|  |  |
| --- | --- |
| Герб МГТУ | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

|  |  |
| --- | --- |
|  | |
| ФАКУЛЬТЕТ | «Робототехника и комплексная автоматизация» |
| КАФЕДРА | «Системы автоматизированного проектирования (РК-6)» |
|  | |
| **РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА** | |
| ***К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ*** | |
| ***НА ТЕМУ:*** | |
| ***«Разработка сетевых методов автоматизированного запуска распределённой системы выделенных серверов Unreal Engine 4»*** | |
|  | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент | РК6-41М |  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Боженко Д.В.** |
| (Группа) | (Подпись, дата) | (Фамилия И.О.) |
| Руководитель ВКР | | | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Витюков Ф.А.** |
| (Подпись, дата) | (Фамилия И.О.) |
| Нормоконтролёр | | | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Грошев С.В.** |
| (Подпись, дата) | (Фамилия И.О.) |

|  |
| --- |
|  |
| *2025 г.* |

|  |  |  |  |
| --- | --- | --- | --- |
| **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** | | | |
|  | УТВЕРЖДАЮ | | |
| Заведующий кафедрой | | | РК6 |
| (Индекс) |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | Карпенко А.П. | |
| (Фамилия И.О.) | |
| « » 202 г. | | | |

|  |  |  |
| --- | --- | --- |
| **ЗАДАНИЕ** | | |
| **на выполнение выпускной квалификационной работы** | | |
| Студент группы | РК6-41M | |
| Боженко Дмитрий Владимирович | | |
| (фамилия, имя, отчество) | | |
| Тема квалификационной работы: «Разработка сетевых методов автоматизированного запуска распределённой системы выделенных серверов Unreal Engine 4» | | |
| При выполнении ВКР: | | |
| Используются / Не используются | | Да/Нет |
| 1) Литературные источники и документы, имеющие гриф секретности | | Нет |
| 2) Литературные источники и документы, имеющие пометку «Для служебного пользования», иных пометок, запрещающих открытое опубликование | | Нет |
| 3) Служебные материалы других организаций | | Нет |
| 4) Результаты НИР (ОКР), выполняемой в МГТУ им. Н.Э.Баумана | | Нет |
| 5) Материалы по незавершенным исследованиям или материалы по завершенным исследованиям, но ещё не опубликованные в открытой печати | | Нет |
| Тема квалификационной работы утверждена распоряжением по факультету РК №\_\_\_\_\_\_\_\_\_\_\_ от « » 2025 г. | | |
| **Техническое задание** | | |
| **Часть 1.** *Реализация программы менеджера серверов и программы-демона.* | | |
| *Реализовать программу менеджера серверов для регистрации запросов на создание сессии. Реализовать программу-демона для запуска экземпляров выделенного сервера Unreal Engine 4. Разработать сетевой протокол и настроить сетевое взаимодействие между сервисами. Запустить систему на машинах Linux-Debian.* | | |
| **Часть 2.** *Реализация механизма авторизации для контроля доступа к запуску выделенного сервера.* | | |
| *Проанализировать доступные механизмы авторизации пользователя для контроля доступа к запуску выделенного сервера Unreal Engine 4. Реализовать выбранный метод и предоставить обоснование решения. Проанализировать общепринятые подходы хранения данных о транзакциях на стороне клиента, выбрать наиболее подходящий метод и реализовать его.* | | |
| **Часть 3.** *Реализация механизма балансировки загрузки и мониторинга нагрузки.* | | |
| *Реализовать балансировку загрузки клиентов по запущенным экземплярам выделенных серверов. Обеспечить систему механизмом мониторинга нагрузки каждого выделенного сервера и узла системы через консольный интерфейс.* | | |
| **Оформление выпускной квалификационной работы** | | |
| Расчетно-пояснительная записка на \_\_\_ листах формата A4. | | |
| Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.): | | |
| |  | | --- | | *Работа содержит 8 графических листов формата A1.* | |  | |  | |  | | | |
| Дата выдачи задания: «9» февраля 2025 г. | | |
| В соответствии с учебным планом выпускную квалификационную работу выполнить в полном объеме в срок до «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_\_ г. | | |

|  |  |  |
| --- | --- | --- |
| **Руководитель квалификационной работы** | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Витюков Ф.А.** |
|  | (Подпись, дата) | (Фамилия И.О.) |
| **Студент** | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Боженко Д.В.** |
|  | (Подпись, дата) | (Фамилия И.О.) |

|  |
| --- |
| Примечание:  1. Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедре. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** | | | | | |
| **ФАКУЛЬТЕТ** | РК |  | УТВЕРЖДАЮ | | |
| **КАФЕДРА** | РК6 | Заведующий кафедрой | | | РК6 |
| (Индекс) |
| **ГРУППА** | РК6-41М | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | Карпенко А.П. | |
| (Фамилия И.О.) | |
| « » 202 г. | | | | | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **КАЛЕНДАРНЫЙ ПЛАН** | | | | | | | | |
| **выполнения выпускной квалификационной работы** | | | | | | | | |
| Студент группы | | | РК6-41М | | | | | |
| Боженко Дмитрий Владимирович | | | | | | | | |
| (фамилия, имя, отчество) | | | | | | | | |
| Тема квалификационной работы: «Разработка сетевых методов автоматизированного запуска распределённой системы выделенных серверов Unreal Engine 4» | | | | | | | | |
| **№ п/п** | **Наименование этапов выпускной квалификационной работы** | | **Сроки выполнения этапов** | | **Отметка о выполнении** | |
| **план** | **факт** | **Должность** | **ФИО, подпись** |
|  | Задание на выполнение работы. Формулирование проблемы, цели и задач работы | | 09.02.2025 | \_\_\_\_\_\_\_\_\_ | Руководитель ВКР | Витюков Ф.А. |
|  | 1 часть. Аналитическая часть | | 18.02.2025 | \_\_\_\_\_\_\_\_\_ | Руководитель ВКР | Витюков Ф.А. |
|  | Утверждение окончательных формулировок решаемой проблемы, цели работы и перечня задач | | 28.02.2025 | \_\_\_\_\_\_\_\_\_ | Заведующий кафедрой | Карпенко А.П. |
|  | 2 часть. Практическая часть 1 | | 15.04.2025 | \_\_\_\_\_\_\_\_\_ | Руководитель ВКР | Витюков Ф.А. |
|  | 3 часть. Практическая часть 2 | | 01.05.2025 | \_\_\_\_\_\_\_\_\_ | Руководитель ВКР | Витюков Ф.А. |
|  | 1-я редакция работы | | 15.05.2025 | \_\_\_\_\_\_\_\_\_ | Руководитель ВКР | Витюков Ф.А. |
|  | Подготовка доклада и презентации | | 24.05.2025 | \_\_\_\_\_\_\_\_\_ | Студент | Боженко Д.В. |
|  | Заключение руководителя | | 17.05.2025 | \_\_\_\_\_\_\_\_\_ | Руководитель ВКР | Витюков Ф.А. |
|  | Допуск работы к защите на ГЭК (нормоконтроль) | | 31.05.2025 | \_\_\_\_\_\_\_\_\_ | Нормоконтролер | Грошев С.В. |
|  | Внешняя рецензия | | 03.06.2025 | \_\_\_\_\_\_\_\_\_ |  |  |
|  | Защита работы на ГЭК | | 06.06.2025 | \_\_\_\_\_\_\_\_\_ |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Студент | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Боженко Д.В. | Руководитель ВКР | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Витюков Ф.А. |
|  | (подпись, дата) | (Фамилия И.О.) |  | (подпись, дата) | (Фамилия И.О.) |

// Здесь должно быть направление на защиту

АННОТАЦИЯ

В представленной работе описан поэтапный процесс разработки распределенной системы автоматизированного запуска выделенных серверов Unreal Engine 4. Работа содержит в себе подробное техническое описание каждого этапа разработки системы и описание проведенного исследования в области решаемой задачи и обоснования выбранного решения. В работе описаны и проиллюстрированы следующие этапы реализации проекта: разработка программы менеджера серверов; разработка программы-демона; реализация механизма авторизации для контроля доступа к запуску выделенного сервера; реализация механизма хранения информации о транзакциях на стороне клиента; реализация системы мониторинга нагрузки приложения через консольный интерфейс. Расчетно-пояснительная записка содержит 36 с., 7 рис., 5 табл., 7 источников, 1 прил.

Тип работы: выпускная квалификационная работа.

Тема работы: «Разработка сетевых методов автоматизированного запуска распределённой системы выделенных серверов Unreal Engine 4».

Объекты исследований: разработка сетевых методов, разработка распределенной системы, развертывание и запуск выделенных серверов Unreal Engine.

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

*Unreal Engine 4 (UE 4)* – движок Unreal Engine 4, разрабатываемый и поддерживаемый компанией Epic Games.

*TCP* – протокол транспортного уровня, обеспечивающий надежную доставку данных.

*Epic Online Services (EOS)* – встроенная в движок подсистема, предоставляющая низкоуровневые программные инструменты.

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 13](#_Toc168490510)

[1. Постановка задачи 15](#_Toc168490511)

[2. Обзор литературы 17](#_Toc168490512)

[3. Создание 3d-сцены 19](#_Toc168490513)

[3.1. Создание ландшафта в ПО WorldMachine 19](#_Toc168490514)

[3.2. Импорт ландшафта в Unreal Engine 22](#_Toc168490515)

[3.3. Использование системы PCG в Unreal Engine 5 для наполнения сцены растительностью 24](#_Toc168490516)

[3.4. Место отдыха и приготовления пищи 29](#_Toc168490517)

[3.5. Дикие звери (противники) 32](#_Toc168490518)

[3.6. Создание диких зверей на уровне 40](#_Toc168490519)

[4. Создание персонажа 42](#_Toc168490520)

[4.1 Атрибуты персонажа 42](#_Toc168490521)

[4.2 Инвентарь персонажа 44](#_Toc168490522)

[4.3 Класс персонажа 45](#_Toc168490523)

[4.4 Настройка анимаций персонажа 48](#_Toc168490524)

[4.5 Действия персонажа 54](#_Toc168490525)

[4.5.1 Охота на диких зверей 55](#_Toc168490526)

[4.5.2 Рубка дерева 57](#_Toc168490527)

[4.5.3 Приготовление пищи 59](#_Toc168490528)

[4.5.4 Добавление древесины в костер 61](#_Toc168490529)

[4.5.5 Поглощение пищи 63](#_Toc168490530)

[5. Интеграция llama.cpp в Unreal Engine 65](#_Toc168490531)

[5.1 Сборка llama.cpp 65](#_Toc168490532)

[5.2 Параметры сборки llama.cpp 65](#_Toc168490533)

[5.3 Результаты сборки llama.cpp 66](#_Toc168490534)

[5.4 Создание плагина для работы с LLM в Unreal Engine 66](#_Toc168490535)

[5.5 Добавление файлов llama.cpp в плагин 68](#_Toc168490536)

[5.6 LLMComponent и LLMWorker 69](#_Toc168490537)

[5.7 Выполнение запросов к языковой модели из дерева поведения 71](#_Toc168490538)

[6. Примеры работы приложения 72](#_Toc168490539)

[ЗАКЛЮЧЕНИЕ 74](#_Toc168490540)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 75](#_Toc168490541)

[ПРИЛОЖЕНИЕ А. Графическая часть выпускной квалификационной работы 77](#_Toc168490542)

ВВЕДЕНИЕ

Одним из важнейших аспектов работы с Unreal Engine является запуск выделенных серверов, которые обеспечивают многопользовательскую работу, взаимодействие между клиентами и поддержание сетевой архитектуры. Однако автоматизация процесса запуска и управления такими серверами представляет собой сложную задачу, особенно в контексте масштабируемых и распределенных систем, где необходимо учитывать множество факторов, включая доступность ресурсов, производительность и отклик системы.

На данный момент существует немалое количество программных инструментов, которые позволяют масштабировать любое приложение и автоматизировать запуск процессов, в том числе и для приложения на Unreal Engine. Ярким примером таких инструментов является Kubernetes, Terraform и Minikube. Однако они могут быть эффективно использованы не для каждой задачи и не для всех типов приложений. Поэтому крайне важно подходить к выбору методов автоматизации запуска приложения и его процессов взвешенно, с учетом специфики задачи; проводить исследовательскую работу в области решения проблемы; итеративно оценивать и внедрять выбранные технологии в проект.

Целью данной работы является разработка и внедрение системы автоматического запуска выделенных серверов Unreal Engine 4 которая позволяет инициировать запуск сессий по запросу с клиентского приложения и в автоматизированном режиме распределять нагрузку между запущенными экземплярами приложения. Также система должна обеспечивать прозрачность, предоставляя детализированные логи о каждом процессе и запросе, происходящем в ходе её работы, а также фиксировать эти данные для последующего анализа.

Для достижения данной цели

Также важным компонентом является аутентификация при запуске серверов, что обеспечивает безопасный доступ к системным ресурсам и позволяет гарантировать, что только авторизованные пользователи могут управлять запуском или подключением к выделенным серверам. Аутентификация играет критическую роль в предотвращении несанкционированного использования серверных мощностей и защиты пользовательских данных в многопользовательской среде.

Кроме того, внедрение возможностей работы в оффлайн-режиме повышает удобство использования системы для конечных пользователей. Работа в оффлайн-режиме позволяет пользователям сохранять доступ к ключевым функциям и данным, даже в случае временных перебоев с сетью.

1. Постановка задачи

В рамках данной работы рассматривается экспериментальный сценарий использования больших языковых моделей – выбор действия персонажа в зависимости от значений его атрибутов и иных параметров системы.

Рассматриваемая система состоит из следующих компонентов.

* Персонаж (агент, управляемый с помощью языковой модели).
* Окружающая среда:
  + жилище (палатка, костер);
  + лес (деревья, дикие звери).

Персонаж имеет следующие атрибуты.

* Здоровье (health) .
  + При персонаж полностью здоров.
  + При наступает смерть.
  + Персонаж может получить урон во время охоты, тем самым понизить здоровье.
  + Сон восстанавливает здоровье. Скорость регенерации зависит от уровня голода :

где – базовая (максимальная) скорость регенерации, – масштабирующий коэффициент влияния голода.

* Усталость (fatigue) .
  + При персонаж находится в максимально отдохнувшем состоянии.
  + При он мгновенно засыпает, прерывая любое действие.
  + Во время сна персонаж отдыхает, уровень усталости «восстанавливается» со скоростью .
* Голод (hunger) .
  + При персонаж полностью сыт.
  + При наступает смерть.
* Температурный режим (temperature) .
  + При персонаж умирает от переохлаждения
  + При персонаж умирает от перегрева.
* Скорость перемещения (movement speed) .
  + Зависит от уровней голода и усталости :

где – базовая (максимальная) скорость перемещения, – масштабирующий коэффициент влияния голода, – масштабирующий коэффициент влияния усталости.

* Урон, наносимый диким зверям (damage) .

Возможные действия (занятия) персонажа.

* Охота.
  + Дикие звери могут нанести персонажу урон.
  + В результате охоты персонаж получает сырое мясо.
* Рубка леса.
  + В результате повала дерева персонаж получает дрова.
  + Дрова попадают в инвентарь персонажа, вместимость инвентаря ограничена.
* Сон.
  + Во время сна персонаж не может выполнять иные действия.
  + Сон восстанавливает уровень здоровья и усталости.
* Приготовление пищи.
  + Персонаж может пожарить сырое мясо на костре.
  + Жарка занимает некоторое время .
* Поглощение пищи.
  + Персонаж может съесть жареное мясо и восстановить свое здоровье.

2. Обзор литературы

Ввиду новизны тематики на момент написания данной работы, статей, описывающих решение подобных задач, крайне мало.

Одной из близких по тематике статей является публикация Банджо Обайоми (Banjo Obayomi) «14 больших языковых моделей сражались в 314 матчах игры Street Fighter. Вот, кто победил» [2]. Автор статьи решил столкнуть две большие языковые модели (LLM) в поединках игры Street Fighter, заставил их соревноваться друг с другом.

Каждый персонаж контролировался своей LLM. Схема управления заключалась в формировании параметризованного запроса, учитывающего текущее состояние игровой сессии (местоположение персонажа, его счет и здоровье). В состав запроса также входили подсказки о возможных ходах и стратегиях.

Из полученного ответа на запрос извлекалось название движения персонажа, затем выполнялось его преобразование в набор игровых команд.

Автор провел 314 подобных матчей с 14 различными моделями и составил таблицу лидеров. Первое место среди всех моделей заняла Anthropic Claude 3 Haiku с 1613.45 очками, первое место среди локальных языковых моделей заняла Mistral 7B с 1490.06 очками.

В процессе исследования автор статьи сделал ряд интересных наблюдений. Иногда модели генерировали несуществующие или неприменимые ходы. Модель Anthropic Claude 2.1 отказывалась играть по причине «нежелания рекомендовать жестокие действия или стратегии». Каждая из моделей вырабатывала свой собственный стиль игры: некоторые играли более агрессивно, другие – более оборонительно, третьи все время использовали одно и то же действие.

Еще одной статьей с приближенной тематикой является публикация Chen Qian, Xin Cong и других «Коммуникативные агенты для разработки программного обеспечения» [3]. Авторы статьи создали ChatDev – команду разработки ПО, состоящую исключительно из больших языковых моделей.

Команда ChatDev работает по принципам, близким к методологии разработки ПО «водопад» (waterfall). Процесс разработки разделен на четыре этапа: проектирование, кодирование, тестирование и написание документации. К каждому этапу привлекается команда LLM-агентов, которые могут выполнять роли программиста, дизайнера, тестировщика, и т.д., общаться друг с другом и налаживать рабочий процесс.

Каждый этап разбивается на атомарные подзадачи с помощью цепочки общения. Это приводит к появлению двойных ролей, позволяет предлагать и проверять решения через контекстуальное общение, делает решение подзадач более эффективным.

В процессе тестирования и анализа система ChatDev зарекомендовала себя как крайне эффективный способ генерации программного кода – ей удалось завершить полный цикл разработки ПО менее, чем за семь минут, менее, чем за один доллар.

3. Создание 3d-сцены

3.1. Создание ландшафта в ПО WorldMachine

WorldMachine – программное обеспечение для процедурного создания 3d-ландшафтов. В данной работе была использована его бесплатная версия – World Machine Basic.

В отличие от традиционных подходов создания ландшафтов путем рисования трехмерными кистями, генерация в World Machine выполняется с помощью dataflow диаграмм. Пользователь работает с узлами (nodes) различных типов, передает данные между ними путем их соединения. Работа ведется с набором 2d-карт (maps), например, карт высот, цветовых карт, и т.д.

В WorldMachine узлы называются устройствами (devices) [4]. Каждое устройство имеет несколько входных и выходных портов (ports).

Основные типы устройств.

* Generators. Используются для создания новых карт с помощью алгоритмов процедурной генерации на основе шума.
* Combiners. Используются для объединения нескольких входных карт по выбранному правилу.
* Filters. Используются для преобразований входных карт.
* Output/View. Позволяют просмотреть результат генерации ландшафта во вьюпорте, а также экспортировать его в другой формат.

В данной работе был создан простой ландшафт небольшого размера (2x2 км) – равнина в окружении возвышенностей/скал по всему кругу. Выбор обоснован необходимостью уложиться в ограничения бесплатной версии (отсутствие возможности работы с отдельными секторами (tiles) карты, пониженное максимальное разрешение выходного файла), а также сделать края карты менее очевидными для пользователя.

Поскольку ландшафт создается для Unreal Engine 5, в настройках проекта WorldMachine был выбран соответствующий пресет, который позволяет задать подходящее для движка разрешение выходного файла (рисунок 1).

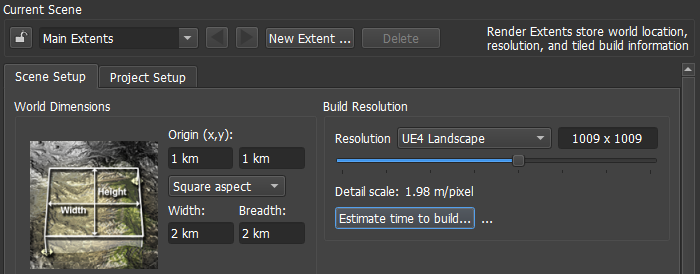


Рисунок 1 – Пресет ландшафта для Unreal Engine

Результирующий граф представлен на рисунке 2.

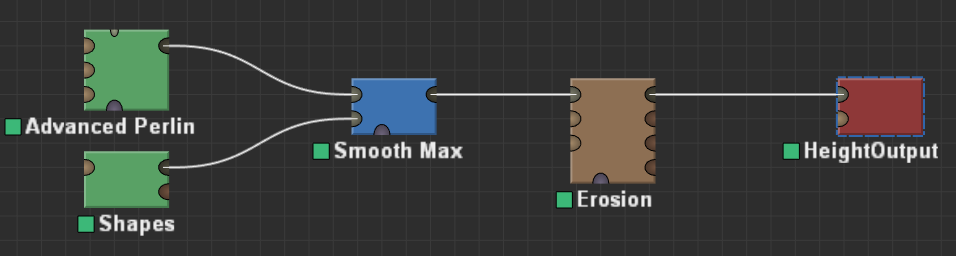


Рисунок 2 – Граф ландшафта

Основная форма ландшафта задается с помощью генератора Shapes. Путем вычитания цилиндра из параллелепипеда с искажением краев цилиндра была получен результат, представленный на рисунке 3.

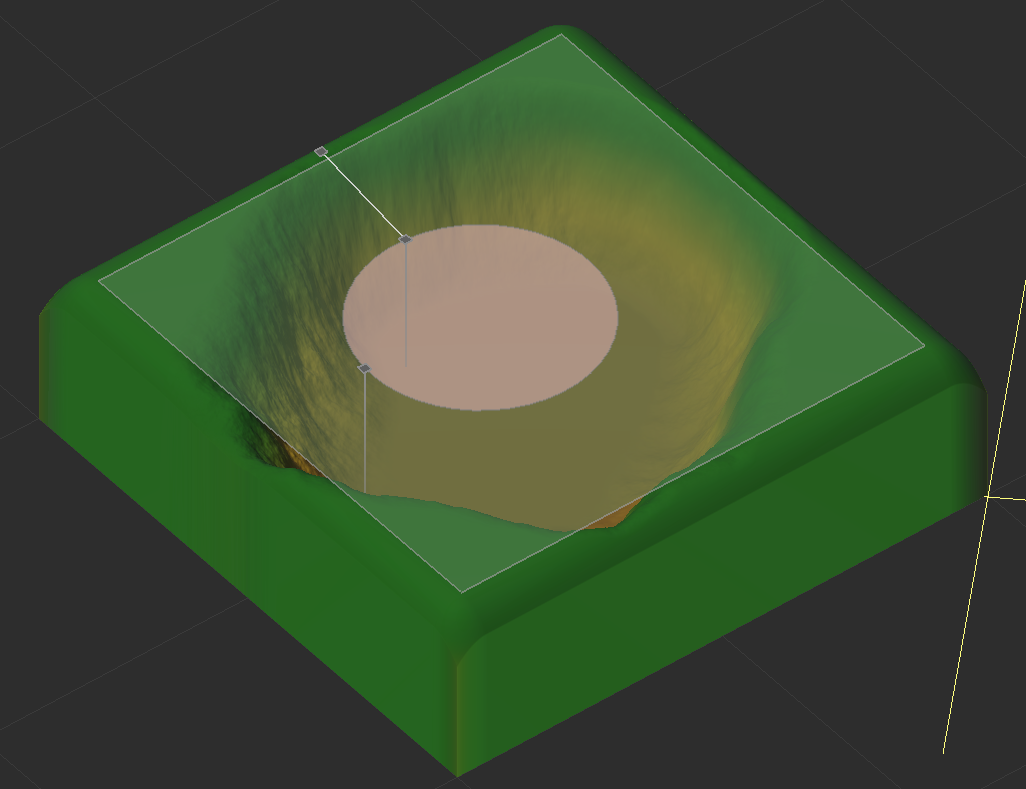


Рисунок 3 – Результат на выходе генератора Shapes

Для добавления неровностей на горизонтальные поверхности был использован генератор Advanced Perlin, позволяющий создать карту высот на основе шума Перлина. Результат изображен на рисунке 4.

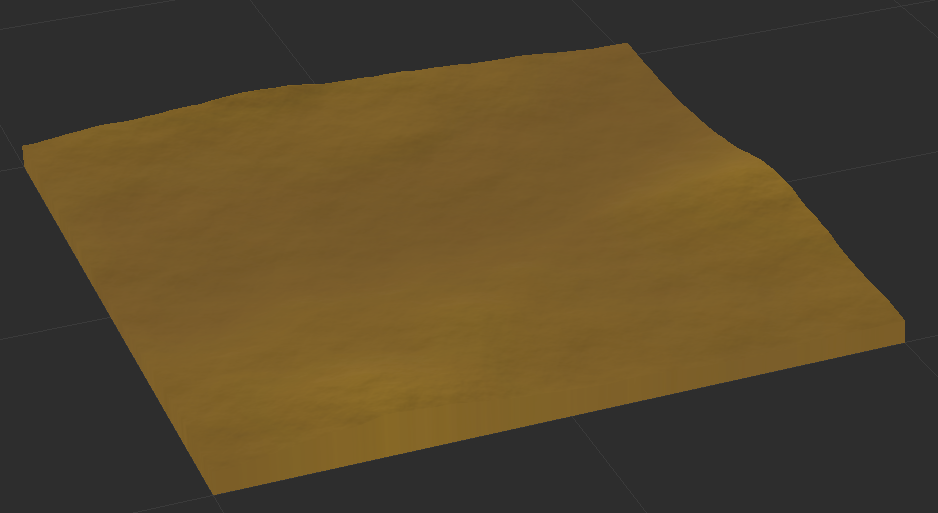


Рисунок 4 – Результат на выходе генератора Advanced Perlin

Результаты, полученные с помощью генераторов, были объединены по правилу Smooth Max. Затем была добавлена эрозия (рис. 5).

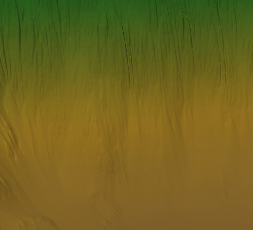
 

Рисунок 5 – Результат объединения до (слева) и после (справа) применения устройства Erosion

Итоговая карта высот была экспортирована в .png-файл с помощью узла HeightOutput.

3.2. Импорт ландшафта в Unreal Engine

Для импорта полученной карты высот в Unreal Engine необходимо перейти в режим работы с ландшафтами Landscape Mode (рис. 6). В меню Manage New нужно выбрать способ Import from File и указать путь к файлу карты высот (heightmap file). При этом в окне просмотра можно увидеть будущий результат импорта (рис. 7). Для завершения процесса нужно нажать кнопку Import.

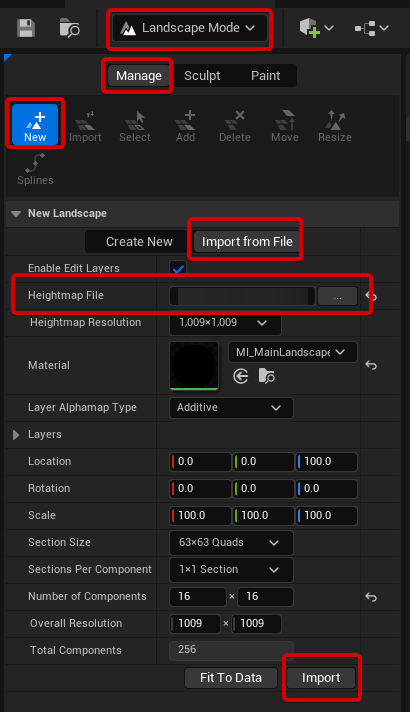


Рисунок 6 – Импорт ландшафта в Unreal Engine

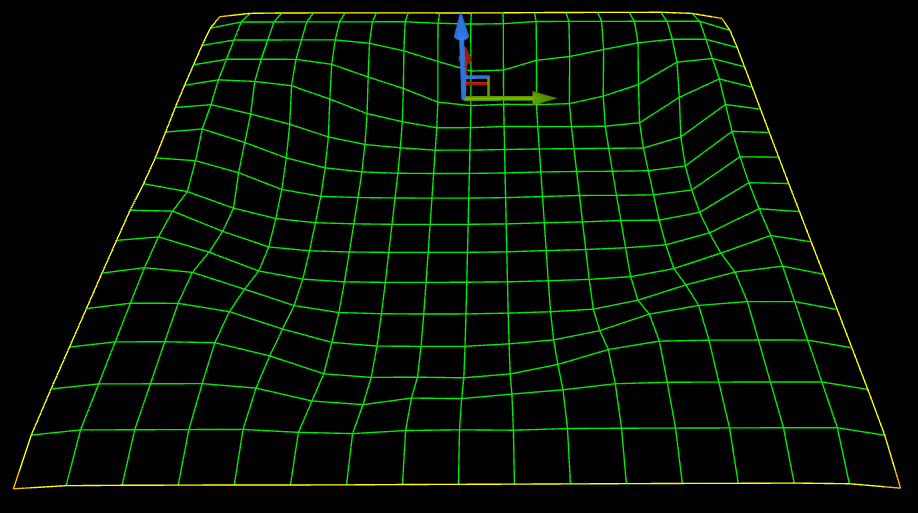


Рисунок 7 – Предпросмотр ландшафта в Unreal Engine

Результат импорта ландшафта с материалом по умолчанию изображен на рисунке 8.

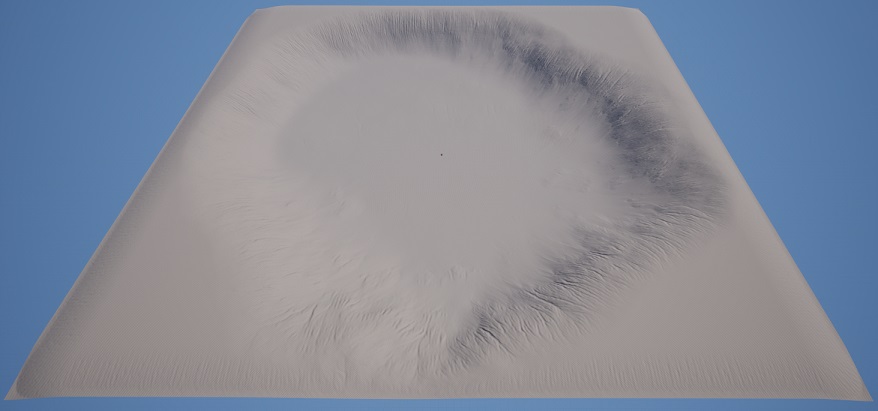


Рисунок 8 – Импортированный ландшафт с материалом по умолчанию в Unreal Engine

Чтобы игрок не мог случайно или намеренно выбраться за пределы карты, по периметру углубления была добавлена невидимая стена. Результат с итоговым материалом изображен на рисунке 9.

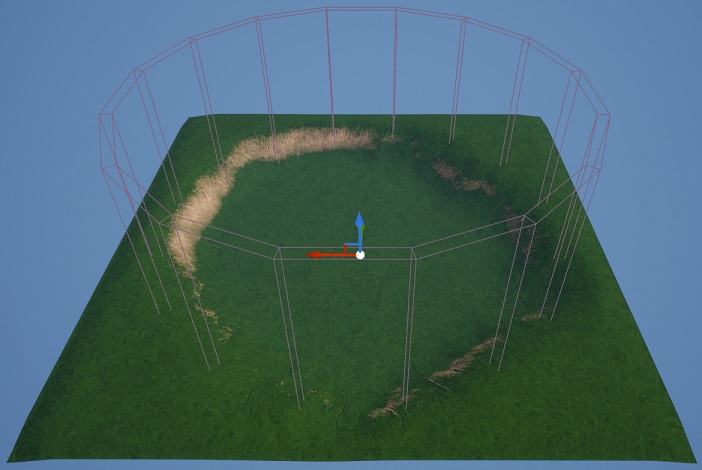


Рисунок 9 – Импортированный ландшафт с итоговым материалом и невидимой стеной в Unreal Engine

3.3. Использование системы PCG в Unreal Engine 5 для наполнения сцены растительностью

PCG (Procedural Content Generation Framework) – набор инструментов для процедурной генерации [5] в UE5, впервые появившийся в версии 5.2.

PCG Framework [6-8] добавляет новый вид ассетов – PCG Graph. В нем можно работать с пространственными данными в виде dataflow диаграмм, по аналогии с Blueprint’ами, материалами и т.д.

Основные типы данных в PCG:

* Spatial Data (пространственные данные) – ссылки на данные 2d- или 3d-пространства. Например: точки, поверхности, сплайны, объемы.
* Composite Data (сложные данные) – результат операций над множествами, таких как объединение, пересечение, вычитание.
* Attribute Sets (наборы атрибутов) – переменные, заданные пользователем. Хранятся в виде метаданных PCG Graph.

При наполнении сцены растительностью на уровне было выделено три зоны:

* «Безопасная зона» – место, зарезервированное для жилища персонажа. Из растительности в ней встречается трава и маленькие ростки. Деревья отсутствуют. Пример представлен на рисунке 10.
* Основная часть леса – место для охоты и рубки деревьев. В нем находится трава, средние ростки, средние по высоте деревья. Пример представлен на рисунке 11.
* Крайняя часть леса – зона, граничащая с невидимой стеной. В ней преобладают большие деревья, растущие близко друг к другу. Пример представлен на рисунке 12.



Рисунок 10 – Растительность в «безопасной зоне»



Рисунок 11 – Растительность в основной части леса



Рисунок 12 – Растительность в крайней части леса

Для разделения зон используются сплайны (рисунок 13).



Рисунок 13 – Сплайны, разделяющие зоны растительности

Чтобы в PCG Graph можно было найти сплайн и получить его пространственные данные, необходимо для актора, владеющего сплайном, создать уникальный тэг (рис. 14).

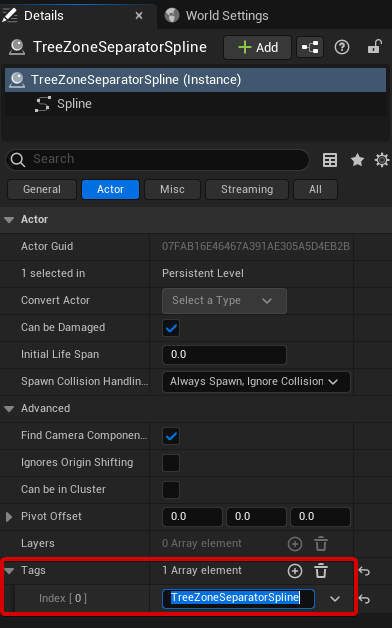


Рисунок 14 – Тэг у актора, владеющего сплайном

Затем с помощью узла Get Spline Data можно будет выбрать необходимый сплайн. Для этого в настройках поиска нужно выставить фильтр для рассмотрения всех акторов на карте (All World Actors), выбрать метод поиска «по тэгу» (By Tag) и ввести ранее созданный тэг (рис. 15).

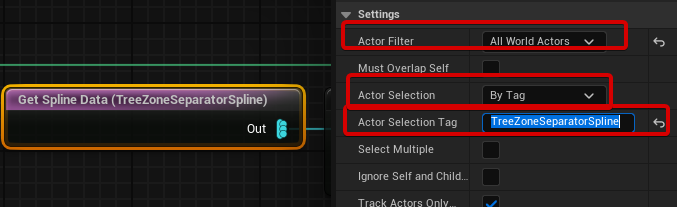


Рисунок 15 – Тэг у actor’а, владеющего сплайном

В данной работе используется два PCG Graph’а:

* PCG\_Grass отвечает за расположение травы и ростков.
* PCG\_Trees отвечает за расположение деревьев

Граф PCG\_Grass представлен на рисунке 16.

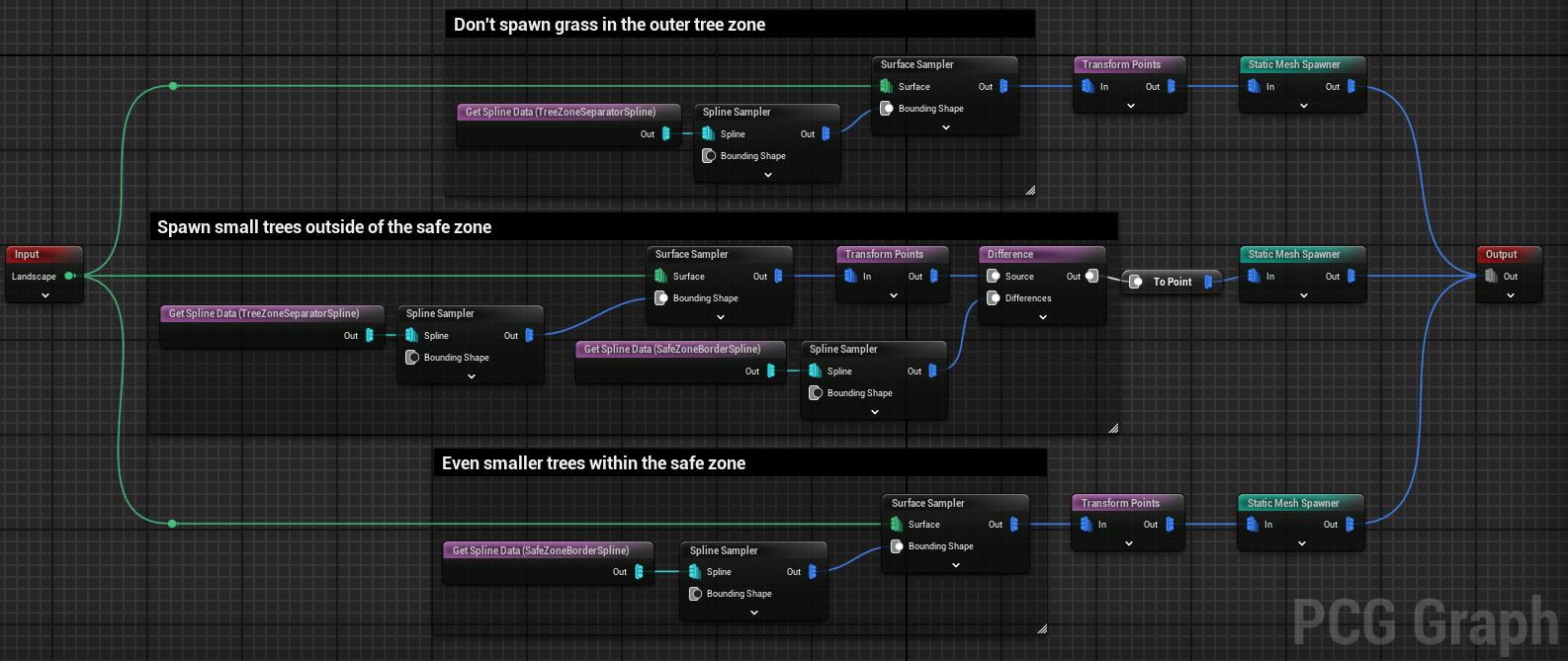


Рисунок 16 – Граф PCG\_Grass

В верхней ветви выполняется генерация травы. Surface Sampler используется для получения точек с меша ландшафта, передаваемый Spline Sampler накладывает ограничение, запрещающее помещать траву в крайней части леса (вне контура сплайна TreeZoneSeparator).

В средней ветви происходит генерация ростков среднего размера в основной части леса. Здесь, помимо ограничения сплайном извне, также вычитается часть точек, заключенных в сплайн SafeZoneSeparator («безопасная зона»).

В нижней ветви выполняется создание маленьких ростков внутри «безопасной зоны».

Граф PCG\_Trees представлен на рисунке 17.

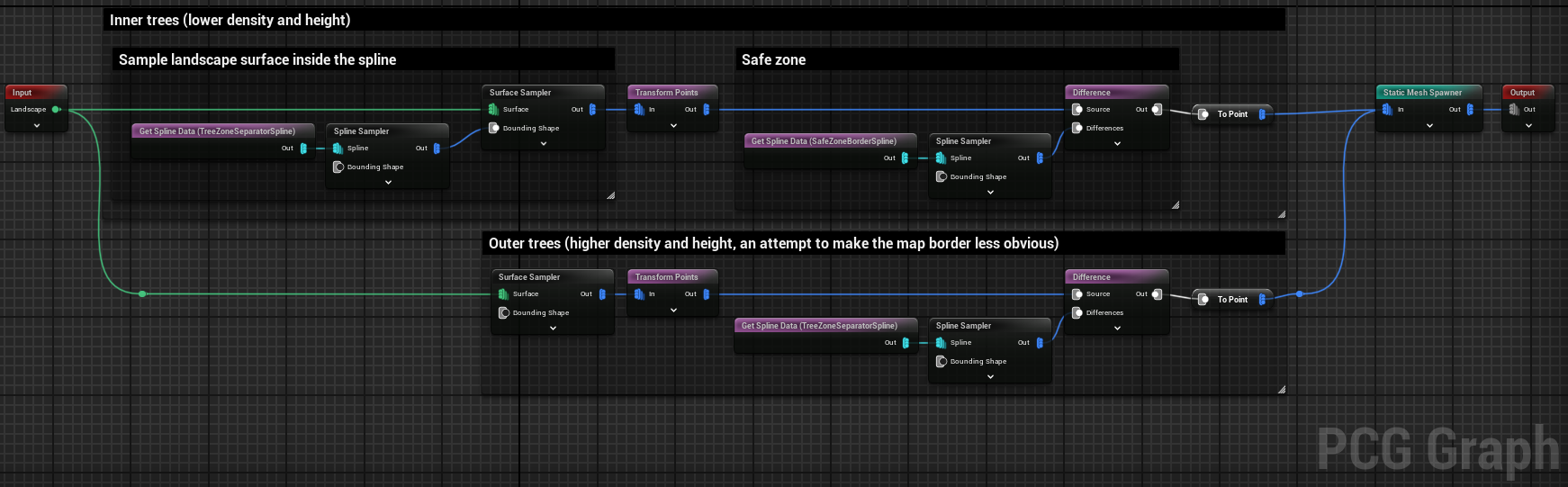


Рисунок 17 – Граф PCG\_Trees

В верхней ветви выполняется генерация маленьких деревьев в основной части леса, «безопасная зона» вновь вычитается.

В нижней ветви создается плотный массив деревьев, расположенных около края карты, в крайней части леса.

3.4. Место отдыха и приготовления пищи

В «безопасной зоне» расположены костер и палатка (рис. 18).



Рисунок 18 – Костер и палатка

Ключевым объектом является костер. Он используется для:

* приготовления пищи;
* поддержания температурного режима в безопасной зоне.

Костер реализован в C++-классе AFirepit, является наследником AActor. Он содержит следующие компоненты.

* USceneComponent – корневой компонент, задающий 3d-трансформацию актора.
* UBoxComponent – коллайдер, используемый при тестировании функциональности костра вручную. Позволяет реагировать на взаимодействие со стороны персонажа.
* UStaticMeshComponent – основной меш костра (камни). К нему впоследствии присоединяются остальные элементы костра (дрова, эффекты огня, и т.д.)
* Два UParticleSystemComponent – компоненты, владеющие эффектами огня и дыма.
* Два UTextRenderComponent – компоненты, позволяющие отображать текст над костром. Один из них показывает текущее количество древесины в костре, он видим постоянно. Другой из них становится видимым в процессе взаимодействия персонажа с костром, отображает название действия (рис. 19).



Рисунок 19 – Название и индикатор прогресса взаимодействия с костром

* UWidgetComponent – компонент, позволяющий использовать UMG-виджеты [9] на 3d-уровне. В данном случае с его помощью отображается прогресс текущего взаимодействия персонажа (рис. 18).
* UAttributeComponent – компонент, управляющий значениями индикатора прогресса.

Костер обладает следующими параметрами:

* текущее количество древесины;
* максимальное количество древесины;
* скорость сжигания древесины (количество в секунду);
* количество обжариваемого мяса за взаимодействие;
* длительность процесса выгрузки древесины;
* длительность процесса обжарки мяса.

При взаимодействии с костром персонаж может выгрузить имеющуюся древесину в костер, тем самым увеличить ее текущее количество (не превышая максимальное значение).

Также при взаимодействии с костром персонаж может обжарить некоторое количество мяса.

Для быстрого изменения параметров создан blueprint-класс BP\_Firepit, наследующий AFirepit (рис. 20).

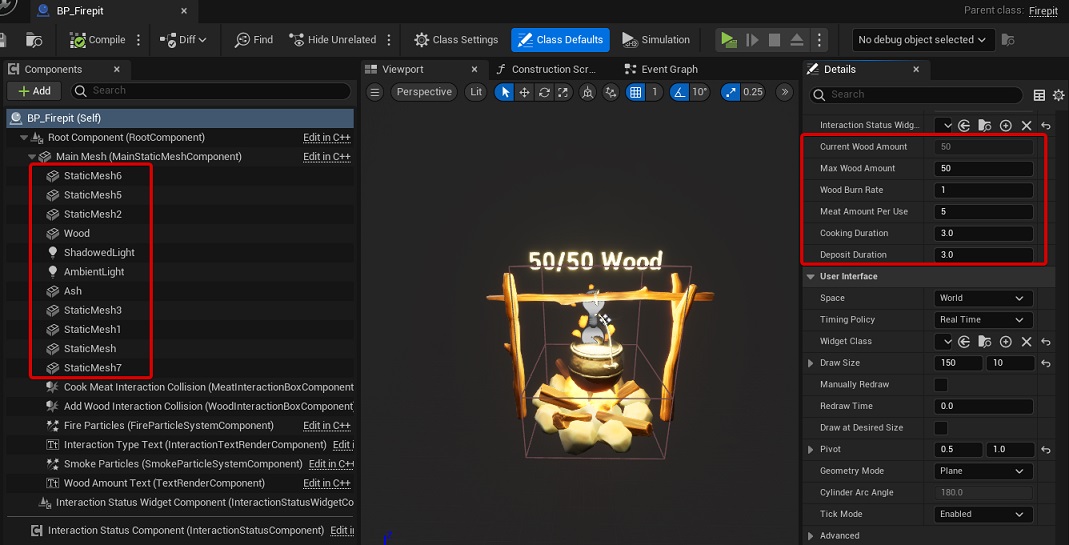


Рисунок 20 – Blueprint костра

В нем к основному мешу присоединяется несколько дополнительных компонентов: источники света и меши деталей костра. Источники света и эффекты огня и дыма становятся невидимыми, когда в костре кончается древесина.

3.5. Дикие звери (противники)

В основной части леса персонаж может охотиться на диких зверей (противников) (рис. 21).



Рисунок 21 – Дикий зверь (противник)

Реализация логики противников представлена в трех C++-классах.

* AEnemy (наследник ACharacter) – основной класс противника. Содержит все его компоненты и параметры.
* UEnemyAnimInstance (наследник UAnimInstance) – класс, являющийся основой анимационного blueprint’а противника. Хранит актуальные значения переменных состояния, используемых для определения переходов между анимациями.
* AEnemyAIController (наследник AAIController) – класс контроллера («мозга») противника.

*AEnemy* наследует все компоненты, необходимые для примитивной настройки персонажа, из класса ACharacter, и в дополнение к ним имеет:

* UAttributeComponent – компонент, отвечающий за здоровье противника;
* UWidgetComponent – компонент, владеющий виджетом, отображающим состояние здоровья противника.
* UPawnSensingComponent – компонент, позволяющий противнику обнаруживать иных «пешек» и персонажей, в частности, игрока [10].
* UBoxComponent – коллайдер, расположенный в области головы противника, позволяющий фиксировать попадания по персонажу во время анимации атаки. Коллизия этого компонента включается только в течение атаки.

*AEnemy* имеет следующие параметры:

* максимальное значение здоровья противника;
* урон противника.

От AEnemy унаследован blueprint-класс BP\_EnemyDeer, в котором обновлено расположение коллайдера, назначены актуальные классы анимационного blueprint’а и контроллера, а также задан меш оленя (рис. 22).

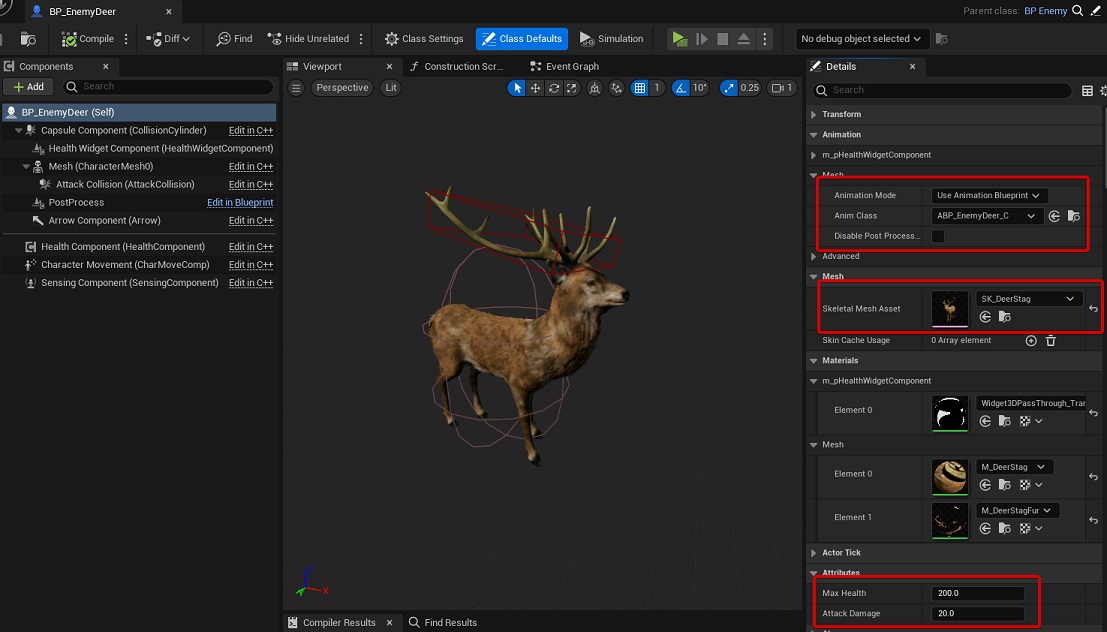


Рисунок 22 – Blueprint противника

Класс *UEnemyAnimInstance* каждый кадр получает значения переменных состояния противника, хранящиеся в AEnemy, например:

* атакует ли противник;
* жив ли противник;
* перемещается ли противник;
* и т.д.

Эти значения используются в анимационном blueprint’е ABP\_EnemyDeer, наследующем UEnemyAnimInstance. На рисунке 23 изображен граф анимаций противника.

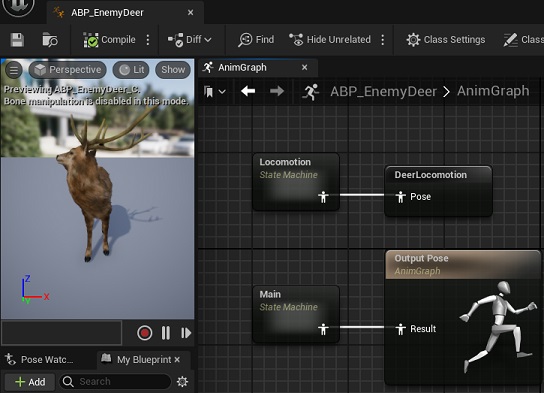


Рисунок 23 – Анимационный граф противника

В нем присутствует два анимационных конечных автомата (state machines). Один из них отвечает за перемещение противника по поверхности (locomotion). Его состояния представлены на рисунке 24.

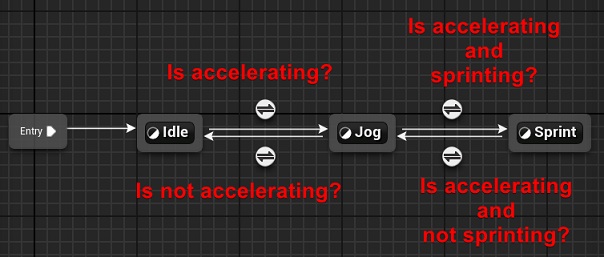


Рисунок 24 – Состояния конечного автомата Locomotion

В нем есть три состояния: Idle (ожидание), Jog (передвижение шагом), Sprint (передвижение бегом). В состоянии Idle циклично проигрывается анимация ожидания. В состоянии Jog используется blend space, состоящий из различных анимаций перемещения оленя (рис. 25).



Рисунок 25 – Блендспейс перемещения противника

Горизонтальная ось соответствует параметру «угол рыскания» (yaw), вертикальная – скорости передвижения. При минимальном значении скорости передвижения используются три анимации на своей базовой скорости воспроизведения: движение вперед (A\_DeerStag\_Walk), движение налево (A\_DeerStag\_WalkTurnL), движение направо (A\_DeerStag\_WalkTurnR). Для максимального значения скорости передвижения используются те же самые анимации, но с повышенной скоростью воспроизведения.

Изначально активируется состояние Idle. Затем, если скорость противника больше нуля, выполняется переход в состояние Jog. Если в UEnemyAnimInstance установлена переменная состояния IsSprinting (бежит ли противник), то выполняется переход в состояние Sprint.

Анимационный конечный автомат Main отвечает за переход между состоянием передвижения и атаки (рис. 26).



Рисунок 26 – Состояния конечного автомата Main

В состояние Walk передается поза DeerLocomotion (рис. 23), в которую сохранен конечный автомат Locomotion. В состоянии Attack циклично воспроизводится анимация атаки оленя. Переход между состояниями осуществляется в зависимости от значения переменной состояния IsAttacking (атакует ли противник).

Класс AEnemyAIController выполняет запуск дерева поведения противника. Также он взаимодействует с соответствующей доской.

* Когда UPawnSensingComponent противника обнаруживает игрока, в доску записываются переменные TargetPlayer с указателем на объект персонажа и TargetLocation с его местоположением. Если этот компонент не находит игрока, в TargetPlayer записывается nullptr, а переменная TargetLocation удаляется.
* Когда противник начинает атаку, в доску записывается булева переменная bAttackInProgress со значением true. При окончании атаки значение этой переменной устанавливается в false.

На рисунке 27 изображено дерево поведения противника. На рисунке 28 изображена используемая им доска (blackboard).

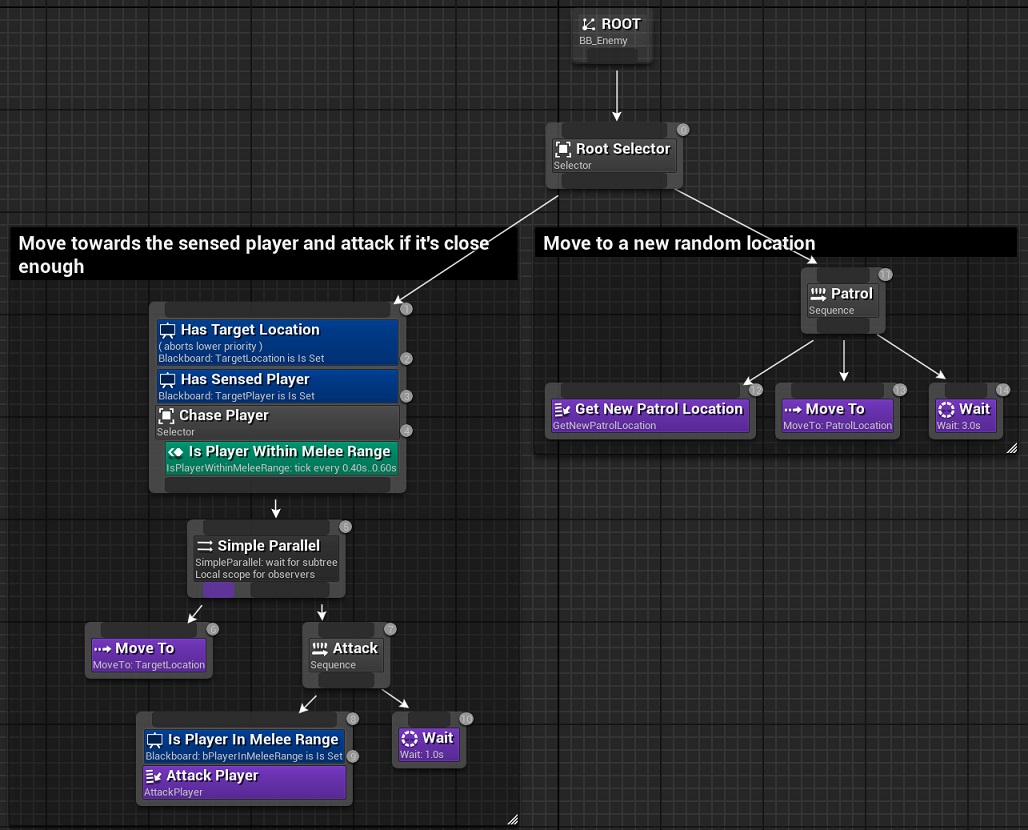


Рисунок 27 – Дерево поведения противника

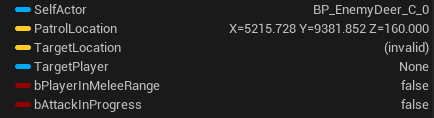


Рисунок 28 – Значения на доске противника

У противника есть два сценария поведения: патрулирование и погоня за персонажем игрока с попытками атаки.

Сценарий патрулирования запускается в том случае, если на доске не заданы переменные TargetPlayer и TargetLocation, т.е. игрок не обнаружен. В таком случае циклически повторяется последовательность из трех действий:

* выбор нового местоположения;
* перемещение в новую позицию;
* ожидание.

Если же игрок обнаружен (установлены переменные TargetPlayer и TargetLocation), противник начинает бежать за ним. Если расстояние между противником и персонажем меньше расстояния атаки, противник пытается нанести урон игроку.

Задача выбора нового местоположения (Get New Patrol Location на рисунке 27) реализована в C++-классе UBTTask\_GetNewPatrolLocation. Ее алгоритм имеет следующие шаги:

1. выбор случайной достижимой дочки в определенном радиусе вокруг противника;
2. запись полученного значения в переменную PatrolLocation (рис. 28) на доске.

Задача атаки игрока противником (Attack Player на рисунке 27) реализована в C++-классе UBTTask\_AttackPlayer. Ее алгоритм имеет следующие шаги:

1. вызов метода начала атаки StartAttack;
   * установка переменной состояния противника IsAttacking в true;
   * установка переменной bAttackInProgress на доске в true;
   * включение коллайдера противника;
2. воспроизведение анимации атаки до срабатывания анимационного оповещения StopAttacking (рис. 29);
3. вызов метода завершения атаки (рис. 30);
   * установка переменной состояния противника IsAttacking в false;
   * установка переменной bAttackInProgress на доске в false;
   * выключение коллайдера противника;
4. завершение задачи.

Задачи передвижения в новое местоположение (Move To) и ожидания (Wait) доступны для использования по умолчанию.

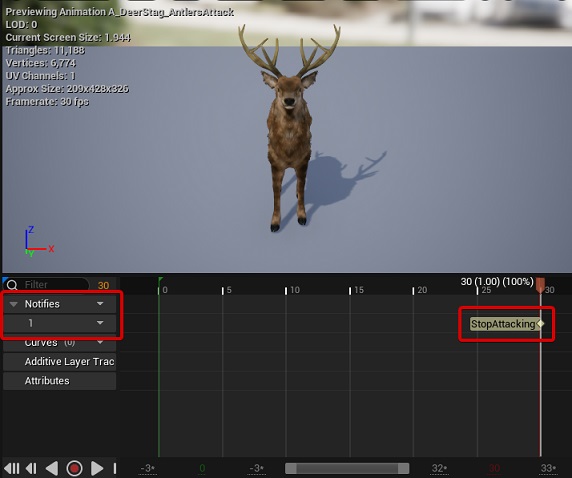


Рисунок 29 – Анимационное оповещение (AnimNotify) о завершении анимации атаки

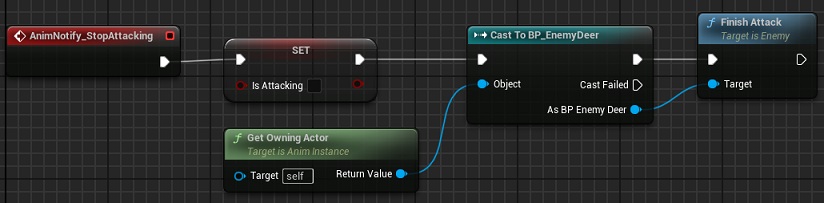


Рисунок 30 – Вызов метода завершения атаки при срабатывании события AnimNotify

3.6. Создание диких зверей на уровне

Для управления созданием диких зверей во время работы игры был разработан C++-класс актора AEnemySpawner (спаунер). Его задача – создавать противников в окрестности заданного местоположения определенное количество раз в секунду. Максимальное количество противников в момент времени может быть ограничено.

Для удобства настройки параметров был создан blueprint-класс BP\_EnemySpawner, наследующий AEnemySpawner.

На рисунке 31 изображено три спаунера противников. Во время работы в редакторе Unreal Engine с помощью красных окружностей отображаются границы окрестности, в которой могут быть созданы противники.



Рисунок 31 – Спаунеры противников

Спаунер состоит из следующих компонентов:

* USceneComponent – корневой компонент, задающий 3d-трансформацию актора.
* UGizmoCircleComponent – компонент, создающий параметризованную окружность на 3d-сцене. Используется для отображения радиуса спаунера во время работы в редакторе.

Параметры спаунера:

* EnemySubclass – класс актора противника (должен быть дочерним от AEnemy);
* Spawn Range – радиус области создания противников;
* Spawn Rate – частота создания противников (единиц в секунду);
* Max Enemy Count – максимальное единовременное количество противников, созданных спаунером.

Спаунер противников работает по следующим принципам.

* При вступлении в игру (вызове функции BeginPlay) инициализируется таймер, вызывающий функцию создания противника SpawnEnemy раз в заданный временной интервал (значение, обратное Spawn Rate).
* Функция SpawnEnemy выполняет следующие действия:
  1. выбор случайной достижимой позиции в радиусе спаунера;
  2. выбор случайного угла рыскания (yaw);
  3. задание параметров создания противника в структуре FActorSpawnParameters, установка метода разрешения коллизий;
  4. создание противника с помощью функции SpawnActor;
  5. установка местоположения и поворота противника с помощью функций SetActorLocation и SetActorRotation;
  6. обновление внутреннего массива, содержащего информацию о существующих противниках, созданных спаунером.
* При завершении игровой сессии (вызове функции EndPlay) таймер удаляется.
* При изменении частоты создания противников текущий таймер удаляется, вместо него создается новый с обновленным интервалом срабатывания.

4. Создание персонажа

4.1 Атрибуты персонажа

Для работы с атрибутами персонажа разработан C++-класс *UAttributeComponent*, наследующий UActorComponent. Он может быть использован для хранения информации и взаимодействия с любым динамическим параметром, имеющим ограниченный диапазон значений.

Возможности *UAttributeComponent*:

* задание диапазона допустимых значений атрибута;
* автоматическое ограничение текущего значения атрибута в рамках заданного диапазона;
* несколько делегатов – можно назначить функции обратного вызова (callbacks) на случаи достижения минимального и максимального значений, на каждое обновление значения, на каждый тик;
* поддержка Blueprint.

Также был разработан С++-класс *UAttributeWidget*, лежащий в основе UMG-виджета WBP\_AttributeWidget. Он является наследником класса пользовательских виджетов UUserWidget.

*UAttributeWidget* состоит из нескольких элементов.

* UProgressBar (индикатор процесса) используется для визуализации текущего значения атрибута.
* Три UTextBlock’а (блока текста) отображают текущее и максимальное значения атрибута, а также разделительную черту между ними. Могут быть отключены.
* UImage (изображение) используется для отображения иконки атрибута. Может быть отключено.

К UAttributeComponent можно привязать UAttributeWidget, и он будет автоматически обновляться при изменении значений атрибута.

Комбинация из UAttributeComponent и UAttributeWidget используется для отображения состояний атрибутов главного персонажа в HUD, для индикации текущего уровня здоровья у диких зверей, а также для отображения прогресса при выполнении продолжительного действия персонажем. Примеры приведены на рисунках 21 и 32.

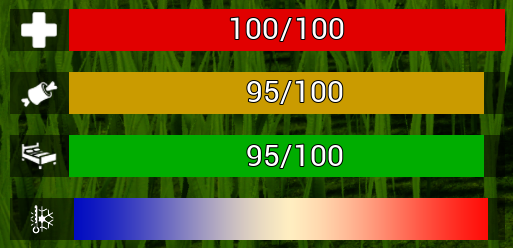


Рисунок 32 – UMG-виджеты атрибутов персонажа

В HUD персонажа отображаются следующие атрибуты (рис. 32, сверху вниз):

* уровень здоровья;
* уровень сытости (голода);
* уровень бодрости (усталости);
* состояние температурного режима (слева – переохлаждение, справа – перегрев).

К компонентам атрибутов персонажа привязано несколько функций обратного вызова:

* при нулевом уровне здоровья и сытости вызывается функция смерти персонажа;
* при нулевом значении бодрости вызывается функция, начинающая процесс сна персонажа;
* при экстремальных состояниях температурного режима также вызывается функция смерти персонажа.

4.2 Инвентарь персонажа

Для работы с инвентарем персонажа разработан C++-класс UInventoryComponent, являющийся дочерним классом от UActorComponent.

В основе инвентаря игрока лежит ассоциативный контейнер – TMap<FString, int32>.

*UInventoryComponent* имеет следующие возможности:

* добавление предметов по ключу;
* удаление предметов по ключу;
* получение количества предмета по ключу;
* получение статуса наличия предмета по ключу;
* проверка на пустоту инвентаря.

К UInventoryComponent привязан UMG-виджет *USurvivalInventoryWidget*, который автоматически обновляется при изменении состояния инвентаря (рис. 33).



Рисунок 33 – Виджет инвентаря игрока

В виджете инвентаря персонажа отображается количество следующих предметов (рис. 33, сверху вниз):

* древесина;
* сырое мясо;
* приготовленное (жареное) мясо.

4.3 Класс персонажа

Внешний вид персонажа, управляемого игроком или большой языковой моделью, представлен на рисунке 34.



Рисунок 34 – Внешний вид персонажа

Реализация логики персонажа представлена в классах:

* ASurvivalCharacter – основной класс персонажа. Содержит все его компоненты и параметры.
* USurvivalCharacterAnimInstance – класс, являющийся основой анимационного blueprint’а персонажа. Хранит актуальные значения переменных состояния, используемых для определения переходов между анимациями.
* ASurvivalPlayerController – класс контроллера персонажа, позволяющий управлять им вручную путем ввода с клавиатуры и мыши. Если контроль делегирован ИИ-контроллеру, данный класс позволяет наблюдать за действиями персонажа.
* ASurvivalPlayerAIController – класс ИИ-контроллера персонажа.

*ASurvivalCharacter* наследует все компоненты, необходимые для примитивной настройки персонажа, из класса ACharacter, и в дополнение к ним имеет:

* Четыре UAttributeComponent’а для атрибутов персонажа – каждый из них представляет один из них: здоровье, сытость, усталость, температурный режим;
* UInventoryComponent – компонент, позволяющий управлять инвентарем персонажа;
* UCameraComponent – компонент камеры, следующей за персонажем;
* USpringArmComponent – компонент, позиционирующий камеру сзади персонажа и приближающий ее к персонажу при коллизии с другими объектами сцены. «Сдерживает» камеру от попадания внутрь мешей;
* Дополнительный UAttributeComponent для отслеживания прогресса поглощения пищи;
* UWidgetComponent для виджета, отображающего прогресс поглощения пищи;
* ULLMComponent – компонент, позволяющий делать запросы к большой языковой модели из Unreal Engine.
* Два USceneComponent’а – один из них задает местоположение топора, используемого при рубке дерева, другой определяет позицию, в которой создаются стрелы во время атаки противников.

*ASurvivalCharacter* имеет следующие параметры:

* параметры передвижения:
  + скорость движения шагом;
  + скорость движения бегом;
* параметры атрибутов:
  + максимальное значение здоровья;
  + максимальное значение сытости;
  + максимальное значение бодрости/усталости;
  + минимальное и максимальное допустимые значения температуры;
  + затраты бодрости и сытости в секунду;
* параметры атаки:
  + расстояние атаки;
  + скорость атаки;
  + урон от попадания;
* скорость рубки дерева;
* параметры приема пищи:
  + количество восстанавливаемых единиц сытости за единицу мяса;
  + количество съедаемых единиц мяса за прием пищи;
  + продолжительность приема пищи;
* продолжительность сна.

От ASurvivalCharacter унаследован blueprint-класс *BP\_SurvivalCharacter*, в котором назначены актуальные классы анимационного blueprint’а и контроллеров, заданы меши персонажа и топора, переопределены некоторые параметры (рис. 35).

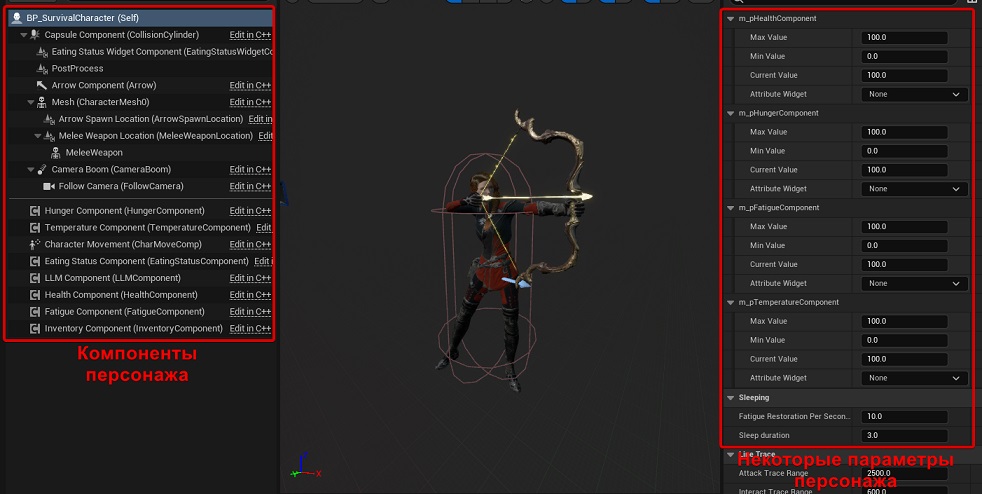


Рисунок 35 – Blueprint персонажа

4.4 Настройка анимаций персонажа

Класс *USurvivalCharacterAnimInstance* каждый кадр получает значения переменных состояния персонажа, хранящиеся в ASurvivalCharacter, например:

* IsDead – мертв ли персонаж;
* IsAccelerating – перемещается ли персонаж;
* IsSprinting – бежит ли персонаж;
* IsInAir – находится ли персонаж в воздухе (в прыжке);
* IsSleeping– спит ли персонаж;
* IsWakingUp – пробуждается ли персонаж;
* и т.д.

Эти значения используются в анимационном blueprint’е *ABP\_SurvivalCharacter*, наследующем USurvivalCharacterAnimInstance. На рисунке 36 изображен граф анимаций персонажа.

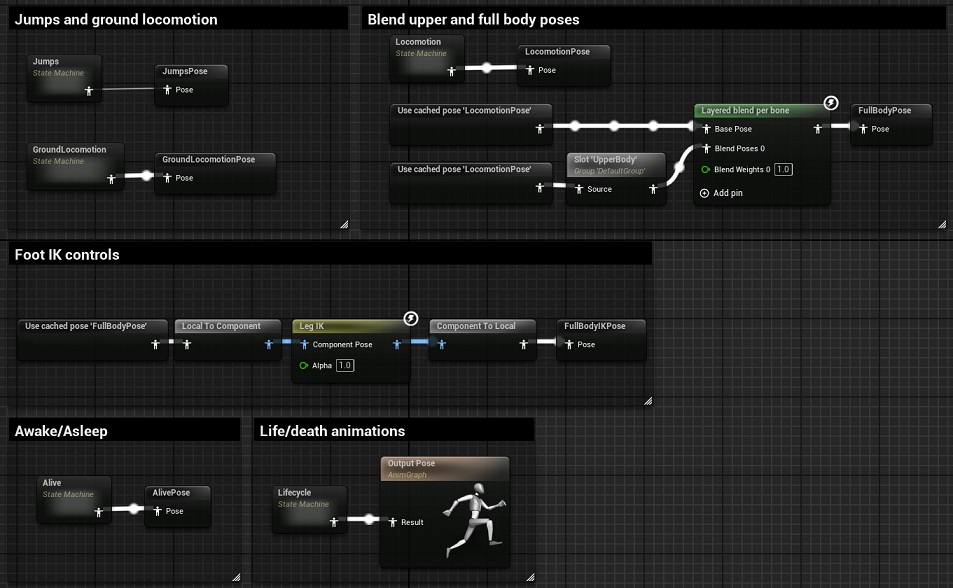


Рисунок 36 – Анимационный blueprint персонажа

В нем присутствует пять анимационных конечных автоматов (state machines). Один из них отвечает за анимацию прыжков, включает в себя три состояния с автоматическими переходами: начало прыжка (JumpStart), нахождение в воздухе (JumpApex) и начало приземления (JumpPreLand) (рис. 37).

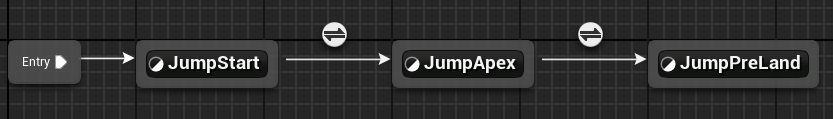


Рисунок 37 – Конечный автомат прыжков персонажа (Jumps)

Конечный автомат Locomotion задает переходы между анимациями перемещения по земле (OnGround), прыжками (Jump) и приземлением после прыжка (Land) (рис. 38).

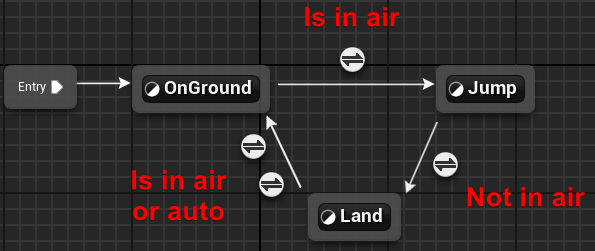


Рисунок 38 – Конечный автомат перемещения персонажа (Locomotion)

В анимационную позу состояния перемещения по земле (OnGround) передается результат интерполяции между отдельными анимациями движения персонажа в зависимости от скорости и направления движения. Их значения считываются из полей класса USurvivalCharacterAnimInstance (рис. 39).

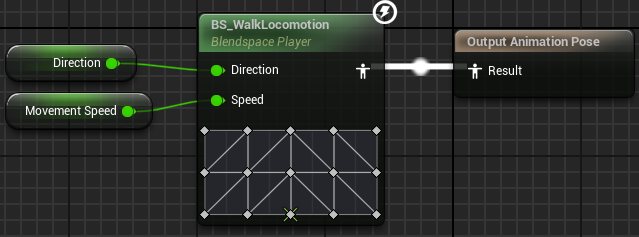


Рисунок 39 – Конечный автомат перемещения персонажа (GroundLocomotion)

Блендспейс персонажа (рис. 40) состоит из пятнадцати анимаций, соответствующих трем ключевым значениям скорости передвижения и пяти значениям угла поворота (рис. 41). Такой подход позволяет создать автоматические переходы между анимациими ожидания, ходьбы и бега.

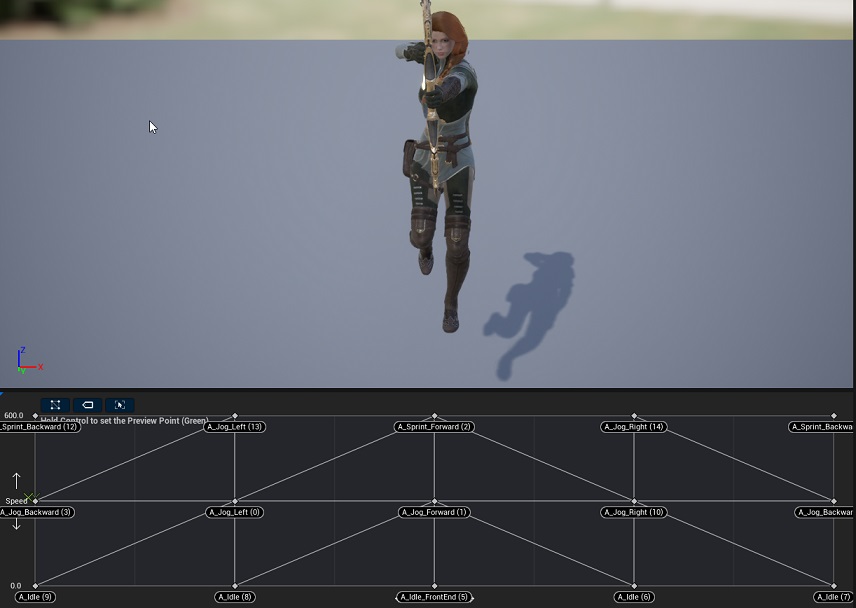


Рисунок 40 – Блендспейс перемещения персонажа

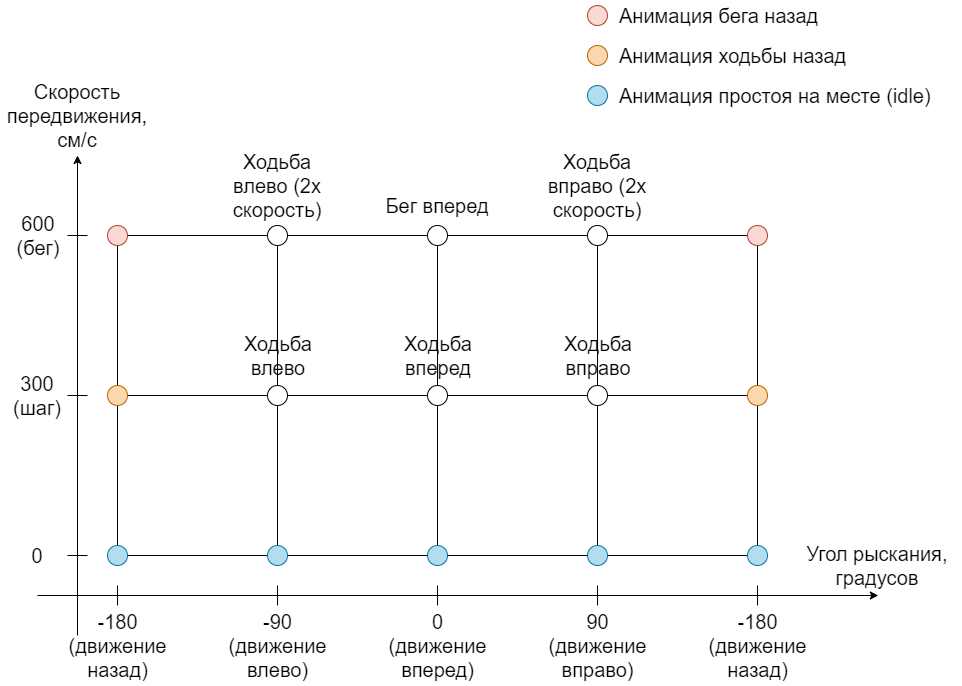


Рисунок 41 – Пояснение к блендспейсу персонажа

Анимационная поза состояния Jump является результатом работы конечного автомата Jumps, представленного на рисунке 36.

Поза состояния Land представляет собой сумму позы на выходе конечного автомата GroundLocomotion и анимации приземления (рис. 42).

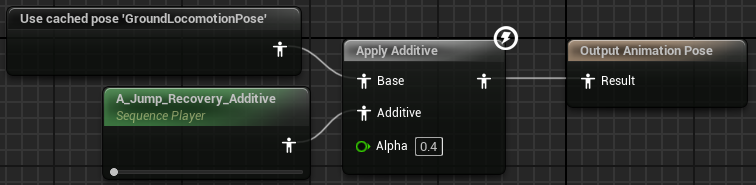


Рисунок 42 – Конечный автомат перемещения персонажа (Locomotion)

За анимацию жизненного цикла персонажа отвечает конечный автомат Lifecycle (рис. 43). При появлении персонажа на уровне выполняется автоматический переход в состояние Spawn. При любом действии персонажа либо по завершении анимации появления выполняется переход в состояние Alive, при смерти персонажа выполняется переход в состояние Dead.

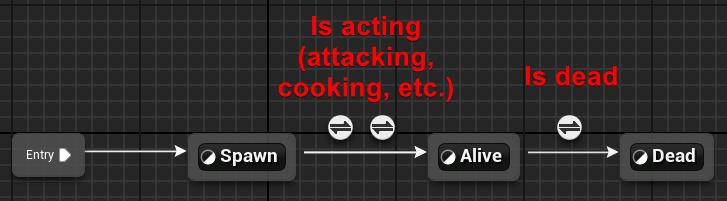


Рисунок 43 – Конечный автомат жизненного цикла персонажа (Lifecycle)

Состояние Alive «покрывает» все возможные состояния, которые могут возникнуть в течение жизни персонажа. В него передается поза из конечного автомата Alive, управляющего анимационными переходами при сне и бодрствовании персонажа (рис. 44).

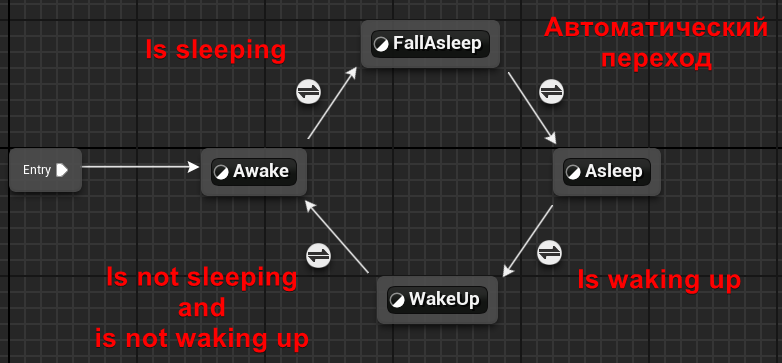


Рисунок 44 – Конечный автомат, используемый во время жизни персонажа (Alive)

В состояние Awake передается поза персонажа, содержащая все необходимые для движения и выполнения действий состояния (FullBodyIKPose на рис. 36).

При установке переменной IsSleeping в true все действия прерываются, выполняется переход в состояние FallAsleep, персонаж ложится на землю. По завершении этой анимации активируется состояние Asleep, циклично воспроизводится анимация сна. При установке переменной IsWakingUp в true персонаж начинает пробуждаться, вставая с земли (WakeUp), при этом переменная IsSleeping устанавливается в false.

В конце анимации пробуждения срабатывает анимационное оповещение FinishWakingUp (рис. 45). На него назначен вызов функции завершения сна (рис. 46), в процессе выполнения которой переменная IsWakingUp устанавливается в false. Если персонаж не спит и не пробуждается, в конечном автомате Alive выполняется переход из состояния WakeUp в Awake.



Рисунок 45 – Анимационное оповещение (AnimNotify) о завершении анимации пробуждения

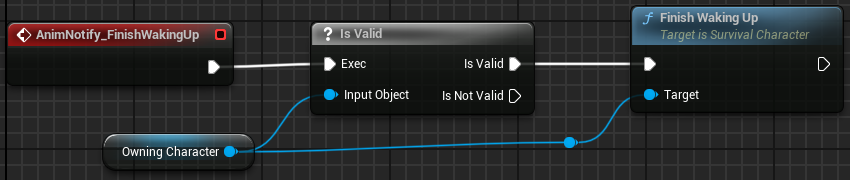


Рисунок 46 – Вызов метода завершения пробуждения при срабатывании события AnimNotify

4.5 Действия персонажа

Персонаж способен выполнять следующие действия:

* охотиться на диких зверей, получать за это сырое мясо;
* рубить деревья, получать за это древесину;
* готовить пищу на костре, «обменивая» сырое мясо на жареное;
* добавлять древесину в костер;
* поглощать пищу;
* спать.

Класс ASurvivalAIController выполняет запуск основного дерева поведения персонажа BT\_SurvivalCharacter.

Для отдельных действий персонажа созданы свои поддеревья поведения, например, BT\_Hunt для охоты на диких зверей, BT\_Cook для приготовления пищи и т.д.

Как основное дерево, так и все поддеревья используют одну и ту же доску (blackboard) для записи и чтения значений переменных – BB\_SurvivalCharacter (рис. 47).

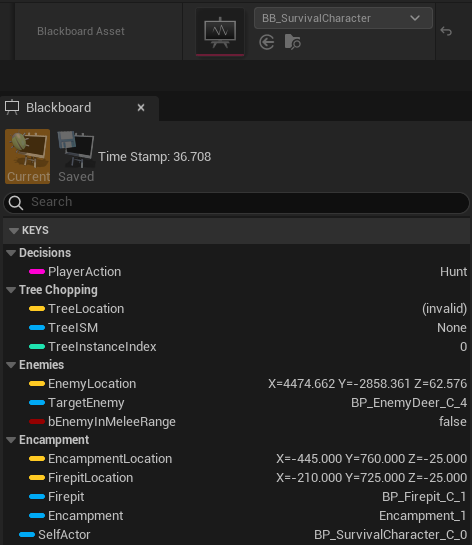


Рисунок 47 – Пример значений на доске персонажа

4.5.1 Охота на диких зверей

На рисунке 48 изображен персонаж в процессе охоты на диких зверей.



Рисунок 48 – Персонаж в процессе атаки противника

На рисунке 49 изображено дерево поведения BT\_Hunt, в котором реализована логика охоты.

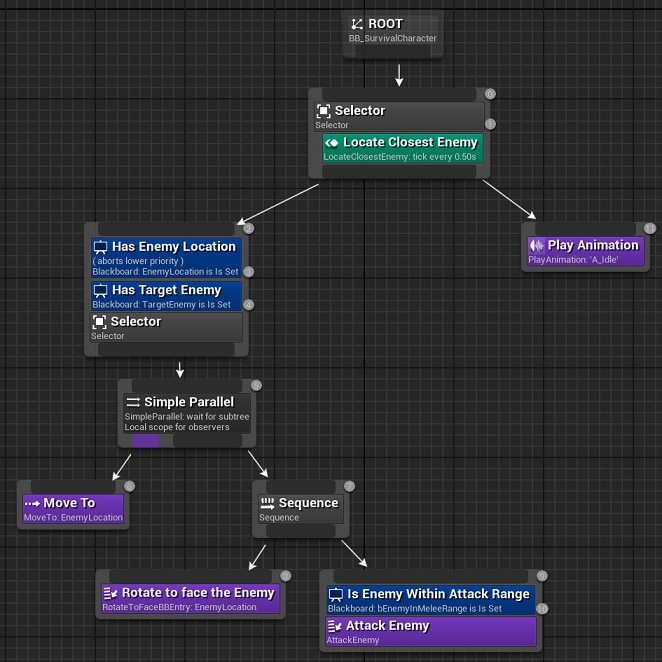


Рисунок 49 – Дерево поведения охоты персонажа на диких зверей

В дереве поведения охоты есть две основные ветви: если вокруг нет противников, воспроизводится анимация ожидания, если противник найден – персонаж начинает двигаться в его направлении и атаковать его при нахождении зверя в радиусе атаки.

Поиск ближайшего противника осуществляется с помощью сервиса, реализованного в C++-классе BTService\_LocateClosestEnemy. Он осуществляет поиск по всем существующим зверям, созданным с помощью спаунеров, и сохраняет местоположение (EnemyLocation) и указатель на объект (TargetEnemy) ближайшего из них в доску.

Задача атаки противника (зверя) игроком (Attack Enemy на рисунке 49) реализована в C++-классе UBTTask\_AttackEnemy. Ее алгоритм имеет следующие шаги:

1. выполняется вызов метода начала атаки StartAttacking;
   * переход персонажа с бега на шаг;
   * установка переменной состояния персонажа IsAttacking в true;
   * запуск таймера атаки, частота срабатывания которого определяется параметром «скорость атаки»;
2. каждое срабатывание таймера:
   * проигрывается анимация атаки из лука;
   * создается и инициализируется актор снаряда (стрелы) класса AProjectile, наносящий урон противнику при попадании;
3. выполняется вызов метода завершения атаки StopAttacking:
   * установка переменной состояния персонажа IsAttacking в false;
   * остановка воспроизведения анимации атаки из лука;
   * сброс таймера атаки;
   * переход персонажа с шага на бег;
4. завершение задачи.

Задачи передвижения в новое местоположение (Move To) и поворота в направлении противника (RotateToFaceBBEntry) доступны для использования по умолчанию.

4.5.2 Рубка дерева

На рисунке 50 изображен персонаж в процессе рубки дерева.



Рисунок 50 – Персонаж в процессе рубки дерева

На рисунке 51 изображено дерево поведения BT\_ChopTree, в котором реализована логика рубки леса.

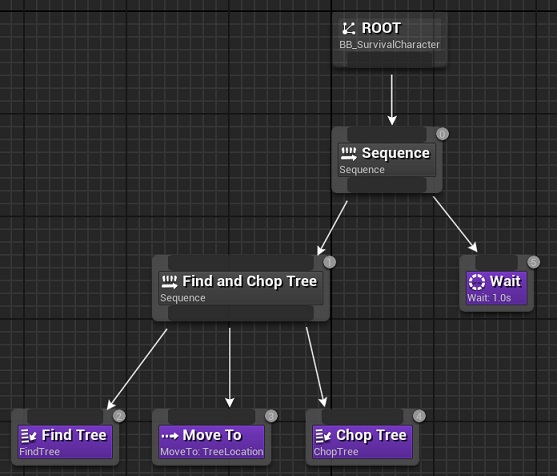


Рисунок 51 – Дерево поведения рубки леса

Алгоритм рубки дерева имеет следующие шаги:

1. поиск дерева;
2. движение персонажа к дереву;
3. рубка дерева.

Все деревья на уровне создаются с помощью системы процедурной генерации PCG, являются инстансами одного из ISMC. В рамках данного проекта в PCG графах используются компоненты инстансинга:

* для деревьев – UFoliageInstancedStaticMeshComponent;
* для травы – UGrassInstancedStaticMeshComponent.

Для того, чтобы срубить дерево, персонаж должен нанести по нему определенное число ударов. Значение «здоровья» каждого дерева хранится в PerInstanceCustomData.

Задача поиска ближайшего дерева (Find Tree на рисунке 49) реализована в C++-классе UBTTask\_FindTree. Ее алгоритм заключается в проходе по всем инстансам всех UFoliageInstancedStaticMeshComponent’ов и сохранении номера инстанса (TreeInstanceIndex), позиции дерева (TreeLocation) и соответствующего ISMC (TreeISMC) в доску.

Задача поиска ближайшего дерева (Chop Tree на рисунке 49) реализована в C++-классе UBTTask\_ChopTree. Ее алгоритм имеет следующие шаги:

1. вызов метода начала рубки дерева StartChopping;
   * переход персонажа с бега на шаг;
   * скрытие 3d-модели лука, отображение 3d-модели топора;
   * установка переменной состояния персонажа IsChopping в true;
   * запуск таймера рубки, частота срабатывания которого определяется параметром «скорость рубки дерева»;
2. каждое срабатывание таймера:
   * проигрывается анимация атаки топором;
   * с каждым ударом значение «здоровья» дерева уменьшается на единицу;
   * если значение «здоровья» становится равно нулю, инстанс этого дерева удаляется, вызывается функция StopChopping;
   * персонажу выдается определенное количество древесины;
3. вызов функции StopChopping:
   * установка переменной состояния персонажа IsChopping в false;
   * сброс переменных доски TreeInstanceIndex, TreeLocation и TreeISMC;
   * остановка воспроизведения анимации рубки дерева топором;
   * скрытие 3d-модели топора, отображение 3d-модели лука
   * сброс таймера атаки;
   * переход персонажа с шага на бег;
4. завершение задачи.

4.5.3 Приготовление пищи

На рисунке 52 изображен персонаж в процессе приготовления пищи на костре.



Рисунок 52 – Персонаж в процессе приготовления пищи на костре

На рисунке 53 изображено дерево поведения BT\_Cook, в котором реализована логика приготовления пищи.

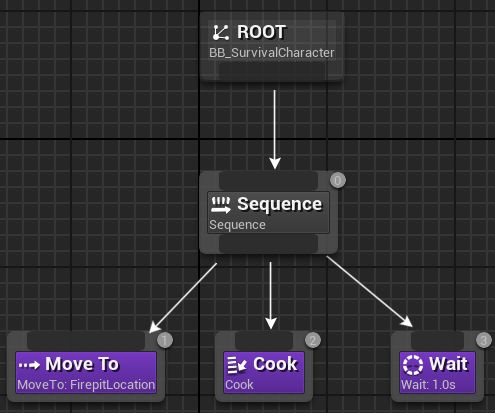


Рисунок 53 – Дерево поведения приготовления пищи

Алгоритм приготовления пищи имеет следующие шаги:

1. движение персонажа к костру;
2. взаимодействие с костром для приготовления пищи.

Задача приготовления пищи (Cook на рисунке 53) реализована в C++-классе UBTTask\_Cook. Ее алгоритм имеет следующие шаги:

1. вызов метода начала рубки дерева StartCooking;
   * остановка всех текущих действий;
   * установка переменной состояния персонажа IsCooking в true;
   * запуск таймера приготовления пищи;
2. каждое срабатывание таймера:
   * обновляется прогресс приготовления;
   * если процесс приготовления завершен, определенное количество сырого мяса заменяется на жареное, вызывается функция StopCooking;
3. вызов функции StopCooking:
   * установка переменной состояния персонажа IsCooking в false;
   * сброс таймера;
4. завершение задачи.

4.5.4 Добавление древесины в костер

На рисунке 54 изображен персонаж в процессе добавления древесины в костер.

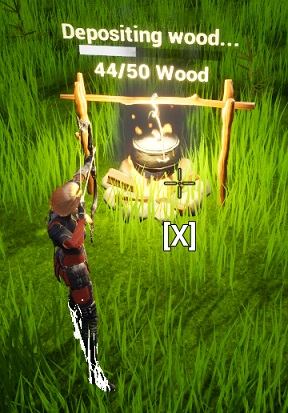


Рисунок 54 – Персонаж в добавления древесины в костер

На рисунке 55 изображено дерево поведения BT\_DepositWood, в котором реализована логика добавления древесины в костер.

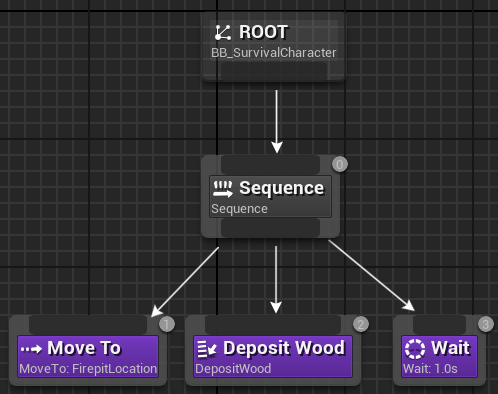


Рисунок 55 – Дерево поведения добавления древесины в костер

Персонаж может добавлять древесину в костер при соблюдении следующих условий:

* в инвентаре есть хотя бы одна единица древесины;
* количество древесины в костре меньше максимального.

Алгоритм добавления древесины в костер похож на алгоритм приготовления пищи и имеет следующие шаги:

1. движение персонажа к костру;
2. взаимодействие с костром для выгрузки дерева.

Задача приготовления пищи (Deposit Wood на рисунке 55) реализована в C++-классе UBTTask\_DepositWood. Ее алгоритм имеет следующие шаги:

1. вызов метода начала рубки дерева StartDepositing;
   * остановка всех текущих действий;
   * установка переменной состояния персонажа IsDepositing в true;
   * запуск таймера выгрузки дерева;
2. каждое срабатывание таймера:
   * обновляется прогресс выгрузки;
   * если процесс выгрузки завершен, максимальное возможное количество древесины удаляется из инвентаря персонажа и добавляется в костер;
3. вызов функции StopDepositing:
   * установка переменной состояния персонажа IsDepositing в false;
   * сброс таймера;
4. завершение задачи.

4.5.5 Поглощение пищи

На рисунке 56 изображен персонаж в процессе поглощения пищи.



Рисунок 56 – Персонаж в процессе поглощения пищи

На рисунке 57 изображено дерево поведения BT\_Eat, в котором реализована логика поглощения пищи персонажем.

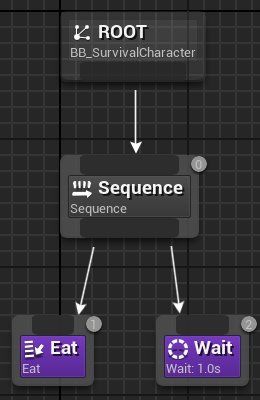


Рисунок 57 – Дерево поведения поглощения пищи

Персонаж может есть при удовлетворении следующих условий:

* в инвентаре есть достаточное количество жареного мяса;
* значение сытости (голода) меньше максимального.

Задача приготовления пищи (Eat на рисунке 57) реализована в C++-классе UBTTask\_Eat. Ее алгоритм имеет следующие шаги:

1. вызов метода начала рубки дерева StartEating;
   * остановка всех текущих действий;
   * установка переменной состояния персонажа IsEating в true;
   * запуск таймера приема пищи;
2. каждое срабатывание таймера:
   * обновляется прогресс приема пищи;
   * если процесс выгрузки завершен, определенное количество жареного мяса удаляется из инвентаря и происходит восстановление кратного ему числа единиц сытости;
3. вызов функции StopEating:
   * установка переменной состояния персонажа IsEating в false;
   * сброс таймера;
4. завершение задачи.

5. Интеграция llama.cpp в Unreal Engine

Llama.cpp – программная библиотека с открытым исходным кодом для взаимодействия с локальными LLM на языках программирования C и C++.

Разработка библиотеки ведется с марта 2023 года. Создателем llama.cpp является Георгий Герганов, на текущий момент в разработке также участвует несколько сотен добровольцев [11].

5.1 Сборка llama.cpp

В рамках данной работы используется llama.cpp версии b2928: https://github.com/ggerganov/llama.cpp/releases/tag/b2928.

Сборка выполнена в ОС Windows 10 с помощью инструментов среды разработки Visual Studio 2022.

Алгоритм сборки llama.cpp состоит из следующих этапов:

1. Скопировать репозиторий в какую-либо временную папку: git clone https://github.com/ggerganov/llama.cpp.
2. Установить CMake версии 3.17 или новее.
3. Открыть powershell в созданной временной папке.
4. Выполнить команду: cmake -B build -DLLAMA\_CUDA=1 -DLLAMA\_CUDA\_DMMV\_X=64 -DLLAMA\_CUDA\_MMV\_Y=2 -DLLAMA\_CUDA\_F16=true -DBUILD\_SHARED\_LIBS=ON -G "Visual Studio 17 2022" -A x64.
5. Результаты сборки будут находиться во временной папке в субдиректории build.

5.2 Параметры сборки llama.cpp

При выполнении команды сборки cmake задается несколько параметров.

* -DLLAMA\_CUDA=1 - использование GPU-ускорения с помощью CUDA.
* -DLLAMA\_CUDA\_DMMV\_X=64 - число значений по оси x, обрабатываемых при CUDA-деквантизации.
* -DLLAMA\_CUDA\_DMMV\_Y=2 - число значений по оси y, обрабатываемых при CUDA-деквантизации.
* -DLLAMA\_CUDA\_F16=true - использование float16.
* -DBUILD\_SHARED\_LIBS=ON - сборка .dll/.so + .lib.
* -G "Visual Studio 17 2022" - использование Visual Studio 2022 для генерации проектных файлов.
* -A x64 - сборка под 64-битную архитектуру.

5.3 Результаты сборки llama.cpp

Из всех результатов сборки в дальнейшем будут использоваться следующие файлы.

* Заголовочные (.h) файлы:
  + ggml.h;
  + ggml-alloc.h;
  + ggml-backend.h;
  + ggml-cuda.h;
  + и т.д.
* Файлы динамических библиотек (.dll):
  + build/bin/release/llama.dll;
  + build/bin/release/ggml\_shared.dll.
* Файлы статических библиотек (.lib):
  + build/Release/llama.lib;
  + build/Release/ggml\_shared.lib.

5.4 С**оздание плагина для работы с LLM в Unreal Engine**

Сперва необходимо создать новый пустой плагин. Для этого нужно выполнить следующие шаги.

1. Открыть меню Edit Plugins.

2. В открытом окне Plugins нажать на кнопку Add (рис. 58).

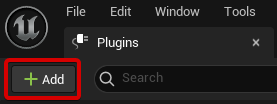


Рисунок 58 – Кнопка вызова диалога создания плагина

3. После нажатия на кнопку Add открывается диалог создания плагина. В нем нужно (рис. 59):

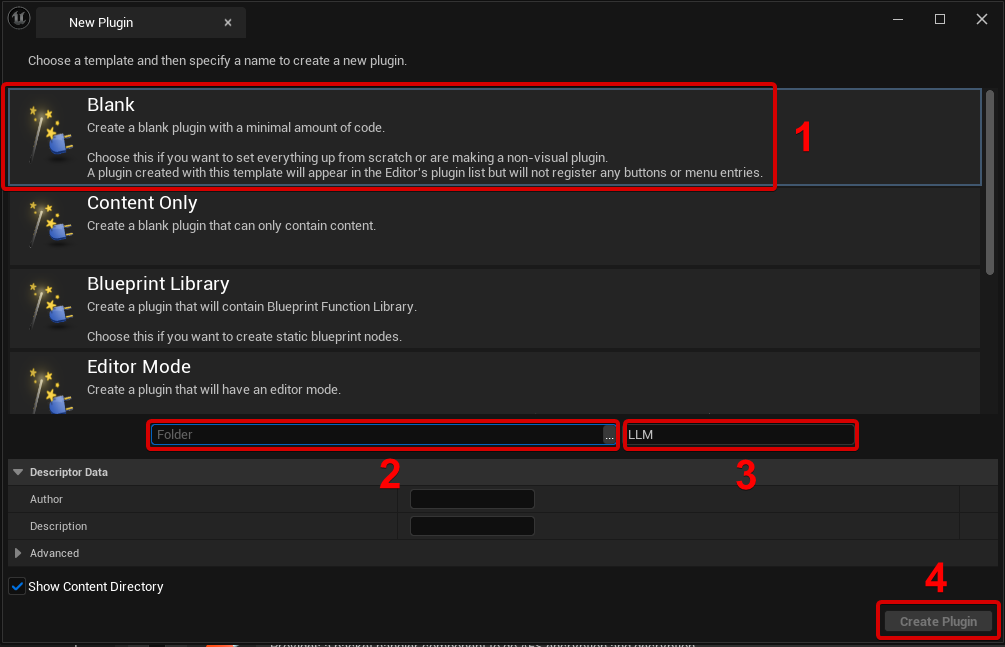


Рисунок 59 – Диалог создания нового плагина

3.1. выбрать тип шаблона плагина – в данном случае достаточно будет создать самый минималистичный вариант (Blank);

3.2. выбрать директорию, в которой будет создан плагин (обычно <директория проекта>/Plugins/<название плагина>);

3.3. задать название будущего плагина;

3.4. нажать на кнопку Create Plugin.

4. Создание плагина потребует перезагрузить проект в Visual Studio. После перезагрузки плагин будет готов к дальнейшей разработке.

5.5 Добавление файлов llama.cpp в плагин

В директории плагина (<директория проекта>/Plugins/<название плагина>) необходимо создать три дополнительные папки:

* Binaries/Win64;
* Includes;
* Libraries.

Заголовочные (.h) файлы нужно поместить в папку Includes, файлы динамических библиотек (.dll) – в папку Binaries/Win64, файл статических библиотек (.lib) – в папку Libraries.

После добавления файлов необходимо обновить Visual Studio проект:

* открыть контекстное меню Unreal Engine проекта (.uproject);
* нажать на кнопку Generate Visual Studio project files (рис. 60).

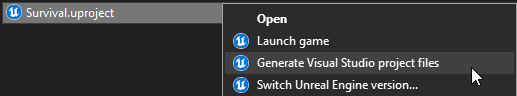


Рисунок 60 – Кнопка создания актуальных версий файлов проекта Unreal Engine

Чтобы файлы llama.cpp распознавались в рамках Unreal Engine проекта, в файле сборки плагина LLM.Build.cs нужно указать (листинг 1):

* путь к новым заголовочным файлам в папке Includes;
* пути к статическим библиотекам Binaries/Win64/llama.lib, Binaries/Win64/ggml\_shared.lib;
* пути к динамическим библиотекам Binaries/Win64/llama.dll, Binaries/Win64/ggml\_shared.dll.

Листинг 1. Правила сборки плагина для работы с LLM

public class LLM : ModuleRules

{

    public LLM(ReadOnlyTargetRules Target) : base(Target)

    {

        // ...

        // Добавление пути к заголовочным файлам llama.cpp

        PublicIncludePaths.Add(Path.Combine(PluginDirectory, "Includes"));

        // Подключение статических библиотек

        PublicAdditionalLibraries.Add(Path.Combine(PluginDirectory, "Libraries", "llama.lib"));

        PublicAdditionalLibraries.Add(Path.Combine(PluginDirectory, "Libraries", "ggml\_shared.lib"));

        // Подключение динамических библиотек

        PublicDelayLoadDLLs.Add(Path.Combine(PluginDirectory, "Binaries", "Win64", "llama.dll"));

        PublicDelayLoadDLLs.Add(Path.Combine(PluginDirectory, "Binaries", "Win64", "ggml\_shared.dll"));

        // ...

    }

}

5.6 LLMComponent и LLMWorker

В процессе работы над проектом был создан C++-класс ULLMComponent, наследующий UActorComponent. Он позволяет делать запросы к большой языковой модели и получать от нее ответы.

Для начала работы с ULLMComponent необходимо присоединить его к целевому актору и указать путь к файлу языковой модели (рис. 61).



Рисунок 61 – Пример указания пути к файлу языковой модели

Обработка запросов к модели осуществляется на отдельном потоке (thread). В API Unreal Engine существует класс для определения логики, исполняемой на создаваемых потоках – FRunnable [12]. Для задания своего собственного поведения, выполняемого на отдельном потоке, необходимо создать пользовательский класс, наследующий FRunnable, переопределить в нем чистую виртуальную функцию Run, а также, опционально, функции Init, Stop и Exit. Именно в функции Run выполняется основная работа, делегированная потоку.

Для обработки запросов к языковой модели на отдельном потоке был создан класс FLLMWorker. Он содержит функциональность для инициализации и выгрузки модели и ее контекста, а также имеет очередь для запросов, которые необходимо обработать.

При вступлении ULLMComponent’а в игру (вызове метода BeginPlay) создается инстанс FLLMWorker, выделяется новый поток. Затем выполняется вызов метода FLLMWorker::Activate, инициализирующего модель.

При выходе ULLMComponent’а из игры (вызове метода EndPlay) вызывается метод FLLMWorker::Deactivate, выгружающий модель, а инстанс FLLMWorker уничтожается (рис. 62).

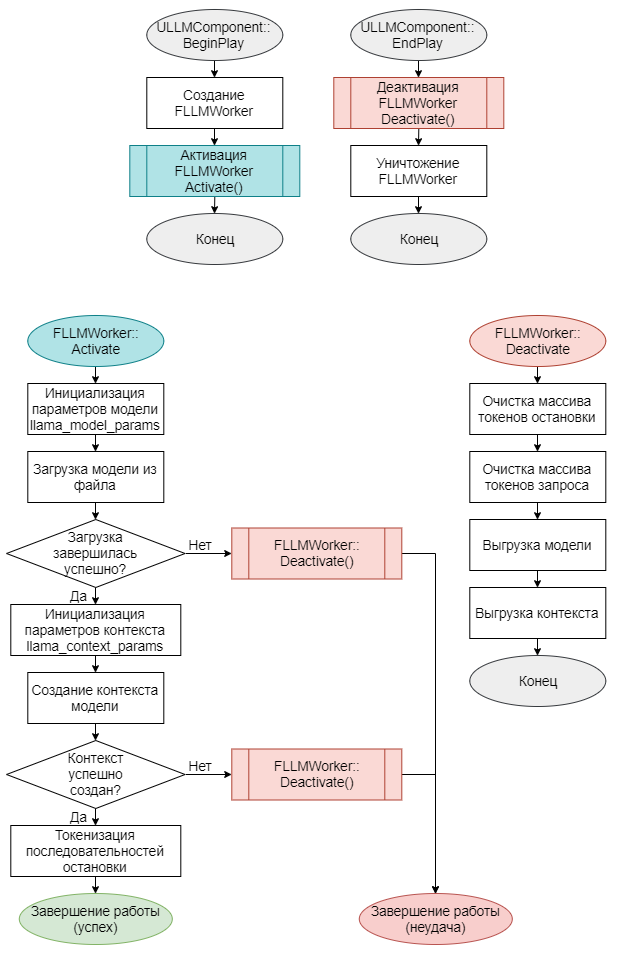


Рисунок 62 – Блок-схема инициализации и выгрузки языковой модели

FLLMWorker имеет очередь для запросов. В методе Run по мере поступления запросов к языковой модели выполняется их обработка. ULLMComponent имеет делегат OnResponseGenerated, срабатывающий при окончании ответа модели. К нему может быть привязано произвольное количество пользовательских функций.

На рисунке 63 приведена блок-схема обработки запроса (prompt’а).

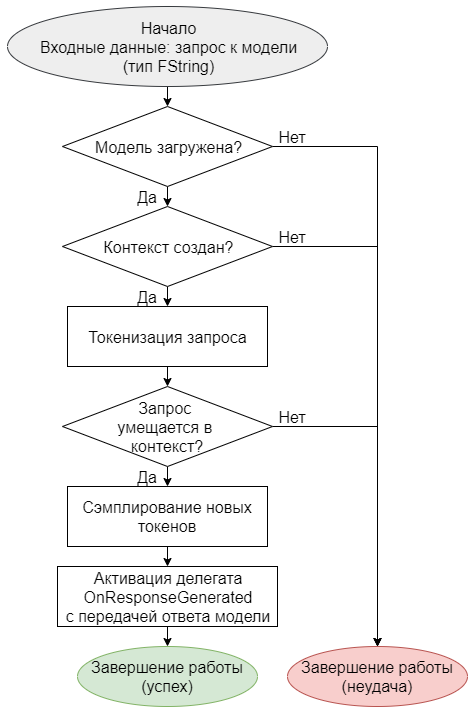


Рисунок 63 – Блок-схема обработки запроса к большой языковой модели

5.7 Выполнение запросов к языковой модели из дерева поведения

На рисунке 64 изображено основное дерево поведения BT\_SurvivalCharacter, в котором выполняется выбор действия персонажа в зависимости от ответа, сгенерированного большой языковой моделью.

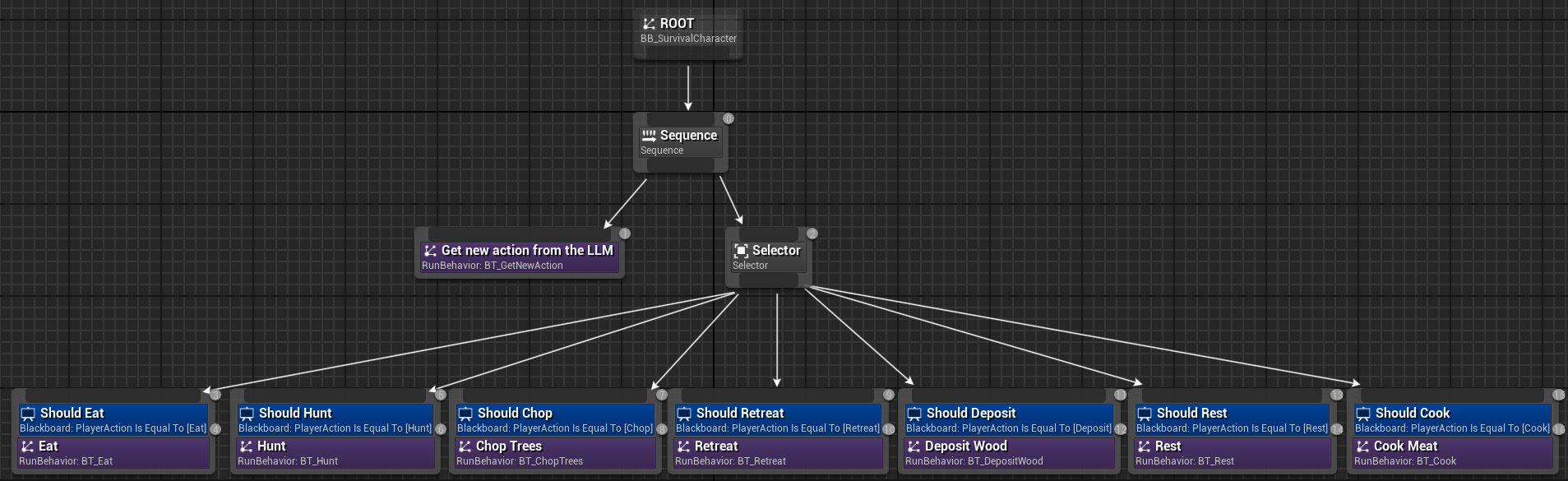


Рисунок 64 – Основное дерево поведения персонажа BT\_SurvivalCharacter

На рисунке 65 изображено дерево поведения BT\_GetNewAction, в котором выполняется запрос к LLM.



Рисунок 65 – Дерево поведения запроса к LLM

Задача приготовления пищи (Get Next Player Action на рисунке 65) реализована в C++-классе UBTTask\_GetNextAction. Ее алгоритм имеет следующие шаги:

1. формирование параметризованного запроса;
2. отправка запроса ULLMComponent’у персонажа;
3. ожидание ответа от языковой модели;
4. парсинг полученного ответа, извлечение названия действия;
5. запись названия действия в доску BB\_SurvivalCharacter;
6. завершение задачи.

6. Примеры работы приложения

Основной цикл управления персонажем имеет следующий вид.

1. Выполнение запроса к языковой модели путем подстановки параметров в шаблон запроса.
2. Получение ответа от языковой модели, извлечение из него названия действия персонажа.
3. Выполнение действия персонажа.

На рисунках 66-68 изображены действия персонажа при получении команд «охотиться», «готовить пищу» и «поглощать пищу».



Рисунок 66 – Выполнение команды «охотиться»



Рисунок 67 – Выполнение команды «готовить пищу»



Рисунок 68 – Выполнение команды «поглощать пищу»

ЗАКЛЮЧЕНИЕ

В результате выполнения работы было рассмотрено создание системы управления персонажем с помощью больших языковых моделей, объединяющей использование деревьев поведения BehaviorTree в Unreal Engine 5 и программной библиотеки для работы с языковыми моделями llama.cpp.

Поэтапно описан процесс создания трехмерной сцены с помощью ПО WorldMachine и системы процедурной генерации контента в Unreal Engine 5. Проиллюстрированы ключевые моменты настройки анимации персонажа при перемещении по уровню и выполнении игровых действий. Подробно рассмотрены алгоритмы действий персонажа, приведены соответствующие деревья поведения. Освоены принципы работы с трехмерным движком Unreal Engine 5 на языке C++. Приобретены навыки работы с ПО для процедурной генерации ландшафтов WorldMachine.

В результате работы над проектом было создано приложение с элементами игры жанра «симулятор выживания», в котором управление персонажа осуществляется путем получения ответов большой языковой модели на поступающие параметризованные запросы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Survival game // Wikipedia, the free encyclopedia [Электронный ресурс] – URL: https://en.wikipedia.org/wiki/Survival\_game. Дата обращения: 21.02.2024.
2. 14 LLMs fought 314 Street Fighter matches. Here's who won // AWS Community [Электронный ресурс] – URL: https://community.aws/content/2dbNlQiqKvUtTBV15mHBqivckmo/14-llms-fought-314-street-fighter-matches-here-s-who-won. Дата обращения: 21.05.2024.
3. Qian C., Xin C., Cheng Y., Weize C., Yusheng S., Juyuan X., Zhiyuan L., Maosong S., Wei L. Communicative Agents for Software Development // arXiv [Электронный ресурс]. URL: https://arxiv.org/pdf/2307.07924. Дата обращения: 21.05.2024.
4. Device Reference // World Machine Help [Электронный ресурс] – URL: https://help.world-machine.com/topics/reference/. Дата обращения: 21.02.2024;
5. Procedural generation // Wikipedia, the free encyclopedia [Электронный ресурс] – URL: https://en.wikipedia.org/wiki/Procedural\_generation. Дата обращения: 21.02.2024;
6. Procedural Content Generation Overview // Unreal Engine 5.3 Documentation [Электронный ресурс] – URL: https://docs.unrealengine.com/5.3/en-US/procedural-content-generation-overview/. Дата обращения: 21.02.2024;
7. PCG Basics – Unreal 5.2 Tutorial – Part 1 // YouTube [Электронный ресурс] – URL: https://www.youtube.com/watch?v=byouBYMuYPw. Дата обращения: 21.02.2024;
8. PCG Splines – Unreal 5.2 Tutorial – Part 2 // YouTube [Электронный ресурс] – URL: https://www.youtube.com/watch?v=zp1pvaZraGM. Дата обращения: 21.02.2024;
9. UMG UI Designer Quick Start Guide // Unreal Engine 5.3 Documentation [Электронный ресурс] – URL: https://docs.unrealengine.com/5.3/en-US/umg-ui-designer-quick-start-guide-in-unreal-engine/. Дата обращения: 21.02.2024;
10. UPawnSensingComponent // Unreal Engine 5.3 Documentation [Электронный ресурс] – URL: https://docs.unrealengine.com/5.3/en-US/API/Runtime/AIModule/Perception/UPawnSensingComponent/. Дата обращения: 21.02.2024;
11. ggerganov/llama.cpp: LLM inference in C/C++ // GitHub [Электронный ресурс] – URL: https://github.com/ggerganov/llama.cpp. Дата обращения: 21.02.2024.
12. Multithreading With FRunnable // Unreal Engine Community Wiki [Электронный ресурс] – URL: https://unrealcommunity.wiki/multithreading-with-frunnable-2a4xuf68. Дата обращения: 21.05.2024.

ПРИЛОЖЕНИЕ А. Графическая часть выпускной квалификационной работы

В графическую часть выпускной квалификационной работы входят 8 чертежей.

* Создание ландшафта в WorldMachine.
* Импорт созданного ландшафта в Unreal Engine.
* Процедурная генерация деревьев и травы с помощью PCG.
* Разделение леса на зоны.
* Настройка анимаций персонажа.
* Основные деревья поведения персонажа и противника.
* Поддеревья поведения персонажа (охота и рубка леса).
* Поддеревья поведения персонажа (приготовление и поглощение пищи, сон, добавление древесины в костер).