**TODO: Не забудь изменить стиль «названия рисунка», изменить сейчас нельзя, потому что Word – инвалид**

**Введение**

Развлечения всегда были неотъемлемой частью жизни человека. По этой причине в данной работе я хотел бы разобрать одно из самых передовых и быстро развивающихся технологий в области развлечений – «лазертаг», а именно программное обеспечение для клиентской части.

**Анализ проблемы**

В конце 70-х годов была представлена одна из первых версий взаимодействия устройств по ИК сигналу. Для обмена данными был придуман устоявшийся протокол передачи – «Miles tag», который дошел до современного вида лазертага. По этому протоколу построены все правила передачи информации между устройствами. Устройствами являются муляжи оружий, повязки на голову и дополнительные устройства для разнообразия сценариев игры. Муляж оружия – «тагер», умеет излучать ИК сигнал для имитации выстрела. Повязка на голову или жилет имеют несколько модулей для приёма ИК сигнала, которые оснащены световыми индикаторами. Радиобаза – устройство для передачи информацией между игроками и компьютером со специальным программным обеспечением. Программное обеспечение является способом взаимодействия клиента с оборудованием для настройки игровых параметров и введения статистики игры.

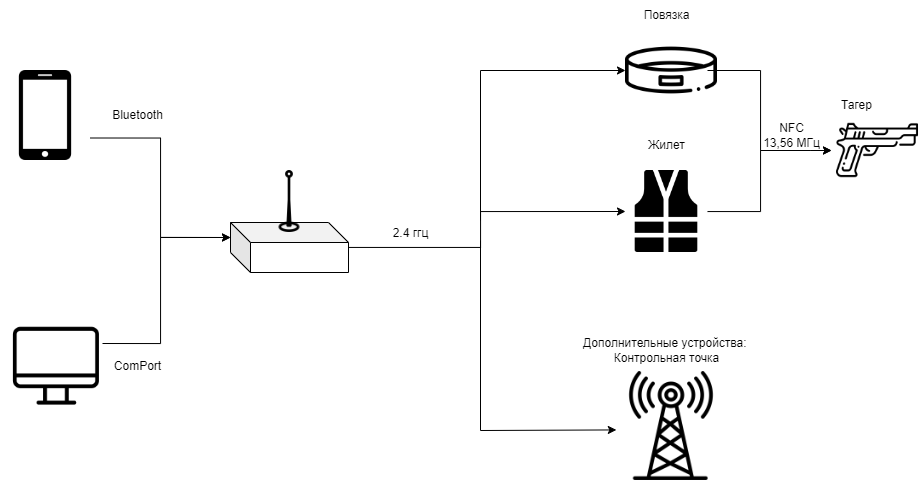


Рисунок - Абстрактное представление работы оборудования

Целью данной работы является разработка вышеупомянутого программного обеспечения, для настройки всех типов устройств специального лазертаг оборудования, учёта статистики и автоматизации игровых событий. Существует множество способов подключения радиобазы для дальнейшей передачи информации, но нужно реализовать 2 – соединение с радиобазой по COMPORT и Bluetooth, для компьютеров и телефонов соответственно.

**3 Разработка**

В данной главе рассмотрена разработка всех классов и методов для построения основного функционала приложения.

**3.1 Создание проекта и сцены под Desktop**

Меню конфигуратора включает в себя множество параметров, которые были утверждены и расставлены в дизайне приложения. Под все основные меню выделены объекты, включающие в себя компонент «CanvasGroup». Данное решение было принято за счёт удобства взаимодействия с этим компонентом. Встроенный класс был расширен двумя методами, отвечающими за прозрачность, с помощью которых реализовано открытие и закрытие основных меню.

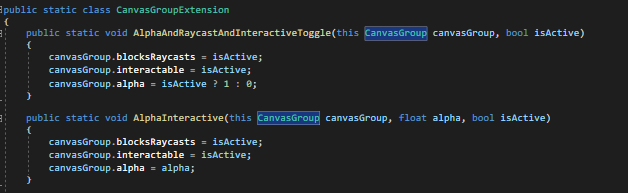


Рисунок - Методы расширения CanvasGroup

Все элементы конфигуратора были и разделены на «настройки повязки», «настройки тагера», а они, разбиты на «основные настройки» и «дополнительные настройки», согласно дизайну.



Рисунок - Пример расположения элементов конфигуратора на сцене

Каждый параметр – объект с наследником класса «MultyTypeIntParameter», который содержит в себе встроенный InputField, нижний и верхний порог значения и событие изменения параметра. Каждый наследник переопределяет метод разбора внесенного пользователем значения. Например, в поле ввода «здоровье», пользователь может занести только целое число не меньше 1 и не больше 999.

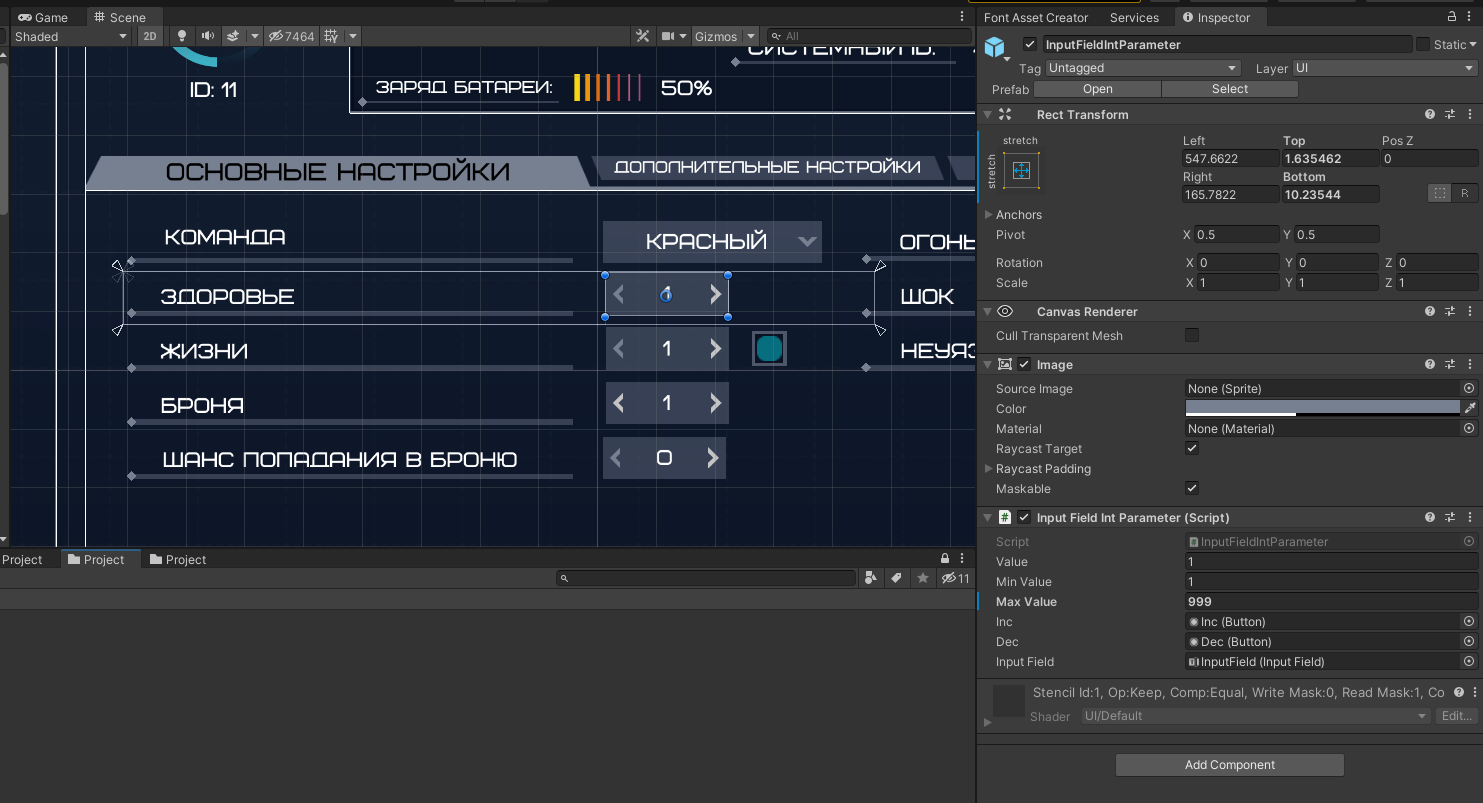


Рисунок - Параметр конфигуратора

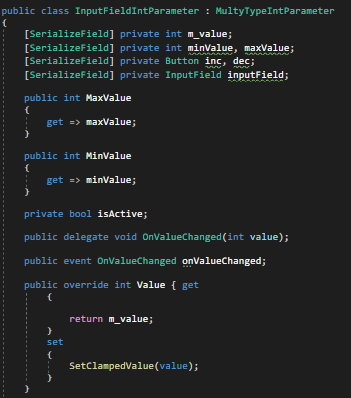


Рисунок – класс наследник MultyTypeIntParameter

Классы конфигуратора, как и UI, разделены на «конфигуратор повязки» и «конфигуратор тагера» - «HeadBandConfiguration», «TaggerConfiguration» соответственно. Каждый содержит в себе соответствующие сериализованные объекты параметров, методы инициализации девайса и добавление слушателей на события изменения параметров. **Всё это можно расписать побольше если нужно будет увеличить объем**

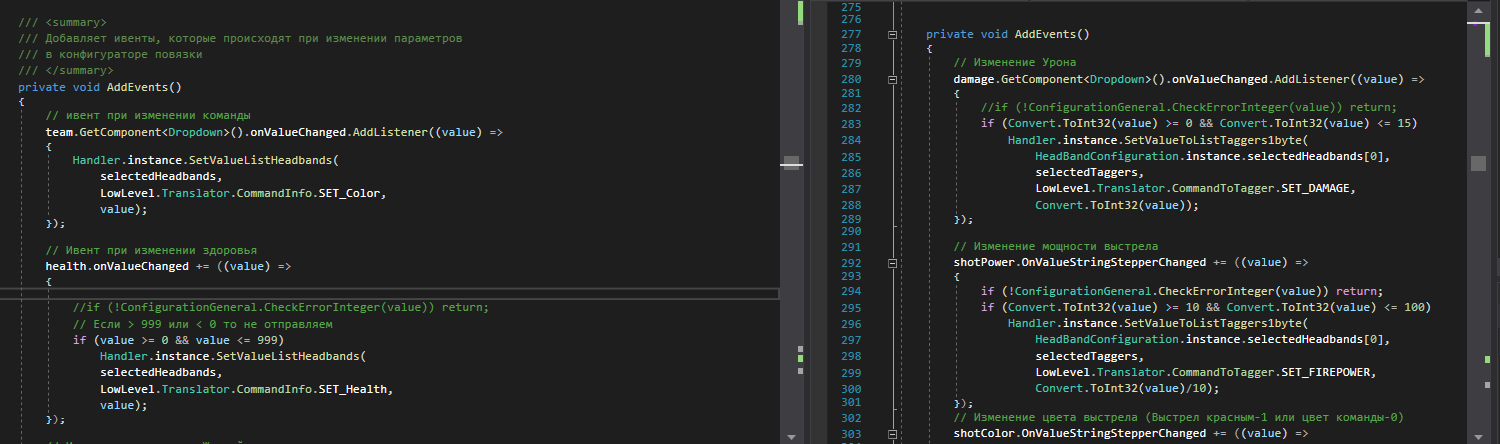


Рисунок - Обработка событий изменения параметров

Каждое изменение параметра обращается к классу одиночке Handler, через который передаются все команды на COMPORT. Он содержит в себе объект реализации интерфейса ICommander, через который идёт основной обмен данными.

**3.3 Разработка методов обмена данными**

Первое, что требовалось реализовать – многопоточность. Для разработки многопоточных приложений на Unity следует использовать сторонние библиотеки или усложнённый функционал, потому что движок запрещает вызывать почти все основные вызовы не из основного потока. Для выполнения поставленной задачи был взят готовый скрипт «UnityThread», который отслеживает возвращение из стороннего потока и синхронизируется с основным во встроенных методах – Update/FixedUpdate. При старте приложения требуется инициализировать UnityThread, после чего создаётся DontDestroyOnLoad объект, который будет производить синхронизацию ответов из других потоков. В Handler инициализируется объект типа «Intaracton», который также, как и Commader реализует все методы интерфейса ICommander. Всё это сделано для того, чтобы объект типа Interaction был мостом для вызова методов Commander, которые должны выполняться уже не из главного потока. Пример работы метода отправки данных на тагер через новый поток:

1. TaggerConfigurator обращается к Handler, чтобы отправить введенные пользователем данные:

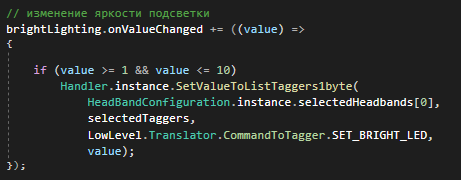


Рисунок - обращение к Handler

1. Handler вызывает метод «SetToTagger» у объекта commander, который имеет тип Interaction:

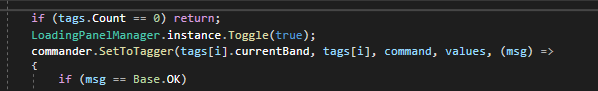


Рисунок - вызов метода в Commander

1. Класс Interaction реализовывает метод интерфейса ICommander так, чтобы создался новый поток и объект типа Commander исполнялся уже не из основного потока, а Action callback вернулся обратно в Interaction для дальнейшей синхронизации потоков в Update через UnityThread:

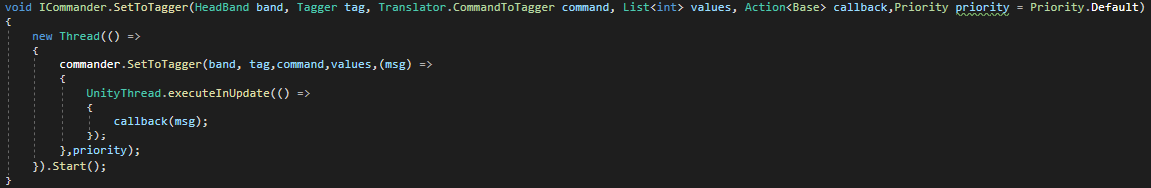


Рисунок - Создание нового потока

1. После исполнения операции через обратный вызов обрабатывается результат выполнения команды:



Рисунок - Приём ответа из метода в Commander

Автоматизация фонового взаимодействия программного обеспечения с радиобазой создала довольно весомое количество отправки и получения не важных данных, из-за чего могла возникнуть проблема переполнения буфера. Для избежания возможных ошибок, было принято разделить методы обмена на приоритеты. Например, запрос на обнаружение новых устройств, который вызывается раз в несколько секунд менее важен, чем запрос на изменения какой-либо конфигурации, которую применил пользователь. Было принято решение о добавление 3 приоритетов:

|  |  |
| --- | --- |
| Приоритет | Описание |
| Low | Низкий приоритет – выполняется только в том случае, если очередь и команд на выполнение пустая. Если на момент вызова метода с низким приоритетом очередь будет занята хотя-бы одним элементом на ожидание, то данный метод не будет выполняться и занимать очередь. |
| Default | Обычный приоритет – выполнится, как только очередь методов с обычным приоритетом дойдёт до него, а очередь с методами высокого приоритета будет пуста. |
| Hight | Высокий приоритет – метод займет очередь для высоких приоритетов и выполнится либо сразу, либо после вызванных ранее методов с таким-же приоритетом. |

Таблица - Приоритеты исполнения команд

Для управления очередями были объявлены переменные, которые хранят в себе индексы методов, которые выполняются и последних в очереди.



Рисунок - Переменные управления очередями

HereWeGoAgain – Флаг, который следит за тем, не занят ли COMPORT передачей данных.

lastQuery – Индекс последнего в очереди на выполнение метода

lastQueryHightPriority – Индекс последнего в высокоприоритетной очереди на выполнение метода

currentQuery – Индекс метода в очереди, который на данный момент выполняется.

currentQueryHightPriority – Индекс метода в высокоприоритетной очереди, который на данный момент выполняется.

COUNT\_ERROR\_WAIT\_QUEUE – Переменная хранящая в себе время до ошибки. Если на протяжении этого времени, умноженного на 10, индексы очереди не изменят значения (очередь не сдвинется), то, возможно, какой-либо метод не смог закончить своё выполнение и не вернул значение. Переменная была создана для исключения необходимости перезапуска программного обеспечения в случае ошибки в коде.

Метод занимания очереди принимает в себя 1 аргумент – приоритет, от которого алгоритм изменяет определенные индексы. По умолчанию приоритет «Default»



Рисунок - Аргумент метода ожидания очереди

Также WaitQueue возвращает ответ типа bool.

True – Можно начинать выполнение команды.

False – Следует отменить выполнение команды.

False вернется только в том случае, если будет низкий приоритет на выполнение.

Если приоритет «Default» - то WaitQueue вернёт значение на True, только тогда, когда очередь высокоприоритетных команд будет пуста и индекс очереди обычных команд дойдёт до нужной.

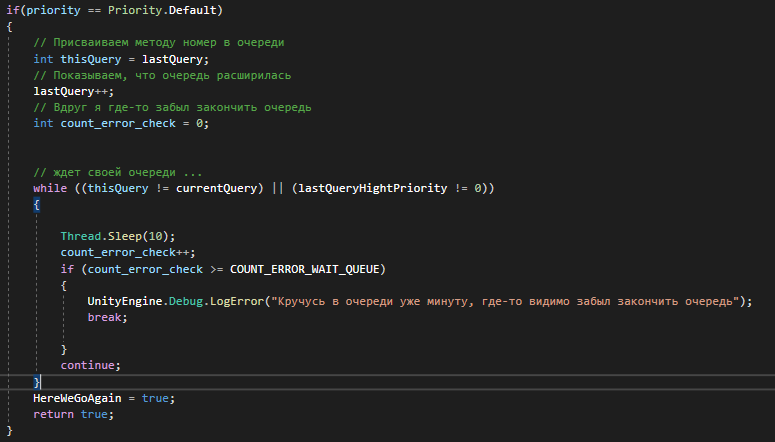


Рисунок - Алгоритм ожидания очереди для команд с обычным приоритетом

Если приоритет «Low» - то WaitQueue вернёт значение True, только если ни одна из очередей не будет занята более приоритетными задачами на выполнение, иначе вернётся False и метод вызывавший WaitQueue должен будет прекратить выполнение.

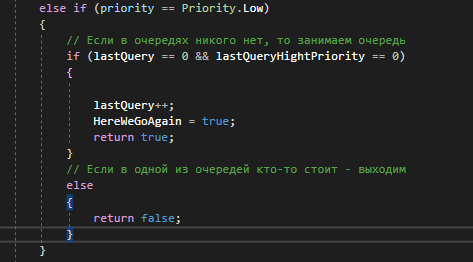


Рисунок - Алгоритм ожидания очереди для команд с низких приоритетом

Если приоритет «Hight» - то WaitQueue вернёт значение на True, только тогда, когда индекс очереди высокоприоритетных команд дойдёт до нужной.

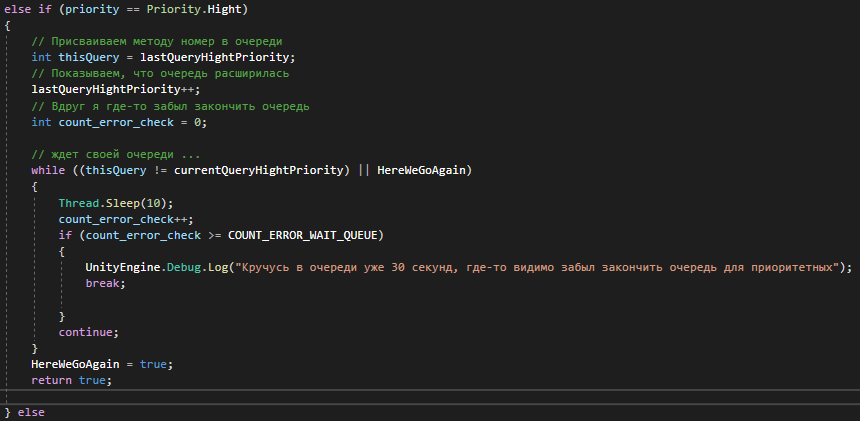


Рисунок - Алгоритм ожидания очереди для команд с высоким приоритетом

Передача и приём данных был реализован в классе Translator. Данный класс содержит в себе методы, которые взаимодействуют с классом из метаданных «SerialPort», через который и реализуется отправка и приём данных по физическому COMPORT’у. В Commander инициализирован объект типа Translator. Несколько основных методов, реализованных в классе Translator:

«Opros\_to» конвертирует все данные на оправку, переданные в аргументах, в тип byte, собирает их в массив и добавляет в буфер на отправку, через встроенный метод Write у класса «SerialPort», заранее очистив буфер. Метод «Opros\_to» переопределён под разные задачи:



Рисунок - Методы для отправки данных

«Opros\_From» принимает данные из COMPORT.

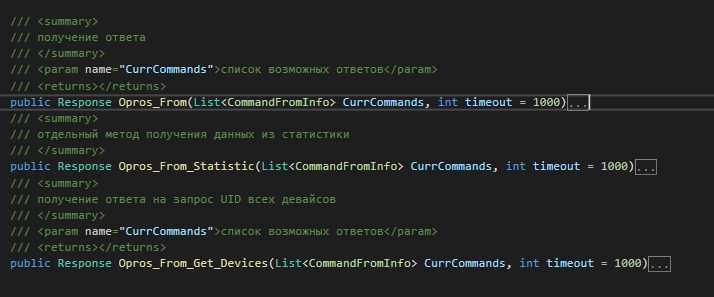


Рисунок - Методы для приёма данных

Данные с COMPORT приходят через разный интервал времени после отправки, поэтому был реализован цикл, в котором ведется опрос на наличие байт в буфере приёма.

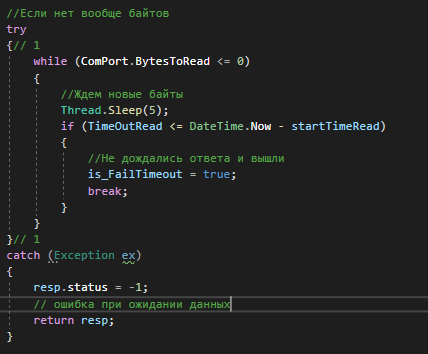


Рисунок - цикл проверки присутствия данных в буфере приёма