [18]. Пример работы алгоритма представлен на рисунке 3.7.

Все районы получают свой тип: горный для местности, содержащей точки выше определённого уровня; водный для местности, содержащей точки ниже определённого уровня; обычный для всех остальных. Карта районов с точками для размещения зданий показана на рисунке 3.8.

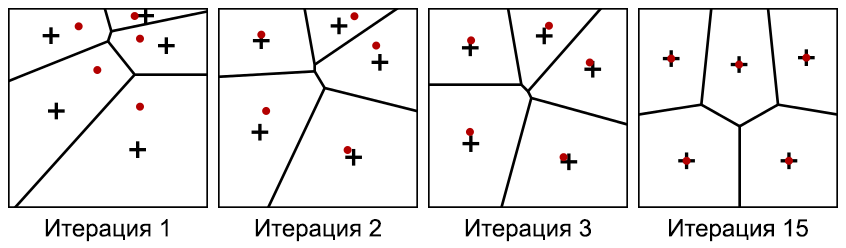


Рисунок 3.7 – Итерации алгоритма Ллойда

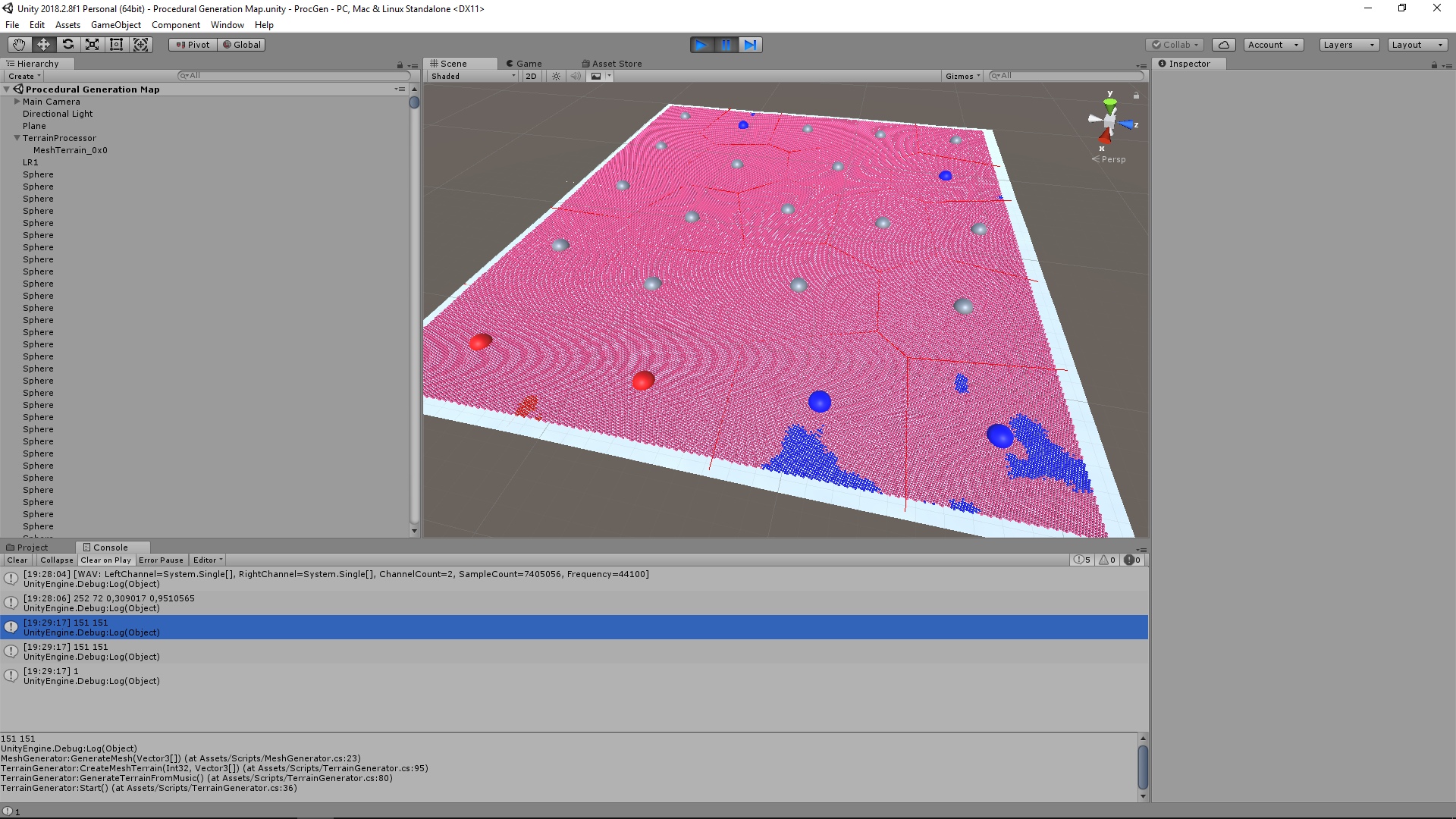


Рисунок 3.8 – Карта районов

В качестве точки отсчёта выбирается центр наиболее близкого к середине карты стандартного района. Если таковых нет, то выбирается центр горного района, либо водного, если горных районов также нет.

Объекты на карте делятся на шесть типов: главное здание, обычные здания, стены, дороги, индустриальные украшения, природные украшения. *Главное здание* всегда одно и размещается оно максимально близко к точке отсчёта. *Обычные здания* размещаются в пределах района и в количестве, равном количеству пиков в одном из частотных интервалов, высчитанном на этапе обработки музыкального произведения. *Стены* размещаются на границах города, *дороги* и *индустриальные украшения* – в пределах этого самого города. *Природные украшения* размещаются за пределами города. Каждый объект имеет свой тип местности, в котором его можно размещать.

Размещение объектов ведётся от точки отсчёта по районам. Когда место в одном районе заканчивается, алгоритм начинает размещать здания в следующем незанятом районе.

Для земли и воды имеется возможность выбрать материал, из которого они будут сделать, и определить, будут ли они случайного цвета. Свойства размещаемых объектов и местности указаны в таблице 1.

Таблица 1 – Свойства размещаемых объектов

|  |  |
| --- | --- |
| **Название свойства** | **Значение свойства** |
| Name | Название объекта, которое будет отображаться в интерфейсе пользователя и меню |
| Building Prefab | Внутриигровая модель объекта для размещения |
| Length | Длина объекта |
| Width | Ширина объекта |
| Building Type | Тип размещаемого объекта |
| Region Type | Тип региона, в котором необходимо разместить объект |

3.2 Обзор средств разработки

Разработка программного обеспечения ведётся на основе мультиплатформенного игрового движка *Unity*, комплекса программных средств, облегчающих и ускоряющих процесс разработки игры [19]. На данный момент существуют тысячи игр, созданных с помощью данного игрового движка, как крупными разработчиками, так и независимыми студиями.

Редактор Unity имеет простой интерфейс, с возможностью манипулирования игровыми объектами при помощи мыши и состоящим из различных окон, благодаря чему появляется возможность производить отладку игры прямо в редакторе. Unity поддерживает два языка сценариев: C# и модифицированный JavaScript.

C#, в отличие от JavaScript, является строго типизированным языком, а также намного чаще используется в разработке игр на Unity. Для написания кода на языке C# в Unity по умолчанию используется среда разработки Microsoft Visual Studio.

Одно из главных преимуществ Unity обеспечивает модульная система компонентов, которая используется для конструирования игровых объектов. «Компоненты» в такой системе представляют собой комбинируемые пакеты функциональных элементов, поэтому объекты создаются как наборы компонентов, а не как жесткая иерархия классов. Такая система являет собой более гибкий подход к объектно-ориентированному программированию, в котором игровые объекты создаются путем объединения, а не наследования [20]. Разница между данными двумя подходами продемонстрирована на рисунке 3.9.

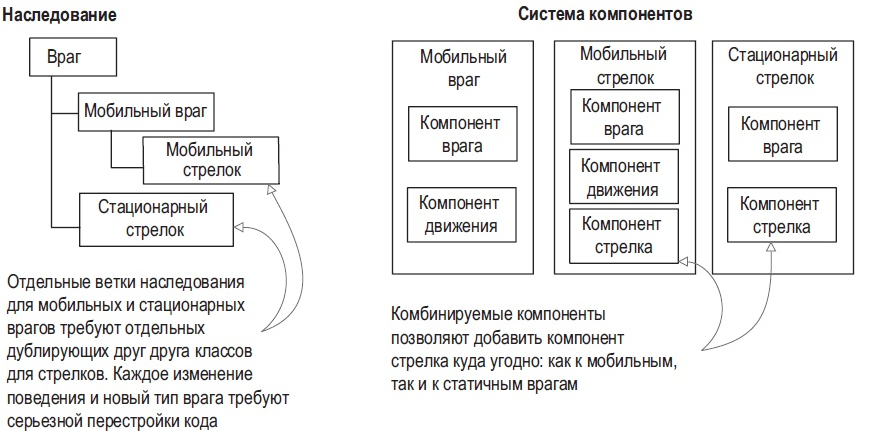


Рисунок 3.9 – Сравнение наследования и системы компонентов

Проект в Unity делится на сцены – отдельные уровни со своим набором объектов, сценариев, и настроек. Сцены могут содержать в себе игровые объекты, которые могут быть пустыми – не иметь какого-либо визуального представления в игре. Объекты, в свою очередь содержат наборы компонентов, с которыми и взаимодействуют пользовательские сценарии. Также, в атрибутах объектов присутствует название (Unity допускает наличие двух и более объектов с одинаковыми названиями), тег (метка) и слоя, на котором данный объект должен отображаться.

У любого объекта на сцене обязательно присутствует компонент *«Transform»* – он хранит в себе координаты местоположения, поворота и размеров объекта по трём пространственным осям. У объектов с видимой геометрией по умолчанию присутствует дополнительные компоненты: *«Mesh Filter»*, определяющий геометрию игрового объекта, и *«Mesh Renderer»*, визуализирующий объект [21].

Объекты могут просчитывать столкновения с помощью так называемых *коллайдеров* (*collider*) – компонентов объекта, ограничивающих область взаимодействия с другими объектами [20]. Коллайдеры представляют собой один или несколько простейших геометрических тел, привязанных к позиции исходного объекта, повторяющих в общих чертах его силуэт.

Коллайдеры из простейших фигур используют для уменьшения времени вычисления процессором пересечений границ сложных геометрических тел, из которых состоит большая часть игровых объектов. У такого подхода есть один недостаток – игровые объекты могут частично накладываться друг на друга.

Стандартный объект «Куб» и его компоненты продемонстрированы на рисунке 3.10.

Также, в Unity встроена поддержка физики твёрдых тел и ткани. В редакторе существует система наследования объектов: дочерние объекты будут повторять все изменения позиции, поворота и масштаба родительского объекта. Система наследования показана на рисунке 3.11.

Сценарии в редакторе можно крепить к объектам в виде отдельных компонентов. Для этого создаваемый в сценарии класс необходимо наследовать от класса *«MonoBehaviour»*.

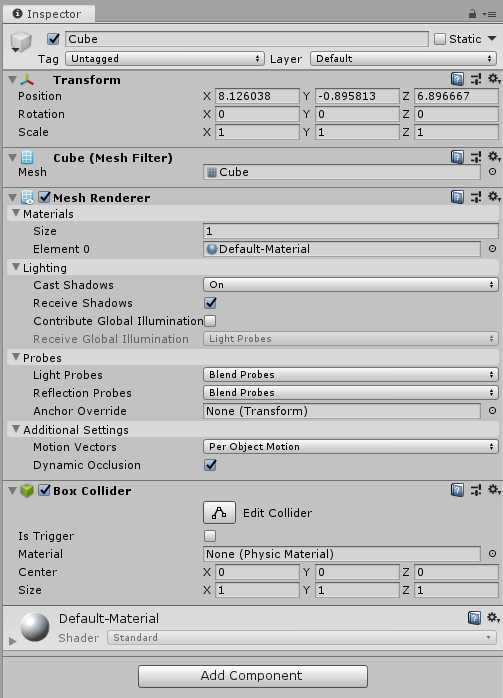


Рисунок 3.10 – Компоненты объекта «Куб»



Рисунок 3.11 – Система наследования в Unity

Редактор имеет компонент для создания собственной анимации, а также поддерживает импортирование предварительно созданной во внешнем 3D-редакторе анимации вместе с моделью объекта.

Игровой движок имеет систему *уровней детализации* (*Level Of Detail* или *LOD*), которая снижает детализацию сложных моделей при отдалении камеры игрока, и систему Occlusion culling, которая не визуализирует геометрию и столкновения объектов, находящихся вне поля зрения камеры, что снижает нагрузку на центральный процессор.

Unity позволяет программистам с лёгкостью делиться разработанным кодом и интегрировать его в свою работу благодаря системе *игровых ресурсов* (или *игровых ассетов*) – неделимых цифровых объектов, использующихся при разработке компьютерной игры.

Для быстрого и удобного обмена созданными ресурсами, Unity имеет цифровую площадку под названием *«Asset Store»*, где каждый может разместить созданные собственноручно ресурсы. Внешний вид площадки показан на рисунке 3.12.



Рисунок 3.12 – Asset Store

4 Проектирование программного обеспечения

4.1 Общая структура в редакторе Unity

Инструментарий Unity автоматически создаёт интерфейс прикреплённых к игровым объектам сценариев для классов, наследованных от MonoBehaviour. Однако существует возможность создать собственный интерфейс редактора с помощью других специализированных классов Unity.

Так как правила редактора не позволяют создавать экземпляры классов, наследованных от MonoBehaviour, и для их использования, необходимо прикрепить сценарий с данным классом к объекту, то для оптимальной структуры приложение должно иметь один класс, наследованный от MonoBehaviour, один класс для реализации интерфейса и использующиеся для работы приложения иные классы, экземпляры которых создаются в первом. Схема данной структуры показана на рисунке 4.1.

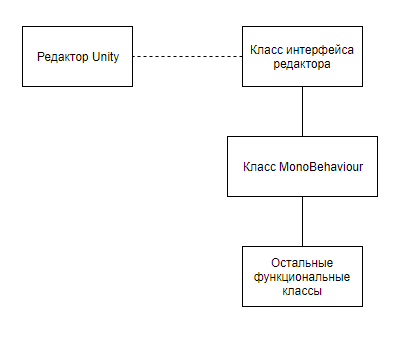
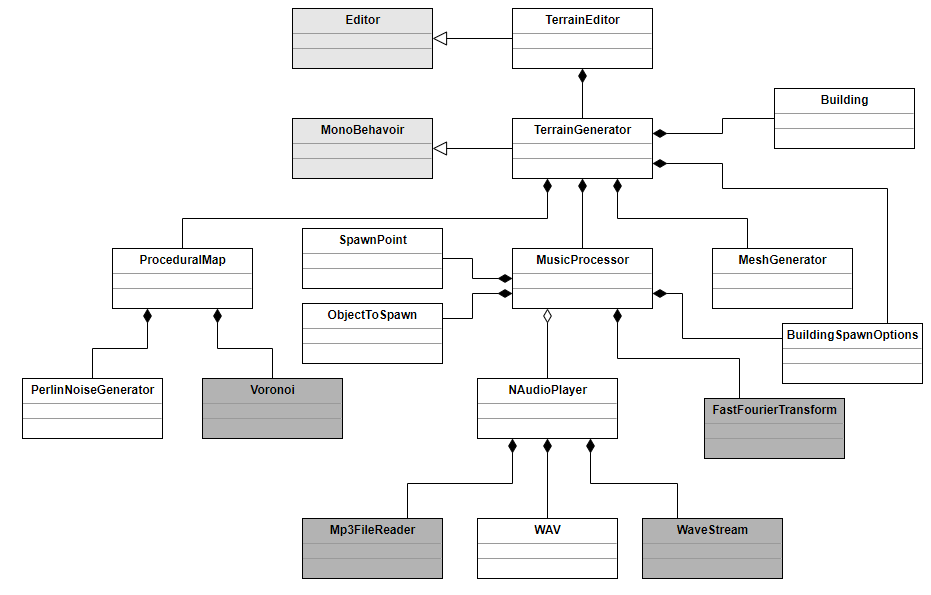


Рисунок 4.1 – Структура приложения

4.2 Модули и классы

Приложение является единым модулем внутри редактора Unity. Диаграмма классов разработанного приложения представлена на рисунке 4.2. Тёмно-серым выделены классы, взятые из сторонних библиотек. Светло-серым выделены стандартные классы Unity, на основе которых реализуются функции редактора.

 Рисунок 4.2 – Диаграмма классов

Реализованные классы:

* *TerrainEditor*. Реализует интерфейс класса TerrainGenerator для работы с пользователем;
* *TerrainGenerator.* Главный класс модуля, управляет работой остальных классов и размещает объекты игрового окружения в окружающем пространстве;
* *Building.* Структура для хранения входных данных объектов для размещения;
* *MusicProcessor.* Обрабатывает музыкальные данные различного типа, создаёт первоначальный набор данных для генерации карты местности;
* *BuildingSpawnOptions.* Структура с местоположением и поворотом размещаемых объектов;
* *SpawnPoint.* Структура с данными обо всех существующих точках для размещения объектов;
* *ObjectToSpawn.* Структура для хранения данных объектов для размещения, используемая во время расчётов;
* *NAudioPlayer.* Статический класс для обработки музыкальных файлов расширения MP3;
* *WAV.* Вспомогательный класс, переводит двоичные данные из музыкального файла в наборы значений амплитуды звука для левого и правого звуковых каналов;
* *ProceduralMap.* Основной класс, которые просчитывает карту, создаёт наборы значений высот и координаты точек размещения объектов;
* *PerlinNoiseGenerator.* Рассчитывает значения шума Перлина;
* *MeshGenerator.* Создаёт массив полигональных сеточных объектов, определяющих форму поверхности карты местности.

Сторонние классы:

* *Voronoi.* Рассчитывает границы диаграммы Вороного для карты местности (Источник кода «csDelaunay», лицензия «MIT» [22]);
* *FastFourierTransform.* Реализует алгоритм быстрого преобразования Фурье (Источник кода «NAudio», лицензия «Ms-PL» [23]);
* *Mp3FileReader.* Вспомогательный класс, считывающий данные MP3 файлов (Источник кода «NAudio», лицензия «Ms-PL»);
* *WaveStream.* Вспомогательный класс, реализовывающий считывание данных MP3 файлов в потоке (Источник кода «NAudio», лицензия «Ms-PL»).

4.2.1 TerrainEditor

На рисунке 4.3 представлено описание класса TerrainEditor на языке UML.

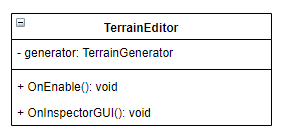


Рисунок 4.3 – Класс TerrainEditor

Поля:

* *generator*. Ссылка на класс, для которого реализуется интерфейс.

Методы:

* *OnEnable*. Метод, определяющий функции, которые будут выполнены при старте работы.
* *OnInspectorGUI*. Метод, определяющий, что и как будет отображено в пользовательском интерфейсе.

4.2.2 TerrainGenerator

На рисунке 4.4 представлено описание класса TerrainGenerator на языке UML.

Поля:

* *musicClip.* Музыкальное произведение, на основе которого алгоритм будет генерировать карту;
* *buildings.* Массив, хранящих входные данные объектов для размещения на карте. Размер по умолчанию – 50. Максимальный размер – 90;
* *terrainMaterial.* Материал, из которого будет сделана земля;
* *randomTerrainColor.* Переменная, определяющая, будет ли цвет земли генерироваться случайным образом или будет взят из свойств материала. Значение по умолчанию – «false»;
* *waterMateria.* Материал, из которого будет сделана вода;
* *randomWaterColor.* Переменная, определяющая, будет ли цвет воды генерироваться случайным образом или будет взят из свойств материала. Значение по умолчанию – «false»;

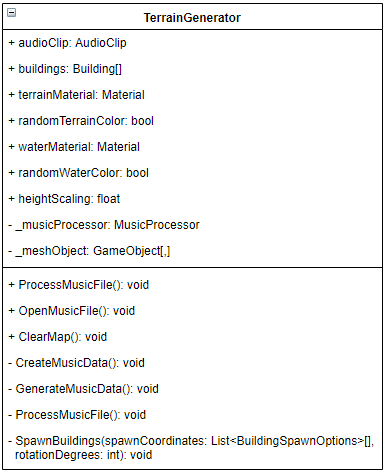


Рисунок 4.4 – Класс TerrainGenerator

* *heightScaling.* Переменная, определяющая коэффициент масштабирования высоты карты. Значение по умолчанию – 1;
* *\_musicProcessor.* Ссылка на класс музыкального обработчика;
* *\_meshObject.* Массив, хранящий геометрию поверхности карты уровня.

Методы:

* *OpenMusicFile.* Открывает внешний музыкальный файл и преобразовывает его в формат музыкального файла редактора Unity;
* *CreateMusicData.* Инициализирует обработку музыкального файла музыкальным процессором;
* *ClearMap.* Удаляет созданную карту;
* *ProcessMusicFile.* Считывает данные из обработанного музыкального файла, создаёт наборы объектов для размещения, вызывает методы дальнейшего создания карты;
* *GenerateTerrainFromMusic.* Создаёт объект класса карты, также инициализирует просчёт местности и точек для размещения объектов;
* *CreateMeshTerrain.* Создаёт поверхность карты местности на основе рассчитанных данных;
* *SpawnBuildings.* Размещает объекты на поверхности карты.

4.2.3 Building

На рисунке 4.5 представлено описание структуры Building на языке UML.

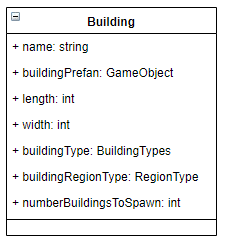


Рисунок 4.5 – Структура Building

Поля:

* *name.* Название здания, отображающееся в редакторе;
* *buildingPrefab.* Модель объект, который будет размещён на карте;
* *length.* Длина объекта;
* *width.* Ширина объекта;
* *buildingType.* Тип объекта;
* *buildingRegionType.* Тип региона, в котором следует разместить объект;
* *numberBuildingsToSpawn.* Количество объектов данного типа, которые должны быть размещены на карте.

4.2.4 MusicProcessor

На рисунке 4.6 представлено описание класса MusicProcessor на языке UML.

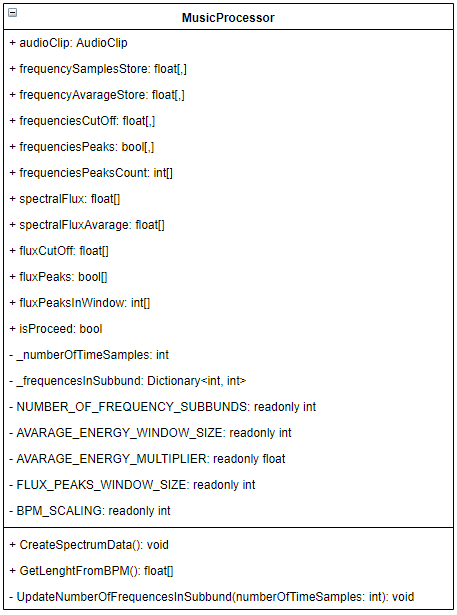


Рисунок 4.6 – Класс MusicProcessor

Поля:

* *audioClip.* Музыкальный файл, который необходимо обработать;
* *frequencySamplesStore.* Массив, содержащий частотные амплитуды, просчитанные через определённый интервал времени;
* *frequencyAvarageStore.* Массив, содержащий среднюю амплитуду частотных спектров на некотором интервале времени;
* *frequenciesCuttOff.* Массив, содержащий значения амплитуды частотных спектров, которые больше среднего;
* *frequenciesPeaks.* Массив, содержащий информацию, присутствует ли пик на частотном интервале в некую единицу времени;
* *frequenciesPeaksCount.* Массив, содержащий количество пиков в частотных интервалах.
* *spectralFlux.* Массив, содержащий амплитуду спектрального потока, просчитанную через определённый интервал времени;
* *spectralFluxAvarage.* Массив, содержащий среднюю амплитуду спектрального потока на некотором интервале времени;
* *fluxCuttOff.* Массив, содержащий большие среднего значения амплитуды спектрального потока;
* *fluxPeaks.* Массив, содержащий информацию, присутствует ли в спектральном потоке пик в некую единицу времени;
* *fluxPeaksInWindow.* Массив, содержащий количество пиков в спектральном потомке в интервал времени;
* *\_numberOfTimeSamples.* Количество замеров амплитуды музыкального произведения, которые будут считаны для обработки за один раз. Значение по умолчанию – 1024;
* *isProceed.* Переменная, хранящее значение, обработан ли музыкальный файл. Значение по умолчанию – «false»;
* *\_frequencesInSubbund.* Словарь, хранящий количество исходных частотных спектров, которые должны быть слиты в конкретный производный частотный спектр;
* *NUMBER\_OF\_FREQUENCY\_SUBBUNDS.* Количество производных частотных спектров. Значение по умолчанию – 128;
* *AVARAGE\_ENERGY\_WINDOW\_SIZE.* Длина окна в одну сторону, в котором будет вычисляться среднее значение амплитуды. Значение по умолчанию – 10;
* *AVARAGE\_ENERGY\_MULTIPLIER.* Коэффициент, на который будет умножено среднее значение амплитуды при расчётах. Значение по умолчанию – 4;
* *FLUX\_PEAKS\_WINDOW\_SIZE.* Длина окна в одну сторону, в котором будет подсчитываться количество пиков спектрального потока. Значение по умолчанию – 86;
* *BPM\_SCALING.* Шаг, через который будет просчитываться темп песни. Значение по умолчанию – 24.

Методы:

* *CreateSpectrumData.* Обрабатывает музыкальный файл и заполняет массивы с данными;
* *UpdateNumberOfFrequencesInSubbund.* Высчитывает количество исходных частотных спектров, которые должны быть слиты в конкретный производный частотный спектр, в зависимости от количества проводимых за одну итерацию замеров амплитуды;
* *GetLenghtFromBPM.* Возвращает массив длины участка карты в зависимости от темпа песни на неким интервале времени.

4.2.5 BuildingSpawnOptions

На рисунке 4.7 представлено описание структуры BuildingSpawnOptions на языке UML.

Поля:

* *buildingCoordinates.* Месторасположение размещаемого объекта;
* *horisontal.* Поворот размещаемого объекта.

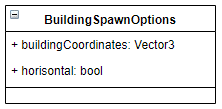


Рисунок 4.7 – Структура BuildingSpawnOptions

4.2.6 SpawnPoint

На рисунке 4.8 представлено описание структуры SpawnPoint на языке UML.

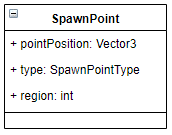


Рисунок 4.8 – Структура SpawnPoint

Поля:

* *pointPosition.* Позиция точки для размещения;
* *type.* Тип точки;
* *region.* Номер региона, к которому относится точка.

4.2.7 ObjectToSpawn

На рисунке 4.9 представлено описание структуры ObjectToSpawn на языке UML.

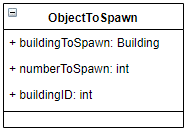


Рисунок 4.9 – Структура ObjectToSpawn

Поля:

* *buildingToSpawn.* Структура объекта, который необходимо разместить;
* *numberToSpawn.* Поле, использующееся для подсчёта оставшегося количества объектов, которые необходимо разместить;
* *buildingID.* Внутренний номер-идентификатор объекта, использующийся во время расчётов.

4.2.8 NAudioPlayer

На рисунке 4.10 представлено описание класса NAudioPlayer на языке UML.

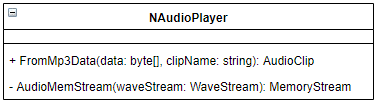


Рисунок 4.10 – Класс NAudioPlayer

Методы:

* *FromMp3Data.* Статический метод, конвертирующий данные из файла формата MP3 в объект AudioClip редактора Unity;
* *AudioMemStream.* Статический метод, создающий байтовые данные, необходимые для конструктора класса WAV.

4.2.9 WAV

На рисунке 4.11 представлено описание класса WAV на языке UML.

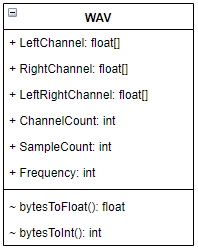


Рисунок 4.11 – Класс WAV

Поля:

* *LeftChannel.* Набор значений амплитуды звуковой волны музыкального произведения для левого канала;
* *RightChannel.* Набор значений амплитуды звуковой волны музыкального произведения для правого канала;
* *LeftRightChannel.* Набор значений амплитуды звуковой волны музыкального произведения для левого и правого каналов;
* *ChannelCount.* Количество каналов;
* *SampleCount.* Количество значений амплитуды в музыкальном файле;
* *Frequency.* Частота дискретизации музыкального произведения.

Методы:

* *bytesToFloat.* Статический метод, использующийся во время преобразования данных из музыкального файла в объект редактора Unity. Преобразует считанные байты в число типа float;
* *bytesToInt.* Статический метод, использующийся во время преобразования данных из музыкального файла в объект редактора Unity. Преобразует считанные байты в число типа int;

4.2.10 ProceduralMap

На рисунке 4.12 представлено описание класса ProceduralMap на языке UML.

Поля:

* *terrainCoordinates.* Массив, содержащий координаты всех вершин поверхности карты;
* *maxX2DCoord.* Длина карты по оси X на виде сверху;
* *maxY2DCoord.* Длина карты по оси Y на виде сверху;
* *terrainDimension.* Длина стороны карты, измеряемая в вершинах;
* *rotationDegrees.* Угол поворота точек для размещения объектов;



Рисунок 4.12 – Класс ProceduralMap

* *spawnCoordinates.* Список координат для размещения объектов;
* *\_frequenciesPeaksCount.* Массив, хранящий количество пиков в частотных интервалах, полученных после обработки музыкального произведения;
* *\_vorDiagram.* Ссылка на диаграмму Воронного для данной карты;
* *\_dots.* Массив точек, на основе которых строилась диаграмма Воронного;
* *\_regionsStartPositions.* Массив, хранящий начальные точки для размещения объектов по каждому региону;
* *\_buildingsProperties.* Массив данных об объектах для размещения;
* *\_pointsRG.* Генератор случайных чисел для данной карты;
* *MOUNTAIN\_TRESHOLD.* Высота, выше которой местность считается горой. Значение по умолчанию – 0,88;
* *WATERLINIE.* Высота, ниже которой местность считается водой. Значение по умолчанию – -0,75;
* *COAST\_TRESHOLD.* Высота окна, в котором местность выше и ниже водной границы считается берегом. Значение по умолчанию – 0,05;

Методы:

* *CreateMap.* Главный метод, создающий карту;
* *CreateAreasFromVoronoiDiagrams.* Вспомогательный метод, создающий диаграмму Воронного для карты;
* *CreatePointsForSpawning.* Вспомогательный метод, создающий список точек для размещения объектов;
* *FindRegion.* Вспомогательный метод, находящий регион, в котором размещена некоторая точка;
* *FindNearest.* Вспомогательный метод, находящий ближайшую к заданной позиции точку из списка;
* *FindDotHeight.* Вспомогательный метод, находящий высоту точки;
* *FormRegionsStartPositions.* Вспомогательный метод, формирующий начальные позиции для размещения объектов для всех регионов;
* *FindNextRegion.* Вспомогательный метод, находящий следующий по списку ближайший к центральному регион;
* *FindStartingSpawnDot.* Вспомогательный метод, вычисляющий начальную точку для размещения объекта;
* *FindRoadSpawnDot.* Вспомогательный метод, вычисляющий начальную точку для размещения объекта типа «Дорога»;
* *FindSpawnPointType.* Вспомогательный метод, вычисляющий тип точки, необходимый для размещения конкретного объекта;
* *PlaceSpawnPoint.* Вспомогательный метод, создающий точку для размещения конкретного объекта.

4.2.11 PerlinNoiseGenerator

На рисунке 4.13 представлено описание класса PerlinNoiseGenerator на языке UML.

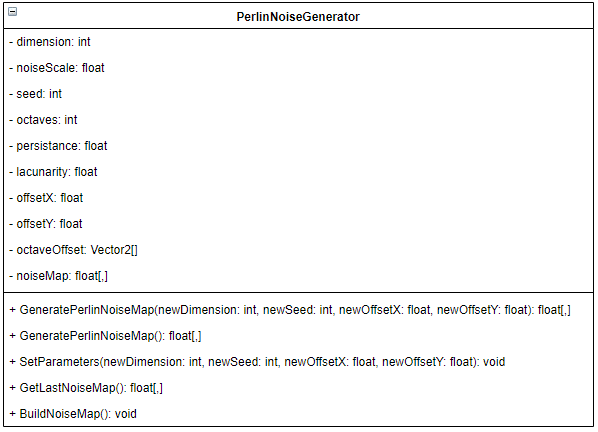


Рисунок 4.13 – Класс PerlinNoiseGenerator

Поля:

* *dimension.* Размер создаваемого шума Перлина;
* *noiseScale.* Масштаб шума Перлина. Значение по умолчанию – 80;
* *seed.* Зерно для генератора случайных чисел. Значение по умолчанию – 32;
* *octaves.* Количество октав. Значение по умолчанию – 20;
* *persistance.* Константа, на которую будет умножаться значение амплитуды каждый шаг. Значение по умолчанию – 0,5;
* *lacunarity.* Константа, на которую будет умножаться значение частоты каждый шаг. Значение по умолчанию – 2;
* *offsetX.* Сдвиг шума по оси X. Значение по умолчанию – 0;
* *offsetY.* Сдвиг шума по оси Y. Значение по умолчанию – 0;
* *octaveOffset.* Массив, хранящий значения сдвигов октав;
* *noiseMap.* Массив высот, сгенерированный шумом Перлина.

Методы:

* GeneratePerlinNoiseMap. Основной метод, инициализирующий создание карты шума Перлина и возвращающий массив высот;
* SetParameters. Метод, необходимый для установки параметров сдвига карты шума Перлина;
* GetLastNoiseMap. Возвращает последнюю сгенерированную карту шума Перлина;
* BuildNoiseMap. Вспомогательный метод, который генерирует массив высот с помощью шума Перлина;

4.2.12 MeshGenerator

На рисунке 4.14 представлено описание класса MeshGenerator на языке UML.

Поля:

* *\_height*. Высота карты на виде сверху;
* *\_width*. Ширина карты на виде сверху;
* *NUMBER\_OF\_DOTS\_IN\_ONE\_MESH.* Константа, определяющая максимальный размер стороны одного участка поверхности карты местности в точках. Значение по умолчанию – 240;
* *meshArray.* Массив участков поверхности карты местности.

Методы:

* *GenerateMesh.* Создаёт поверхность карты местности в игровом пространстве.

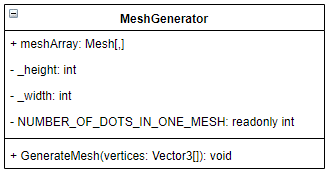


Рисунок 4.14 – Класс MeshGenerator

5 Реализация программного обеспечения

Благодаря наследованию от классов Unity, редактор автоматически подхватывает созданный модуль в качестве компонента игровых объектов. На рисунке 5.1 показан пустой игровой объект *«TerrainProcessor»* с компонентом созданного модуля.

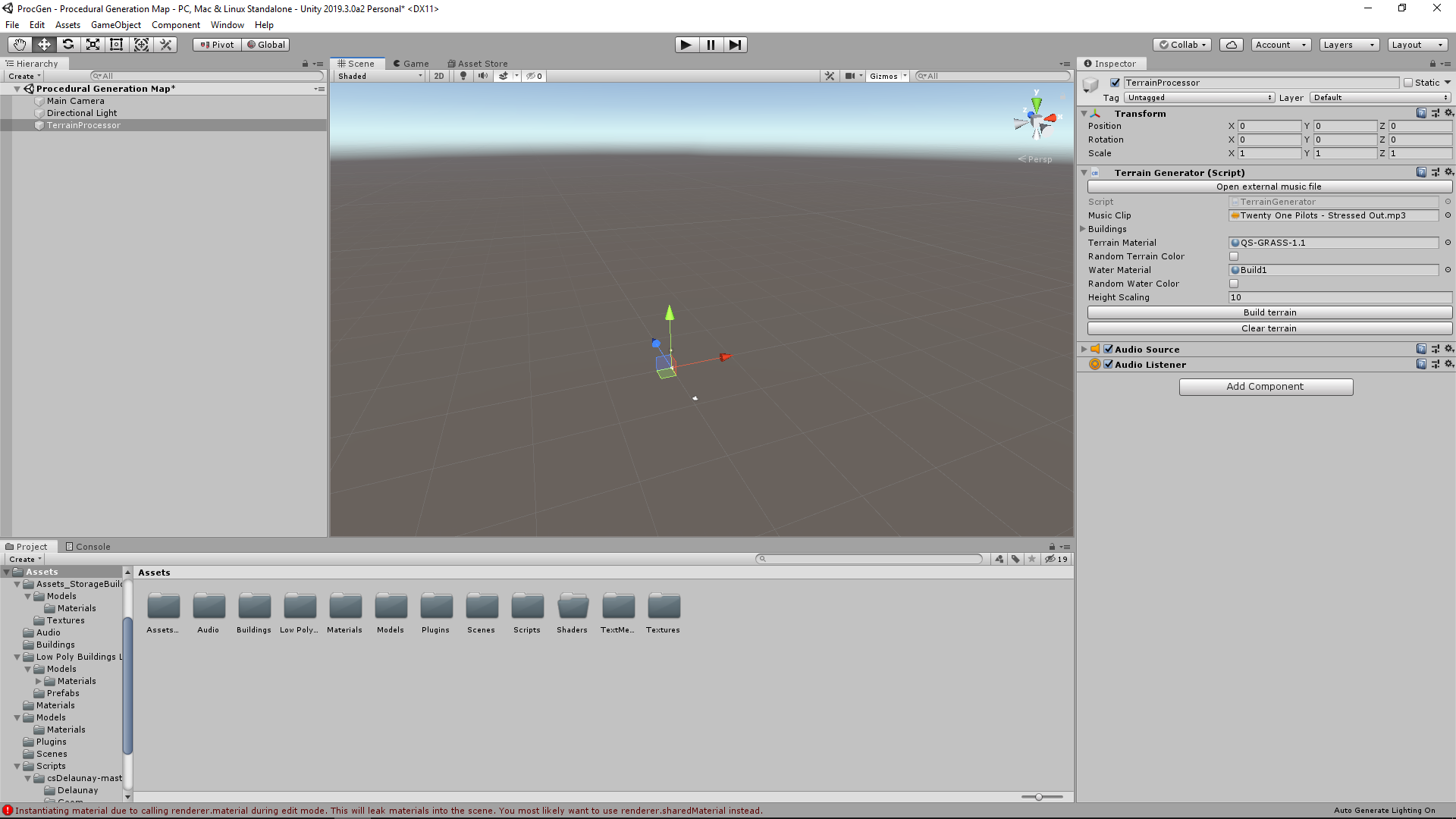


Рисунок 5.1 – Класс MeshGenerator

На рисунке 5.2, интерфейс модуля рассмотрен более детально. Кроме полей, модуль имеет несколько кнопок:

* *«Open external music file»* для открытия внешнего музыкального файла без добавления его в материалы игрового проекта;
* *«Build terrain»* для постройки карты уровня с выбранными объектами;
* *«Clear terrain»* для очистки карты уровня.

Данные полей могут быть изменены прямо в интерфейсе редактора, без необходимости открывать код и что-то менять.

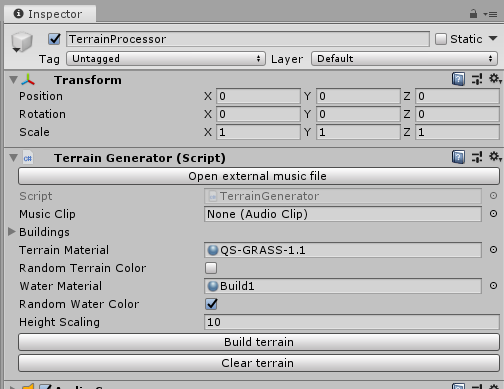


Рисунок 5.2 – Интерфейс модуля

При нажатии на кнопку «Open external music file», открывается окно выбора музыкального файла, который будет преобразован в класс Unity «Audio Clip». Это окно показано на рисунке 5.3.

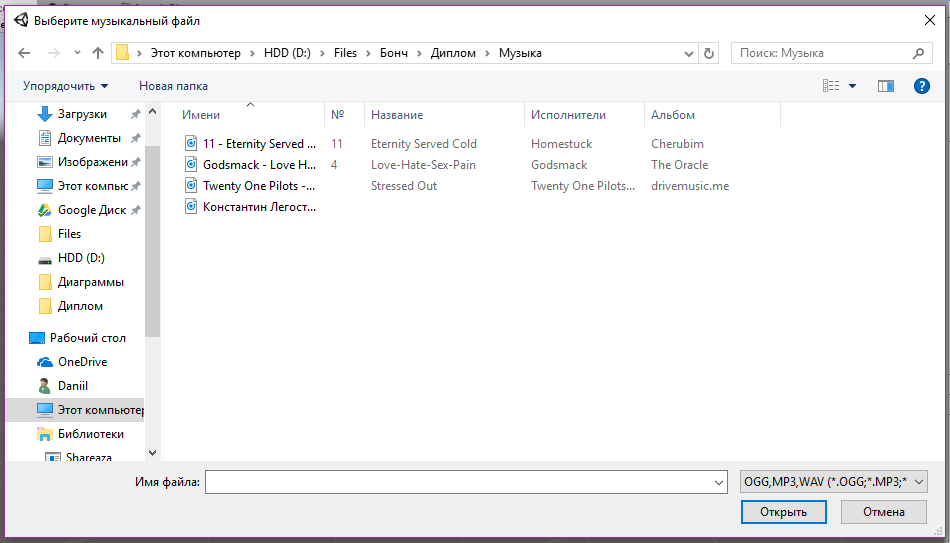


Рисунок 5.3 – Окно выбора музыкального файла

Загруженный в модуль внешний музыкальный файл продемонстрирован на рисунке 5.4.

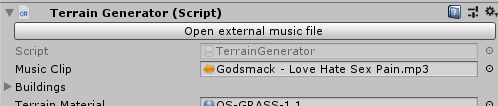


Рисунок 5.4 – Загруженный музыкальный файл

Как показано на рисунке 5.5, в поля материалов загружаются одноимённые компоненты Unity, созданные на основе шейдеров.

*Шейдер* – это программа, описывающая визуальную составляющую геометрических объектов и предназначенная для исполнения видеокартой.

*Материал* – это экземпляр шейдера с установленными параметрами (текстурой, базовым цветом, прозрачностью и прочим).

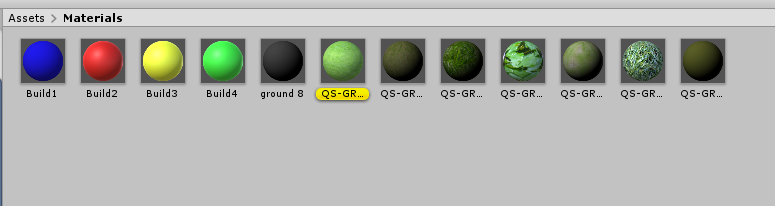
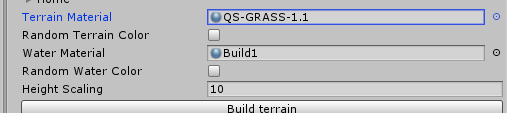


Рисунок 5.5 – Загруженные материалы

На рисунке 5.6 показана реализация интерфейса массива загружаемых объектов для дальнейшей расстановки. Поле «Building Prefab» предназначено для загрузки уже готовых объектов зданий, копии которых будут расставлены на карте.

Пользователь должен, также, ввести ширину и длину объекта и выбрать его тип и местность, на которой данный объект следует располагать. Пользовательское имя объекта не влияет на работу алгоритма и несёт исключительно декоративную цель.

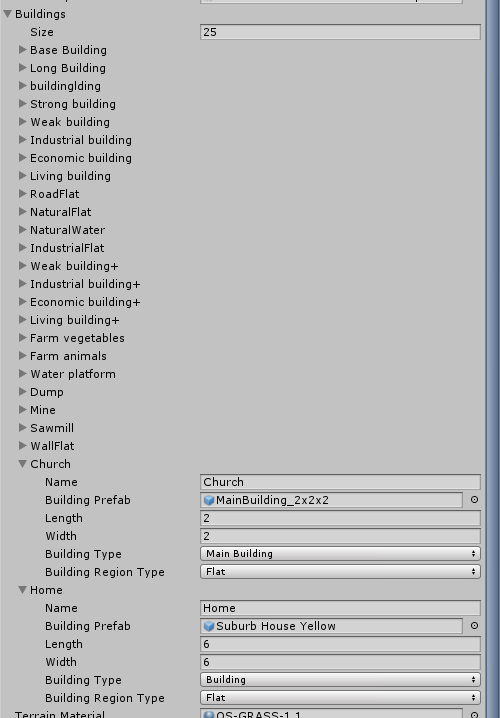


Рисунок 5.6 – Массив загружаемых объектов

После нажатия на кнопку «Build terrain», при условии загруженного музыкального файла, модуль создаст карту местности с пользовательскими объектами на ней. Скорость создания карты и её размер зависит от длины музыкального произведения и количества объектов в массиве.

Пример построенной карты местности для музыкального произведения длиной 2 минуты 30 секунд показан на рисунке 5.7.



Рисунок 5.7 – Построенная карта местности

При выборе случайного цвета местности и водной поверхности, этот цвет также будет зависеть от музыкального произведения. Пример построенной карты местности со случайными цветами для музыкального произведения длиной 3 минуты 30 секунд показан на рисунке 5.8.

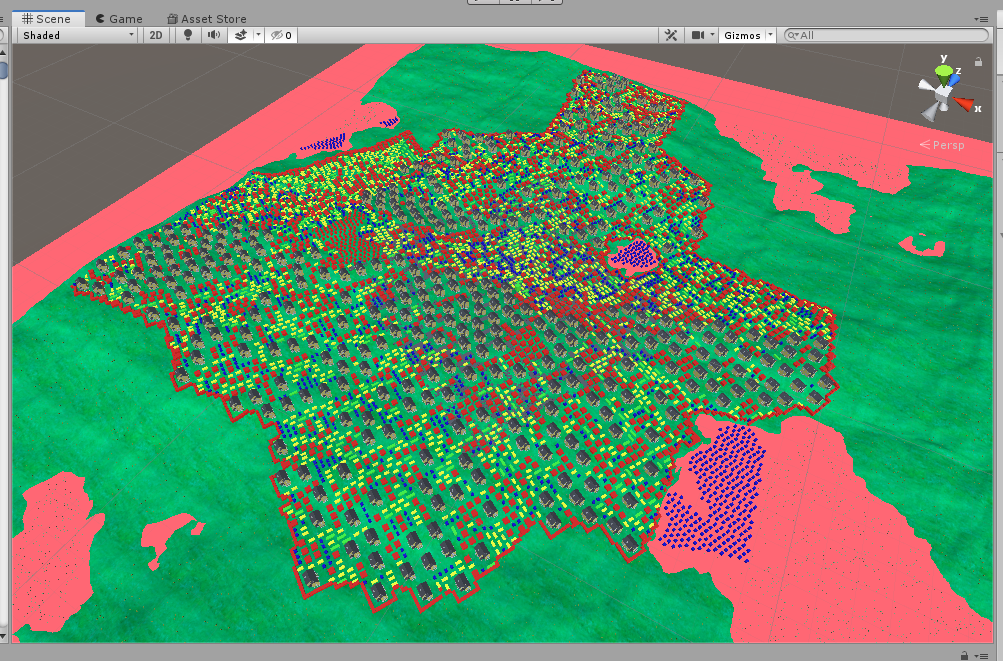


Рисунок 5.8 – Карта местности со случайными цветами

6 Тестирование программного обеспечения

6.1 Обзор методов тестирования

Известно большое количество методов тестирования программного обеспечения, однако не все из них подходят для тестирования разрабатываемого ПО. Рассмотрим те из них, которые имеет смысл провести:

1. *Функциональное тестирование* – тестирование программного обеспечения для проверки соответствия его функциональным требованиям, возможности решать поставленные задачи.
2. *Нагрузочное тестирование* – подвид тестирования, включающий в себя сбор показателей и определение производительности вместе со временем отклика программы в ответ на внешний запрос с целью установления соответствия требованиям, предъявляемым к данной программе.

6.2 Результаты тестов и принятые меры

6.2.1 Функциональное тестирование

Во время тестирования стояла задача не только проверить документированный функционал программы, но и работу не по инструкции.

Была обнаружена ошибка, позволяющая размещать на карте больше, чем 90 типов объектов, что вызывало ошибку с выходом за границы массива. Ошибка была исправлена и при старте обработки была добавлена проверка, которая отсекает лишние объекты при их наличии.

Также, была обнаружена недоработка, из-за которой после сохранения сцены в Unity и загрузки её заново, поле с музыкальным файлом было заполнено, однако оно было не обработано. Для устранения этой недоработки, была добавлена проверка на наличие в поле музыкального произведения после нажатия кнопки построения карты.

При постройке новой карты, она создавалась поверх старой. Ошибка исправлена удалением предыдущей карты перед созданием новой.

При удалении карты, удалялся также и сам главный объект, содержащий код сценария построения карты. Ошибка была исправлена.

Было обнаружено, что при загрузке музыкального файла маленького размера, программа работала некорректно. Был установлен минимальный лимит на количество замеров музыкального произведения – 1 000 000.

Обнаружена ошибка при вводе размеров объектов ниже единицы. Ошибка исправлена и минимальный размер объекта теперь равен единице.

Была обнаружена и исправлена ошибка, из-за которой в ближайшем к точке (0, 0, 0) регионе карты не размещались объекты.

6.2.2 Нагрузочное тестирование

Во время работы, программа обрабатывает музыкальный файл и строит карту исходя из входных данных. Для разных продолжительности музыкальных произведений и количества зданий требуется разное время на обработку.

Были проведены замеры временных затрат на построение карты в зависимости от количества замеров в музыкальных произведениях и объектов для размещения на карте. Результаты отображены в таблице 2.

Таблица 2 – Результаты нагрузочного тестирования

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Количество замеров**  **Объекты** | 6 682 752  замеров | 8 809 344 замеров | 8 925 696 замеров | 22 095 360 замеров |
| 25 объектов | 7 секунд | 12 секунд | 25 секунд | 2 минуты  10 секунд |
| 75 объектов | 40 секунд | 1 минута  38 секунд | 1 минута  13 секунд | 16 минут |

Из полученных данных видно, что при увеличении объёма данных, время работы приложения увеличивается экспоненциально. Однако, так как даже для 22 миллионов замеров 75 типов объектов заполняли всю карту, то данные пиковые значение редко когда встретятся пользователям при работе с программой.

Время работы программы для средних значений объектов и замеров является удовлетворительным.