ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

ДИПЛОМНА РАБОТА

по професия код 481020 „Системен програмист“
специалност код 4810201 „Системно програмиране“

Тема: Multiplayer игра на Unreal Engine

Дипломант: *Александър Стоянов Хаджиев*

Дипломен ръководител: *Виктор Кетипов*

СОФИЯ

2 0 2 2

ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

Дата на заданието: 14.12.2021 г.
Дата на предаване: 14.03.2022 г.

Утвърждавам:.....
/проф. д-р инж. Т. Василева/

ЗАДАНИЕ
за дипломна работа

ДЪРЖАВЕН ИЗПИТ ЗА ПРИДОБИВАНЕ НА ТРЕТА СТЕПЕН НА ПРОФЕСИОНАЛНА КВАЛИФИКАЦИЯ
по професия код 481020 „Системен програмист“
специалност код 4810201 „Системно програмиране“

на ученика Александър Стоянов Хаджиев от 12 А клас

1. Тема: Multiplayer игра на Unreal Engine

2. Изисквания:
   - Различни видове огнестрелни оръжия.
   - 2 слота за оръжия.
   - Система за кръв.
   - Изкуствен интелект с поведенчески дървета.

3. Съдържание 3.1 Теоретична част
   3.2 Практическа част
   3.3 Приложение

Дипломант :.....
/ Александър Хаджиев /
Ръководител:.....
/ Виктор Кетипов /
Директор:.....
/ доц. д-р инж. Ст. Стефанова /

ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ СОФИЯ

Отзив от научния ръководител и предложение за рецензент

Дипломното задание е изпълнено добре. За сравнително малкото време за изпълнение, дипломантът успя да навлезе в разработката на игри с "Unreal Engine", както и да се запознае с добрите практики за писането на качествен геймплей код. Дипломантът се сблъска с различни проблеми, решения на много от които успя да измисли сам или с помощта на форумите на "Unreal Engine". Проблемите, на които не успя да намери решение се допита до мен и така спести ценно време. Опитът, който дипломантът придоби по време на разработката ще му е от голяма полза в случай, че реши да продължи развитието си в разработката на игри.

На база на използваните технологии предлагам за рецензент Александър Ангелов.

Ръководител: .....

/Виктор Кетипов/

Глава 0

Увод

Целта на дипломната работа е да се създаде игра, която чрез нейните функционалности позволява лесно надграждане, което ще позволи по-голяма креативност и по-добър продукт.

Механиките може да са най-простите - оръжия, система за кръв, обаче ако тези системи бъдат направени точно и добре, то тогава надграждането с нови функционалности не само ще е лесно, но и новите функционалности ще се интегрират със базовите да създадат по-добро преживяване на играча.

Глава 1

Разработка на игри с Unreal Engine 4

1.1 Програмиране на C++ в Unreal Engine

1.1.1 Класове и обекти в Unreal Engine

UObject[9]

Почти всичко в една Unreal Engine игра е UObject. UObject е класа за всички обекти и в него се имплементират неща като garbage collection, UPROPERTY макро за работа с променливи в Unreal Editor, мрежова репликация и други.

AActor[1]

AActor е клас за обект който може да бъде поставен в ниво. Той наследява UObject класа. В него има имплементация за multiplayer репликация, има особен начин за garbage collection и има tick. Тъй като World Object пази списък от всички Actor обекти, те не могат да се изчистват по стандартния начин, и затова ако един Actor трябва да бъде изтрит се вика Destroy() на този обект и получава състояние "pending kill"и остава до следващия garbage collection.

Към Actor класа могат да се добавят компоненти, които добавят функционалности към един такъв обект. При създаването си те се асоциират с Actor обектът, в който са създадени. Всички компонен-

ти в един Actor са йерархични, като има променлива RootComponent от тип USceneComponent, която определя основния компонент. Самите Actor обекти нямат 3D трансформации и затова те разчитат на главния си компоненент за тези функционалности. В случаи че главният компонент не поддържа 3D трансформации, Actor обектите също няма да имат тази функционалност. Всички други прикрепени компоненти имат трансформация относителна с основния компонент.

Actor компоненти[3]

Actor компонентите са няколко типа, като всички наследяват UObject. Те не могат да съществуват сами и винаги трябва да бъде прикрепени към Actor обект като събобекти. При създаване на нов Actor обект, всички компоненти, които се съдържат в него се създават специално за този инстанциран обект - всеки инстанциран Actor обект има своите уникални компоненти.

Най-важните типове компоненти биват:

- Actor компоненти (клас UActorComponent) - най-полезни за абстрактни функционалности като движение, инвентар, система за кръв и други нефизически идеи. Те нямат трансформации, което означава че нямат физическо място в света. Всички други компоненти наследяват от тях.

- Сценови компоненти (клас USceneComponent, наследява UActorComponent) - компоненти, които поддържат поведения базирани на позиция, но които не се нуждаят от геометрично изобразяване. Такива биват пружинни рамена (spring arm), камери, физически сили или препядствия (но не физически обекти) и аудио.

  Трансформациите на сценовите компоненти се случват чрез класа FTransform, който съдържа локация (позиция), ротация и мащаб на компонента.

  Сценовите компоненти могат да сформират дървета, поради това че те са йерархични. Един Actor обект може да зададе един компонент като своя основен (root), като по него се определят позицията, ротацията и мащабът на този обект. Единствено сценови

компоненти (вички, които наследяват USceneComponent), могат да се закачат едни за други, тъй като от това зависи тяхна трансформация. Всеки сценови компонент може да има произволен брой деца, но само един родител.

Има два начина на закачане на сценови компоненти един за друг. Първият, който работи за компоненти, които не са все още регистрирани и се ползва в конструктори, е чрез функцията SetupAttachment. Другият начин, който е по-полезен по време на игра е чрез AttachToComponent функцията, която веднага закача един компонент за друг. Системата за закачане позволява и два Actor обекта да бъдат закачени един за друг, ако основният компонент на единия се закачи за компонент на другия.

- Примитивни компоненти (клас UPrimitiveComponent, наследява USceneComponent) - сценови компоненти с геометрично изобразяване които се използват за да се рендърнат визуални ефекти или за да се застъпят или сблъскат с физически обекти. Това включва статични или скелетни мешове, спрайтове, particle системи, както и кубични, сферични и капсулни обеми.

  Най-разпространените примитивни компоненти са кубичните (Box Component), сферичните (Sphere Component) и капсулни (Capsule Component), които са невидими геометрии в пространството и се ползват за колизия и статични (Static Mesh Component) и скелетни (Sleketal Mesh Component) мешове, които съдържат пре-генерирана геометрия за рендърване и могат да се ползват за детекция на колизия.

За да могат компонентите да се актуализират на всеки кадър, Unreal Engine трябва да ги регистрира. Обикновенно, когато един Actor се инициализира, с него се инициализират и неговите компоненти. Когато те се инициализират, те се и регистрират. Това става обаче само за компоненти, които се създават като събобекти в поцеса на създаване на Actor обектът. След това за компоненти съсздадени в процеса на игра, Unreal Engine позволява ръчно извикване на RegisterComponent функцията, която регистрира обектът, ако той е вече асоцииран със съответният Actor обект. За да бъде премахнат един Actor компонент от

актуализация, симулация и рендър, той се дерегистрира с функцията UnregisterComponent.

Актуализация на компоненти

По подразбиране компонентите не се актуализират. Ако един компонент трябва да се актуализира то няколко неща трябва да се извикат:

- Първо - в конструктура на този компонент да се сложи PrimaryComponentTick.bCanEverTick на true.
- Второ - или в конструктура, или някъде другаде да се извика функцията PrimaryComponentTick.SetTickFunctionEnable(true). След това за спиране на актуализация се ползва PrimaryComponentTick.SetTickFunctionEnable(false).

Ако компонентът не се нуждае от актуализация, PrimaryComponentTick.bCanEverTick се оставя на false, което подобрябва производителността.

Компонентите се актуализират с функцията TickComponent, която работи по подобен начин на Tick функцията за Actor обектите.

Пешка (Pawn)[6]

Класът за пешка (Pawn) е базовият клас за всички Actor обекти, които могат да бъдат притежавани (Possesed) от контролери. Пешките могат да бъдат контролирани от изкуствен интелект или от играча. Класът дефинира това как играчът или изкуственият интелект изглежда визуално, но и също така как си взаимодейства със света чрез колизия и други физически интеракции. Въпреки че в някои случаи пешките нямат модел или аватар, те са физическото представяне на изкуствения интелект или играча.

По стандарт един контролер може да контролира само една пешка. Също така пешки създадени по време на игра не се притежават от контролери автоматично.

DefaultPawn класът се различава от базовия Pawn клас с това, че добавя нови компоненти и функционалности, които ги няма в базовия клас. Класът съдържа вроден компонент

DefaultPawnMovementComponent, сферичен компонент за колизия и компонент за статичен меш. За да може компонента за движение DefaultPawnMovementComponent да бъде контролиран има булева променлива, която определя дали класа да използва стандартните клавиши за движение, като тя е зададена като true по стандарт.

Компонентът за движение DefaultPawnMovementComponent дефинира движение в стил летящо и без гравитация. Освен стандартните променливи намиращи се в компонентита за движение, този компонент съдържа в себе си и променливи за максимална скорост (MaxSpeed), за ускорение (Acceleration) и за забавяне. (Deceleration). Тези променливи са достъпни за всички Blueprint обекти, които се получават от DefaultPawn класа.

SpectratorPawn класът е подклас на DefaultPawn, който се използва за спектриране на играта. Разликата между този клас и DefaultPawn е че неговият модел не се създава, което го оставя само със сфера за колизия. SpectratorPawnMovementComponent е компонентът, който класът за спектриране използва, като той дефинира същия начин за движение като DefaultPawnMovementComponent, но с добавен код за обработване или игнориране на забавяне на времето, когато е необходимо.

Герой (Character)[2]

Героят е пешка, която е предначначена за вертикален играч. Това означава, че той по дефиниция може да ходи, тича, скача, плува и лети през света. Класът също така има имплементация за мрежови функционалности и входове. Това се получава чрез добавката на скелетен меш за модела, който позволява сложни анимации, капсулен компонент, който отговаря за колизията и CharacterMovementComponent, който отговаря за движението на героя.

Капсулният компонент се използва за колизия свързана с движението на героя. За да могат да бъдат пресметнати сложни изчисления свързани с движението, компонентът за движение приема, че колизионния компонент на героя и вертикална капсула.

Компонентът за движение позволява на аватари, които не използват rigid body да се движат. Този компонент е специфичен за класа на героя и само той може да го ползва. Част от нещата, които могат

да бъдат конфигурирани в компонента са триене по време на търкане и падане, скорост по време на движение във въздуха, водата или земята, гравитационна скала и какви физически сили героят може да упражнява върху физически обекти.

1.1.2 Таймери в Unreal Engine[7]

Таймерите в Unreal Engine са начин за планиране на функции да бъдат изпълнени след дадено време или да бъдат изпълнени продължително с разстояние между извикванията.

Таймерите се контролират от глобален управител (manager) на таймерите - класът FTimerManager. Той съществува глобално на инстанията на играта и също така на всеки свят (World). Таймерите се задават с две главни функции - SetTimer и SetTimerForNextTick, като и двете функции поддържат множество от различни параметри. Всяка от двете функции може да се закачи въхру всеки тип обект или върхи делегат за функия, като също така SetTimer може да се използва за да се повторени извиквания на функция през даден интервал. Ако обектът върху който таймерът ще бъде извикан бъде унищожен преди приключване на чакането, функцията няма да бъде извикана и handle структурата на таймера ще бъде инвалидирана.

За да се достъпи управителя на таймерите обикновенно се използва AActor класът, като се използва функцията GetWorldTimerManager, която извиква GetWorldTimerManager функцията дефинирана в UWorld класа. За да се използва глобалния управител на таймери се използва GetTimerManager от UGameInstance класа. Глобалния управител може също така да бъде използван в случай че светът няма собствен управител на таймерите и ако извикванията на функциите от глобалния управител на зависят от съществуването на специфичен свят.

Таймерите в C++ могат да работят и с TFunction обекти и с делегати.

Задаване и изчистване на таймери

С функцията SetTimer, която е част от управителя на таймерите, може да се зададе таймер да изпълни функция или делегат след определено

време или извикването да се повтаря безкрайно. Тези функции използват структура за управлението на таймерите - FTimerHandle, чрез която те могат да бъдат спрени или пуснати на ново, може да се провери или промени оставащото време или да се спре таймерът.

Друг начин за използването на таймери е чрез функцията SetTimerForNextTick, която извиква функцията подадена като аргумент на следващия кадър. Тази функция обаче не приема структурата FTimerHandle.

За да се изчисти един таймер, структурата използвана за да се започне този таймер трябва да бъде подадена на функцията ClearTimer, която е част от FTimerManager. След извикването подадената структура ще се инвалидира и ще може да бъде ползвана за нов таймер. Друг начин за спиране на таймер е ако се започне нов със същата структура. Ако се извика SetTimer със стуктура, която е вече ползвана за започване на таймер, предишният таймер ще спре и ще се и ще бъде заменен с нов.

За да се изчистят всички таймери асоциирани с даден обект се извиква ClearAllTimersForObject.

За паузиране на таймери се използва функцията PauseTimer. При паузиране, таймерът запазва оставащото си време и само спира способността му да извика фунцията, която му е зададена. За да започне отново таймерът се използва UnPauseTimer.

За да се провери дали един таймер е активен се използва функцията IsTimerActive, която приема като аргумент структурата, с която търсеният таймер е асоцииран.

1.1.3 Делегати в Unreal Engine[4]

Делегатите са начин функции да бъдат извикани върху даден обект по безопасен начин. Делегат може да бъде динамично обвързан с функция принадлежаща на произволен обект и след това да бъде извикана върху този обект по-късно, независимо дали този който вика функцията знае типа на обекта. Делегатите могат да бъдат копирани безопасно. Делегатите също така могат да бъдат подавани по стойност, но това не е препоръцително, защото процеса алокира памет на heap. Препоръчително е където е възможно делегатите да бъдат подавани чрез

референции.

Има три типа делегати които се поддържат от Unreal Engine:

- Единични (Single)
- Мултикаст (Multicast)
- Динамични (Dynamic)

Деклариране на делегати

Декларирането на делегати става чрез определени макрота, които са базирани на сигнатурите на функциите, които биха се закачили за делегата. Всяко макро има параметри за името на новия тип делегат заедно с типа, който функцията връща (ако не е void) и параметрите на функцията. Делегатите поддържат всякакви комбинации от:

- Функции, които връщат нещо.
- Функции, декларирани като const.
- Най-много четири товарни (payload) променлви.
- Най-много осем функционални параметри.

Делегатите имат същите спецификации като UFUNCTION функциите, но за делегатите се използва UDELEGATE макрото.

Делегатите могат да бъдат декларирани в глобален обхват, в или извън namespace и в дефиницията на клас, но не могат да бъдат декларирани в тялото на функция.

1.1.4 Проследявания (Traces) в Unreal Engine[8]

С проследяванията в Unreal Engine е възможно да се провери какво се намира на пътя на дадена линия. Тази линия се определя като ѝ се дадат начални и крайни координати и системата за обработване на физиката създава линия между двете точки, като връща всички Actor обекти с колизия, които са на пътя на линията. Линейните проследявания са аналогични на Raycasts и Raytraces, които се използват в други

Function signature	Declaration macro
`void Function()`	`DECLARE_DELEGATE(DelegateName)`
`void Function(Param1)`	`DECLARE_DELEGATE_OneParam(DelegateName, Param1Type)`
`void Function(Param1, Param2)`	`DECLARE_DELEGATE_TwoParams(DelegateName, Param1Type, Param2Type)`
`void Function(Param1, Param2, ...)`	`DECLARE_DELEGATE_<Num>Params(DelegateName, Param1Type, Param2Type, ...)`
`<RetValType> Function()`	`DECLARE_DELEGATE_RetVal(RetValType, DelegateName)`
`<RetValType> Function(Param1)`	`DECLARE_DELEGATE_RetVal_OneParam(RetValType, DelegateName, Param1Type)`
`<RetValType> Function(Param1, Param2)`	`DECLARE_DELEGATE_RetVal_TwoParams(RetValType, DelegateName, Param1Type, Param2Type)`
`<RetValType> Function(Param1, Param2, ...)`	`DECLARE_DELEGATE_RetVal_<Num>Params(RetValType, DelegateName, Param1Type, Param2Type, ...)`

Фигура 1.1: Таблица с примери как се декларират делегати.[4]

програми. Линейното проследяване е сигурно и евтино от страна на ресурси нужни за пресмятане.

Ако проследяванията са извикани от Actor обект, проследяванията могат да игнорират този обект и неговите компоненти, което позволява на обекта да прави проследявания през себе си.

Проследяване по канали (Channels) или по обекти (Objects)

Тъй като линейните проследявания използват системата за физика, те могат бъдат настроени с какво да взаимодействат. Двете основни категории са канали и обекти. Каналите се използват за видимост и за камерите и почти винаги са свързани с проследявания. Обектите са физическите обекти с колизия в нивото - Actor обектите и техните компоненти. Възможно е добавянето на нови канали и обекти при нужда.

Единично или множествено проследяване

Unreal Engine позволява проследявания на само един блокиращ удар (единично проследяване) или проследяване на множество ударени

обекти, като последният елемент от масива от удари е блокиращ а останалите са припокриващи се. Функцията за множествено проследяване генерира единствено най-близкия блокиращ удар и не прави повече тестове след това.

И функциите за единично и функциите множествено проследяване позволяват проследяване по канал и или по обект.

Проследяване чрез форми

Освен проследявана чрез линия, Unreal Engine позволява и проследявания чрез форми - кутия, капсула и сфера. Тези проследявания също могат да бъдат единични или множествени и да проследяват или по канал, или по обект. Тъй като тези проследявания използват фигури, допълнителна информация за размера и ориентацията на тези фигуре е нужна, освен началото и края на проследяването. Също така проследяванията чрез форми са по-сложни за пресмятане тъй като се използват форми, а не линии.

Връщане на резултати (Hit Result)

Функциите за проследяване връщат структура HitResult, която съдържа следната информация:

Член на структурата	Дефиниция
Actor	Actor обектът ударен от проследяването.
bBlockingHit	Индикира дали ударът е резултат от блокираща колизия.
BoneName	Името на bone (от ударения скелетния модел), ако такъв модел е ударен.
bStartPenetrating	Дали проследяването е започнало с проникване - с първоначално блокиращо припокриване.
Component	Примитивният компонент ударен от проследяването.
Distance	Разстоянието от TraceStart до Location в пространството на света.
ElementIndex	Индекс към предмета който е ударен. Зависи от ударения примитив.
FaceIndex	Индекс на удареното лице (за сложни припокривания с триъгълни модели).
ImpactNormal	Нормалът на удара в пространството на света, за обектът ударен от sweep, ако има такъв.
ImpactPoint	Локация в света на контакта на формата за проследяване с ударения обект.
Item	Още информация за предмета, който е ударен (зависи от ударения примитив).
Location	Място в света, където движещия се обект би се намирал срещу ударения обект, ако има ударен обект.
MyBoneName	Името на собствения bone, който е участвал в удар с друг скелетен модел (при удар на два скелетни модела).
Normal	Нормалът на удара в света за обекта който е пометен.
PenetrationDepth	
PhysMaterial	Физическия материал който е ударен.
Time	"Времето"на удара покрай проследената посока (варира от 0.0 до 1.0), ако е имало удар, индикирайки времето между TraceStart и TraceEnd.
TraceEnd	Крайната локация на проследването, но не къде ударът е станал, а най-крайната точка на проследяването.
TraceStart	Началната локация на проследяването.

Таблица 1.1: Информацията съдържаща се в HitResult структурата.[5]

Тя можа да бъде ползвана за намиране на уцелени обекти и техните компоненти. Чрез етикети могат да се търсят определени обекти и да се създава логика спрямо това.

Глава 2

Подбор на технологии за разработка на игри с Unreal Engine 4

2.1 Функционални изисквания към разработката

Разработката на играта има за цел да постави основите на качествено разработен проект, който ще позволява лесно надграждане върху него.

Всички класове са имплементирани на C++.

2.1.1 Изисквания към оръжията

- Параметри на оръжията, които позволяват висока разновидност.
- Функция за пълнител и презареждане на пълнителя.
- Функция за приближаване (zoom).
- Различни режими на стрелба.
  - Полуавтоматичен (Semi-automatic).
  - Залпов (Burst).
  - Автоматичен (Automatic).

- Стрелба и откриване на удари чрез лъчево проследяване.

2.1.2 Изисквания към системата за кръв

- Щит (Shield), които се възстановява за определено време, след поемане на удар.
- Живот (Кръв, Health), който не се възстановява от самосебе си.
- Предмет, който може да бъде използван от играча, ако животът му е под максималния.

2.1.3 Изисквания към слотовете за оръжия

- Възможност за пазене на 2 оръжия и смяна между двете по време на игра.

2.1.4 Изисквания към изкуствения интелект

- Навигиране на ниво.
- Намиране на играча чрез поглед.
- Стрелба по играча.

2.2 Избор на Unreal Engine 4

За игрови двигател е избран Unreal Engine, като по-точно версия 4.27. Разликата между версия 4.27 и 4.26 е най-вече във визуалните подобрения, но най-важното за тази версия, отнасящо се до разработката на проекта е новите алгоритми за компресиране на файлвоте нужини на Unreal Engine и файловете използвани от разработчиците на проекта. Това прави крайния продукт по-малкът от страна на размер, скорост на зареждане и предаване на мрежови пакети.

Unreal Engine е избран вместо други игрови двигатели поради следните причини:

- Използва се C++ за разработка на функционалностите.

- Кодът му е свободно достъпен за всеки разработчик.
- Разработен е с 3D игри в предвид и позволява лесна работа в 3D пространството.
- Има всичко нужно за разработването на игри и в повечето случаи не се нуждае от външни добавки за да може да върши добра работа.

Разликата между Unreal Engine и други игрови двигатели, Unity например, е че Unreal Engine е направен за 3D игри. Unity е фокусиран върху 2D и мобилни игри, а Unreal Engine - към 3D игри за настолни компютри и конзоли. Разликата между Unreal Engine и други по-известни и големи игрови двигатели, е това че кодът на Unreal Engine е напълно отворен за четене от всеки разработчик на продукти с Unreal Engine. Това позолява по-лесно откриване на проблеми в кода и по-добро разбиране на базовите класове, върху които работят всички други части на Unreal Engine.

2.3 Избор на програмен език

За разработка на игри в Unreal Engine 4 се използва C++. При стандартна работа със C++ в Unreal Engine се използват макрота, които позволяват по-лесна работа с класовете и дефиниране на типовете нужни за работа с обектите в Unreal Engine.

2.4 Избор на среда за разработка

2.4.1 Игрови редактор (Unreal Editor)

За игрови редактор е избран Unreal Editor - редакторът, който идва с Unreal Engine. Той има много функционалности, които са нужни за направата на игри, като например редактор на схемите за анимация, редактор на системите за чатици, редактор на техните шаблони и много други. Също така интеграцията му с кода става лесно, като нов код може да се компилира без да се налага редакторът да бъде затворен. Възможна е и интеграцията с GitHub.

2.4.2 IDE (Integrated Development Environment)

За среда на разработка на прокета е избран Visual Studio 2019 Community Edition. Той се интегрира лесно с Unreal Engine, дебъгването с него става лесно и компилирането на проекта става сравнително бързо. IntelliSense, който е част от Visual Studio помага на по-бърза разработка, завършването на кода работи добре и се интегрира добре с начина на писане на код за Unreal Engine.

Visual Studio също така се интегрира лесно с GitHub и повечето действия са възможни направо от средата за разработка.

2.4.3 Среда за контрол на версиите

За контрол на версиите е избран github, както и интеграцията му с git LFS (Large File Storage). Git LFS позволява качването на големи файлове, най-често бинарни, които се използват често при работата с Unreal Engine. Github позволява лесен начин за интегриране на нови свойства и промени на проекта, както и проверка на минали версии на проекта.

Глава 3

Разработка на проекта

3.1 AMultiplayerFPSFirearm

Класа за оръжие дефинира обект, който може да бъде персонализиран по много начини, което позволява голямо разновидности от възможни оръжия и това е основната цел на класа - с този клас да могат да бъдат създадени голям брой различни и уникални оръжия.

3.1.1 Дефиниция на класа

Преди дефиницията на класа е дефиниран един Enum, който отговаря за режимите на стрелба. Чрез него се определя как оръжието се държи когато спусъкът бъде натиснат. Масивът от този Enum (EFireMode) - е масивът, който отговаря за позволените режими на стрелба.

В самия клас са дефинирани моделът на оръжието, кутията му за колизия, която все още не се използва, няколко FTimerHandle структури, които отговарят за стрелбата на оръжието, променливите, които дефинират интервалите, през които оръжието стреля. Също така са дефинирани и функциите за стрелба, презареждане, смяна на режима на стрелба и приближаване.

Масивът за задържано стреляне (HeldFiringIntervalsArray) е огледален на другите два масива отнасящи се за режимите на стрелба - FireModesArray (масивът, който съдържа всички позволени за това оръжие режими на стрелба) и bIsHeldArray, който определя дали оръжието може да бъде стреляно чрез задържане на бутона за стрелба.

3.1.2 Конструктор на класа

В конструктора на класа първоначално се инизиализира моделът чрез шаблона CreateDefaultSubobject. След това на модела са задава играча да бъде единствения, който ще го вижда. След това генерирането на сенки е спряно за да не се виждат. След инициализацията на моделът, той е зададен като главния компонент.

Следва инициализацията на кутията за колизия, която също бива инициализирана с шаблона CreateDefaultSubobject. След задаването на размер и позиция спрямо модела, кутията се закача за моделът като дете в йерархичната структура.

След инициализацията на моделът и кутята се задават стойностите по подразбиране на променливите, които отговарят за залповия режим на стрелба, както и времето за презареждане.

3.1.3 BeginPlay

В BeginPlay на оръжието се задава текущия режим на стрелба и се задава стойноста на променливата CurrentMagazineCapacity. Стойността на пълнителя се задава в BeginPlay, защото максималният обем на пълнителя се задава в редактора и за да може при стартиране на игра играчът да има пълен пълнител, поменливата получава стойността си в тази функция. Текущия режим на стрелба се задава като първия елемент от масива, който съдържа всички възможни за това оръжие режими на стрелба. Ако случайно масивът е празен, играта ще спре, защото се прави достъп на разен масив.

3.1.4 Стрелба

Логиката на стрелбата се случва в класа на оръжието. Стрелянето започва от функцията StartFiring, която определя как оръжието стреля спрямо зададените стойности. В тази функция първо се проверява дали случайно текущият режим на стрелба не е част от масива, който определя режимите на стрелба достъпни на оръжието. Ако текущия режим не е част от от масива с достъпни режими, функцията извърля грешка и приключва. Ако режимът е част от масива - функцията продължава.

След това се проверява какъв всъщност е режимът на стрелба и в зависимост от него, се извиква съответния блок от код:

- Ако режимът на стрелба е автоматичен се започва таймер, който ще изстрелва проследявания (куршуми) през определен брой време, което е определено от променливата от масива, която съвпада с индекса на режима на стрелба. Този таймер ще бъде активен и ще извиква функцията Fire, която отговаря за стрелянето на проследявания, докато бутонът за стрелба не бъде пуснат, което кара функцията StopFiring да се извика и да спре таймера за стрелба с FTimerHandle - HeldFiringInterval Timer.

- Ако режимът за стрелба е залпов - проверява се дали в масива bIsHeldArray стойността отговаряща на индекса на режима на стрелба е true и ако е, то оръжието може да бъде стреляно чрез задържане на бутона за стрелба. Ако това обаче не е възможно - стойността е false, то оръжието може да бъде стреляно в този режим само при натискане, а не задържане на бутона за стрелба. При позволено задържане на бутона за стрелба се стартира таймер, чиито интервал се намира на индекса огледален на индекса на режима на стрелба в масива HeldFiringIntervalsArray. На всяко извикване на този таймер извиква функцията BurstFire, която в своето тяло пък започва друг таймер, който отговаря за отделното изстрелване на всеки куршум в даден залп. Броячът BurstsFired отговаря за проверката да не бъдат изстреляни повече куршуми в даден залп от необходимото (променливата, която дефинира броя куршуми на всеки залп е BurstCount).

- Ако режимът за стрелба е полуавтоматичен се проверява дали спусъкът може да бъде задържан и ако да - се започва таймер, който извиква Fire, докато спусъкът не бъде пуснат. Ако обаче спусъкът не може да бъде задържан, Fire се извиква само веднъж.

Във функцията Fire се изстрелва една линия за проследяване и когато тя удари играч, се проверява ударения компонент. Ако този компонент има етикет "HeadHitbox играчът поема повече щета, а ако

има етикет "BodyHitbox той поема нормална щета. Правенето на щета върху уцеления играч става чрез TakeDamage функцията.

3.1.5 Смяна на режимите на стрелба

В оръжията могат да се съдържат повече от един режим на стрелба. Това става с помоща на масив, в който се пазят възможните за оръжието режими на стрелба. Този масив се пълни от дизайнера в редактора, което позволява разновидност на оръжията.

Когато играчът иска да смени режимът на стрелба на оръжието, първо се проверява размерът на масива и ако в него се съдържа само един елемент, функцията връща съобщение, че оръжието притежава само един режим на стрелба. Масивът не може да бъде празен, тъй като в BeginPlay се достъпва първият елемент на масива и ако масива е празен програмата ще се счупи и ще спре.

След проверката, ако масивът има повече от един елемент се търси индексът на текущия режим на стрелба (CurrentFireMode) и ако той не бъде открит по накаква причина, функцията изхвърля грешка в логовете на редактора и приключва изпълнението си. Ако индексът е намерен, се проверва дали индексът на текущия режим прибавен с едно е по-голям от последния елемент на масива. Ако той е по-голям от последния елемент, като текущ режим на стрелба се задава първият елемент на масива, а ако не е - се задава следващият елемент в масива. Тази имплементация позволява на играчът да сменя режимите си на стрелба само с един клавиш, тъй като при сигане на последния елемент на масива, следващият режим ще бъде първият в масива.

3.1.6 Презареждане

Оръжията имат безкраен брой пълнители - могат да бъдат презареждани безкрайно много пъти, стига играта да го позволява. Пълнителите обаче имат краен брой патрони, които могат да бъдат изстреляни. Това означава че при достигане на празен пълнител, играчът трябва да презареди. Презареждането използва таймер за да създаде период от време, през което играчът не може да стреля с оръжието си.

В началото на функцията се проверява дали таймерът за стреляне в

залпов (Burst) режим е активен. Ако е активен функцията приключва, защото играчът не може да презарежда по време на стрелба, а залповият режим не трябва да бъде прекъснат докато е активен. Ако обаче таймерът не е активен, функцията продължава, като извиква StopFire функцията, която спира стрелбата, ако играчът задържа спусъка (бутона за стрелба). След спиране не стрелбата, се извиква функцията на играча SetIsReloading, която поставя играчът в режим на презареждане и не му позволява да стреля. След поставяне на играча в режим на презареждане, се започва таймер, който ще извика същата функция върху играча, като се използва делегатът AllowFiringDelegate, на който като обект за изпълнение е подаден обектът на играча. Интервалът след който ще се извика делегатът се определя от променливата ReloadTime, която определя колко време играчът не може да стрела или по друг начин казано - колко дълго презарежда. Тази променлива има за начална стойност 1 секунда, но в редактора тази стойност може да бъде променена.

След започването на таймера текущия брой патрони в пълнителя се задават да са максималния брой възможен за оръжието. След приключване на таймера функцията SetIsReloading ще се извика отново и ще позволи на играчът отново да стреля.

3.1.7 Приближаване

Приближаването се случва, като ъгълът на полезрението на играча бъде променено към по-ниска стойност от 90 градуса - начлната стойност на полезрението на играча. Приключването на приближаването става като радиуса на полезрението се върне към 90 градуса.

Във функцията за приближаване се случват две неща - скриване на моделите, които играчът може да види и са част от него (оръжието и ръцете) и промяна на ъгълът на полезрението на играча. Това става с функциите HideFPMeshes и SetFOV, който са част от класа на играча. Като аргумент на SetFOV е подадена променливата ZoomFOV, която определя ъгълът на полезрението за приближаване. Стойността на тази поменлива се задава в редактора.

За да се върне играчът към първоначалното си състояние преди приближаването се случват обратните неща - показват се моделите,

които са били скрити от играча и ъгълът на полезрението се връща на 90 градуса.

3.2 AMultiplayerFPSCharacter

Класът дефинира играчът, от който всички други класове за играчите наследяват. В него се имплементират контролите, движението и това как играчът си взаимтодейства с нивата и другите обекти, компоненти и системи. В него се съдържат и чрез него се контролират оръжията, системата за кръв се закача за играчът чрез този клас и класът взаимодейства с предметът за регенериране на кръв, както и целите в режимите на игра.

3.2.1 Дефиниция на класа

Във файлът MultiplayerFPSCharacter.h е дефиниран класът AMultiplayerFPSCharacter. В него са дефинирани слотовета за оръжия, как системата за кръв и оръжията си взаимодействат с играчът, контролите на играчът.

Първоначално е дефиниран конструктурът на класа (подсекция 3.2.2), и след това са дефинирани сценовите компоненти на играчът. Той има два модела - един, който само играчът вижда, който представлява само ръце и цяло тяло, което всички останали играчи виждат (ред 17 и 20). След това е дефинирана камерата на играчът, през която то вижда светът (ред 23). Следващите два компонента са две кутии за колизия, които се използват за преценяване на коя част от играчът е ударена от оръжията. Те не се ползват за нищо друго от това да блокират лъчите от оръжията и да преценят в коя част на тялото играчът е ударен.

Системата за кръв се съдържа в играча като променлия, която се създава като подобект на класа и чрез нея се смятат щетите, които играчът поема.

Слотовета за оръжия са дефирнирани като два масива - един, който съдържа класа на оръжията (FirearmClassArray) и втори, който съдържа вече създадените оръжия (FirearmArray). Играчът също така има

променлива, която казва кое оръжие е в ръцета на играча, и с нея се обхожда масивът от вече създадените оръжия.

На ред 35 е дефиниран масив от булеви променливи, които казват на играчът дали може да стреля оръжията си. Тя се използва по време на презареждането, за да не може играчът да стреля оръжията си докато презарежда. Булевите променливи bIsReloading и bIsZoomedIn се отнасят съответно дали играчът презарежда оръжието или приближава с него.

Функцията SetupPlayerInputComponent се използва за добавяне на нови действия към контролите играчът. Това става чрез компонент наречен UInputComponent, който работи със стек от входове обработван от контролера на играчът (PlayerController).

На редове 79 и 81 са дефинирани функциите за BeginPlay на играчът и за неговата актуализация (tick). Tick функцията на играчът не се използва, тъй като играчът не се нуждае от функционалности, които се изпълняват на всеки кадър. В BeginPlay се взема името на играчът и се създават оръжията (подсекция 3.2.4).

Следващите три дефиниции (редове 84 87 и 90) се отнасят към интеракцията на системата за кръв на играчът. OnHealthChanged се извиква всеки път, когато системата за кръв промени животът на играчът, OnHealEvent е сигнатура на делегатът, чиято функция е да регенерира кръвта на играчът при припокриване с предмет за регенериране на кръв (секция 3.4.4). Функцията TakeDamage се извиква от оръжието, върху играчът, който оръжието е уцелило, и нейната функция е да предава щети към системата за кръв, за да може тя да ги обработи. На ред 100 е дефинирана функцията, с която играчът казва на системата за кръв че трябва да му се регенерира кръвта. Тази функция се извиква чрез делегатът на играчът - OnHealEvent.

Функциите от ред 121 (StartFiring) нататък се използват за контрол на играча върху оръжията. Те дефинират какво се случва, която играчът получи вход за работа с оръжието - дали да започне или спре да стреля, дали да презареди или да приближи. Функцията SetFOV се използва върхи камерата на играчът да промени полезрението на играчът (field of view) към такова с по-малък радиус, което кара погледът на играчът да приближи всички предмети и така се постига приближаването с оръжието.

HideFPSMeshes и ShowFPSMeshes са две функции, чиято работа е да скриват или показват моделите на оръжията и ръцете на играчът при приближаване. Това е нужно поради това, че когато радиуса на полезрението на играчът се променя към по-ниска стойност, моделите на ръцете и оръжията пречат на зрението на играчът и затова те се крият, като това е само визуално и трае само докато играчът е приближен.

3.2.2 Конструктор на класа

В конструктора на играчът се случват няколко неща. Първо се конфигурира движението на играчът и след това е редът на камерата, моделите и кутиите за колизия.

Първо се създава камерата с CreateDefaultSubobject шаблонът, като му се подава типа на камерата - UCameraComponent и му се задава името "FirstPersonCamera". След това тя се закача за капсулния компонент на играчът, като той се взема чрез функцията GetCapsuleComponent. За всички класове, които наследяват от ACharacter класът капсулния компонент е основния компонент за обектът. След това и се задава релативна позиция спрямо капсулата, така че, когато се добавят моделите, всички се намират в капсулата на играчът, и не излизат от нея. Това обикновенно е нужно, тъй като капсулата на играчът блокира повечето геометрични обекти и с нея се смята къде играчът да се сблъсква в обектите. Ако има обекти, които излизат от капсулата на играчът, те ще навлизат в други обекти, като например стени и това не изглежда добре по време на игра. Чрез променливата bUsePawnControlMovement (ред 14) се казва на камерата да използва ротацията на играчът като своята.

След камерата се инициализира моделът за ръцете на играчът. Създава се със шаблонът CreateDefaultSubobject и неговия клас - USkeletalMeshComponent и му се задава името "FirstPersonMesh". След създаването на моделът той се задата така че само играчът да го вижда. Тъй като другите играчи не трябва да виждат моделът за ръцете на играчът, Unreal Engine предоставя възможността моделът да бъде скрит от всички освен играчът. След това той се закача като дете на камерата в йерархията на компоненти. Сенките на моделът се скриват (редове 19 и 20), тъй като моделът е само ръце и ако играчът виж-

да сенките на само ръцете си а не цялото тяло ще е странно и затова те просто не се генерират. Релативните координати на моделът (позиция и ротация) се задават спрямо камерата, така че позицията им да изглежда добре когато играчът гледа през камерата.

Инициализацията на моделът за цялото тяло на играчът става по подобен начин - създава се с CreateDefaultSubobject, като типът на компонента е същия като този на моделът за ръцете. Името на компонентът се задава като "FullBodyMesh". На ред 25 моделът се задава така че играчът (собственикът на моделът) да не го вижда. След това се задава сенките му да се генерират, играчът няма да ги вижда, тъй като моделът е скрит от него, което скрива и сенките. Това означава че от камерета, играчът не вижда никакви сенки отнасящи се до неговите модели.

След инициализацията на моделите се създава и системата за кръв като подобект на играча и ѝ се задава името "HealthSystem". На ред 31 към делегатът, който съществува в системата за кръв и се отнася за поемането на щета се добавя функцията като се използва макрото AddDynamic. Функцията която ще се извиква при промяна в кръвта на играчът е OnHealthChanged.

Към делегатът, който се отнася за регенерирането на кръвта на играчът OnHealEvent се закача функцията Heal, която ще се извика всеки път, когато играчът регенерира кръвта си.

Кутиите за колизия отнасящи се за местата, където оръжието може да уцели играчът се инициализират с CreateDefaultSubobject, като на кутията за тялото (BodyHitboxBox) се задава името "BodyHitboxBox"а на кутията за главата (HeadHitboxBox) - "HeadHitboxBox". След инициализацията се задават размерите на кутиите, така че те да съвпадат с размерите на моделът за цялото тяло, тъй като този модел другите играчи ще виждат и ще стрелят по него. След оразмеряването на кутиите им се задава позицията, така че да съвпада с поциите на частите на тялото на моделът за цялото тяло. Колизията им се задава като QueryAndPhysics, което означава че те ще могат да правят и получават и пространствени, и физически заявки.

След инициализирането на компонентите на играча се задават стойности на няколко от променливите. Първо се задава, че този обект може да се актуализира (има tick). След това той се задава като ре-

пилиращ се обект. Променливата за това кое оръжие е в ръцете на играчът (кое оръжие той може да стреля) се задава като 0 - първото в масива и след това булевите променливи относно състоянието на играча (bIsReloading и bIsZoomedIn) се задават като false.

С това приключва конструкцията на един играч при създаването му.

3.2.3 BeginPlay

В BeginPlay на играчът се случват две неща - задава се името му (PlayerName) и се създават оръжията. Оръжията се създават в BeginPlay за да е сигурно че вече всички компоненти на играчът са инициализирани и тогава да се зъдават новите обекти.

3.2.4 Създаване на оръжията

Функцията създава оръжията като обхожда масивът от класовете на оръжия, създава ги и пълни масива от вече създадени оръжия с тях.

Оръжията се създават чрез света - UWorld и чрез неговата функция SpawnActor. Тази функция приема класът на обектът, който ще бъде създаден - AMultiplayerFPSFirearm. Този клас е взет от масивът от класове на оръжията FirearmClassArray, който се пълни в редактора и в него се задавт какви ще са оръжията. Тъй като по дизайн оръжията са само един клас и нямат подкласове, класът, който ще се съдържа в масивът ще е винаги AMultiplayerFPSFirearm.

Другите параметри, които функцията приема са позицията и ротацията на обектът, които ще му бъдат зададени при инициализация. Последният параметър на функцията са аргументите, с които обекът ще бъде създаден. Един от тези аргументи е кой е обектът, който е притежателя на създадения обект. Като притежател е зададен играчът. Другият аргумент, който е зададен е при какви обстоятелства обектът се създава. В този случай е AdjustIfPossibleButAlwaysSpawn, което означава че функцията за създаване ще потърси най-близкото място където обектът няма колизия с нищо, но ще го създъда винаги, дори когато не може да намери такова място.

След извикаването на функцията SpawnActor, обектът, който тя

създава се записва във временна променлива, която се проверява дали е валидна. Ако не е валидна - имало е грешка при създаването на обекта, програмата изхвърля грешка и приключва. Ако обектът е валиден, прави му се cast към AMultiplayerFPSFirearm и резултатът се записва във втора временна променлива. След това се проверява дали този cast е валиден по същия начин и функцията приключва ако не е валиден. Ако всичко е наред, вече превърната променлива се записва в масивът от създадени оръжия. С това итерацията на цикълът приключва и този алгоритъм се повтаря за всички елементи в масива от класове на оръжия.

След приключване на цикъла, оръжията, се закачат за моделът на ръцете, като първото оръжие се закача за сокетът на моделът, където ръцете ще го държат насочено, а второто се закача за гърба на моделът.

На края на функцията булевият масив, който казва дали играчът може да стреля оръжята си, се запълва със стойността true, като размерът на масивът е същият като размерът на масива от създадени оръжия.

3.3 UMultiplayerFPSHealthSystem

Този клас дефинира системата за кръв, която наследява от UActorComponent, тъй като не се нуждае от физическо изобразяване. Компонентът е част от AMultiplayerFPSCharacter и се инициализира в неговия конструктор (виж секция 3.2). В системата за кръв са дефинирани два вида променливи - щит (Shield) и кръв или живот (Health). Щитът може да се регенерира, като това става чрез използването на таймери, а кръвта се променя само при поета щета или ако играчът използва предмет за регенериране.

3.3.1 Дефиниция на класа

В началото на дефиницията на класа е дефиниран динамичен multicast делгат, чиято функция е да известява играча, че кръвта му е променена. В тялото на класа са дефинирани променливите за максимална кръв и максимален щит, променливата, с която се определя с колко се запълва щита на всяко извикване на функцията

за регенериране на щита (ред 23). След нея е дефинирано времето, което системата за кръв трябва да изчака за да започне регенерирането на щита - ShieldRechargeCooldownAfterDamage (ред 26). ShieldRechargeInterval определя през какво време функцията за регенерация на щита RechargeShield се вика. Стойността, с която се променя променливата за запълване на щита е ShieldRechargeRateIncreaseValue, която увеличава ShieldRechargeRate през определен брой извиквания на RechargeShield.

Променливите дефинирани като частни са CurrentHealth и CurrentShield, които съдържат текущите стойности на кръвта и щита на играча. Тези стойности се репликират за не могат играчите сами да си променят стойностите на кръвта и щита.

Структурите от тип FTimerHandle също са дефинирани като части, като те се използват при работа с таймерите за изчакване на регенериране и регенериране на щита на играча. ShieldRateCounter се използва за броене колко пъти RechargeShield функцията е извикана и след определен брой извиквания ShieldRechargeRate се увеличава със стойността на ShieldRechargeRateIncreaseValue.

Функцията TickComponent не е нужна на компонента тъй като той не се уждае от актуализация. RechargeShield функцията се извиква докато щита не се регенерира напълно, като нейната функция е да запълва щита постепенно. StartShieldRecharge започва регенерацията на щита, като задава цикличен таймер да извиква функцията RechargeShield, докато щита не стигне максималната си стойност.

TakeDamage функцията отговаря за поемането на щета причинена върху играча. Тя се закача за един от динамичните делегати на играча - OnTakeAnyDamage.

Следвачите четири функции дефинират getter-и за максималните и текущите стойности на кръвта и щита на играча.

Функцията Heal отговаря за възстановяването на кръвта.

3.3.2 Конструктор на класа

В конструктора на системата за кръв се задава че компонентът не се актуализира (няма tick), след което се инициализират максималните и текущите стойности на кръвта и щита, както и променливите отгова-

рящи за регенерацията на щита.

3.3.3 BeginPlay

В BeginPlay на системата за кръв се случват две неща. Пръво се задават стойностите на текущите щит и кръв на играчът. Тъй като целта на класа е максималните стойности да се променят в редактора, текущите стойности не могат да приемат стойности при конструкция. След задаване на стойностите на текущите статуси, към делегатът на Actor класовете, който отговаря за поемане на щета се прикрепя функцията TakeDamage на системата за кръв, която отговаря за поемането на щета и следенето на статусите на играча.

3.3.4 Поемане на щета

Поемането на щета става чрез делегатът на AActor класа от който играчът наследява своя клас. Функцията ще се извика всеки път, когато върху играча е извикана TakeDamage функцията (секция 3.1.4).

След извикване на функцията първоначално се взема притежателя на този компонент - играчът и се проверява дали е валиден. Ако не - изхвърля грешка и приключва изпълнението. След вземането на играча се създава нов делгат за таймера, към който ще се прикрепи функцията StartShieldRecharge. Това става с BindUFunction шаблонът, който приема обекът, към който принадлежи функцията и името на функцията, която да се прикрепи към делегатът.

След прикрепяне на StartShieldRecharge към делегатът за таймера се стартират таймерите. Таймерите са нужни, защото когато играчът поеме щета, неговият щит не трябва да започне да се регененрира веднага, а трябва да измине период от време и това става с таймерите на Unreal Engine. Ако играчът поеме щета, таймерът, който изчаква извикването на StartShieldRecharge се нулира и започва от начало. Това става като първо се провери дали таймерът е активен с функцията IsTimerActive, на която се подава handle обектът който отговаря за таймера. Ако IsTimerActive върне true, таймерът се нулира, което се случва с функцията ClearTimer на която се дава съответният FTimerHandle обект. След нулиране на таймера той

се стартира от начало със SetTimer функцията, която приема съответната FTimerHandle структура, делегатът (ShieldRechargeDelegate), който държи функцията, която ще бъде извикана, когато таймерът изчака желаното време, времето което таймерът трябва да изчака (ShieldRechargeCooldownAfterDamge) и дали таймерът се повтаря или не, като в този случай е не (false). Ако таймерът не е активен, той се стартира, като аргументите са същите като тези, които се подават в горния блок от код.

След задаването на таймера за изчакване на регенериране на щита, се проверява дали щита вече се регенерира. Ако той вече се регенерира, се спира, защото ако играчът поеме щета, щитът не трябва да продължи да се възстановява. Ако не е активен, нищо не се случва.

Поемането на щета става като първо се провери състоянието на щита - ако е 0, кръвта на играча поема щета, но ако не щитът намалява докато не стигне 0 и след това е редът на кръвта. Щетите не се пренасят от щит към кръв - ако играчът има 8 щит и поеме 10 щета, щитът свава 0, но кръвта не се променя и разликата между поетата щета и щита се губи. Промяната на стойностите става с функцията Clamp, която приема израз, който връща стойност, минимална стойност и и максимална стойност. Ако резултатът от изразът излезе от тези граници, в зависимост от посоката - дали е по-голям от максималната или по-малък от минималната, като изход на функцията се дават съответната максимална или минимална стойност и резултатът не излиза от границите. Това се използва, за да не се случи, както в по-горния пример щитът да стане -2 а не 0.

На края на функцията се извиква Broadcast на делегата отговарящ за промяната на кръв на играча за да се каже на всички играчи, че дадения играч е поел щета.

3.3.5 Стартиране на регенерация на щита

Както повечето функционалности отнасящи се до щита на играча, тази функция използва таймери за да регенерира щита на играча. Това става като първо се изчисти таймерът за забавяне на регенерация на щита, което е нужно, защото предметът за възстановяване на кръвта (секция 3.4) също започва и регенерацията на щита веднага след

интеракция с него. Тъй като предметът няма достъп до вътрешните таймери на системата за кръв, това може да стане само ако системата сами си чисти таймера. Изчистването на таймера във функцията не е проблем, защото ако тя бъде извикана от таймера за започване на регенерация (секция 3.3.4), това означава че таймерът вече е приключил и изчистването му е позволено и ClearTimer не връща грешка ако таймерът, който се изчиства не е активен.

След изчистването на таймера за започване, се стартира нов таймер, който отговаря за регенерирането на щита. Този таймер се повтаря - работи циклично и се извикава постоянно през определено време. Това време е зададено в променливата ShieldRechargeInterval, чиято стойност се задава в редактора от дизайнерите на играта.

3.3.6 Регенериране на щита

Регенерирането на щита става с таймер, който се извиква на определено време и на всяко извикване, функцията добавя стойност към щита за да го увеличи. Тази стойност се дефинира в променливата ShieldRechargeRate. Системата за кръв има и друга променлива - ShieldRechargeRateIncreaseValue, която след определен брой извиквания се добавя към променливата за запълване на щита и така ускорява регенериране на щита.

След промяна на стойноста за регенериране тя се добавя към щита чрез функцията Clap за да не излиза извън границите, които са 0 и максималната стойност на щита (MaxShield). След промяната на щита, се проверява дали щита е стигнал максималната си стойност и ако да - таймерът за ренериране на щита се изчиства и функцията приключва. Ако все още щита не се е регенерирал напълно, таймерът за ренегериране продължава да извиква функцията докато не бъде спрян.

3.3.7 Регенериране на кръвта

sec:cpphealthsystem:healthregen

Възстановяването на кръвта става, когато играчът използва предмет за възстановяване и тази функция от класа на играча чрез делегата за регенерация на кръв (OnHealEvent).

Функцията регенерира кръвта, като проверява дали подадента стойност е 100.0f и ако е, направо казва кръвта на играчът да е максималната стойност на кръвта (MaxHealth). Ако не е 100.0f, тогава чрез Clamp функцията тази стойност се добавя към кръвта на играча. Това е добавено да позволява на играча да получава кръв от други места и не само от предметите за регенерация.

3.4 AHealthPickup

Този клас наследява AActor като дефинира предмет, който може да бъде поставен на картата и може да бъде взет от играча. Той се състои от два компонента - колизионна кутия (BoxCollision) и модел (HealthPickupMesh). За основен компонент е избран моделът, като кутията е прикрепна към основният компонент, и позицията на кутията се смята спрямо моделът.

3.4.1 Дефиниция на класа

В дефиницията на класа са дефинирани пет функции и един конструктор. В конструктора се инизиализират компонентите, BeginPlay се изпълнява след започване на tick на нивото, Tick се изпълнява на всеки кадър (frame) за обекта, OnBeginOverlap се изпълнява след като кутията за колизия се пресече с друг компонент, и Destroy функциите се викат съответно на сървъра и клиента.

3.4.2 Конструктор на класа

В конструктура първо се инициализира моделът (HealthPickupMesh), като се създава с CreateDefaultSubobject с типа на моделът (UStaticMeshComponent) и като параметър му се подава името на компонентът, който ще бъде създаден, в този случай "SM_HealthPickup". CreateDefaultSubobject е шаблон, който създава нов компонент или събобект, като извиква функцията от класа FObjectInitializer със същото име, която инициализира всички UObject типове.

След инициализацията и създаването на моделът, той се задава като главния компонент.

Кутията за колизия се инициализира по същия начин, по който и моделът - с шаблонът CreateDefaultSubobject, като му се задава името "BoxCollision". След това се задава размерът на кутията и се закача за главния компонент.След това се задава колизията на кутията като QueryOnly - проверките са само пространствени (raycast, sweep, overlap).

3.4.3 BeginPlay

В BeginPlay се извиква макро за извикване на AddDynamic() върху динамични делегати, като съответният делегат е OnComponentBeginOverlap. Функцията която се асоциира с този делегат е OnBeginOverlap, която ще се извиква всеки път когато кутията за колизия започне пресичане с друг компонент.

3.4.4 При припокриване с играч

След извикване на OnBeginOverlap, се проверява дали Actor обектът с който кутията се е пресякла има етикет (tag) "Player". Ако да - функцията продължава, а ако не - изхвърля съобщение в лога на Unreal Editor и излиза.

След проверката се прави cast на OtherActor към AMultiplayerFPSCharacter и се проверява дали този cast е успешен (дали не е върнал nullptr) и ако не е - се изхвърля грешка и функцията приключва. След това се проверява дали кръвта на играчът е на 100% и ако е се излиза от функцията. След провекрата се извиква Broadcast на делегата отнасящ се за регенерирането на кръвта на играча, намиращ се в класа на играча, като му се задава стойност 100.0f, която казва на функцията за регенериране да запълни кръвта на играча (описано в секция 3.2).

Глава 4

Ръководство на потребителя

4.1 Инсталиране на проекта

4.1.1 Необходими приложения и технологии за пускане на проекта

За подкарването на прокета са нужни Unreal Engine и Unreal Editor. Те могат да бъдат инсталирани като се следват инструкциите в документацията на Unreal Engine. Също така е препоръчително да имате инсталиран Visual Studio 2017, 2019 или 2022 Community. Той може да бъде инсталиран като се следват инструкциите на уеб страницата на Visual Studio. Проектът работи най-добре под Windows 10, но е възможно и да бъде подкаран на MacOS. Подкарването на Unreal Engine и Unreal Editor на Linux е възможно и е описано в документацията на Unreal Engine, както и community документацията на Unreal Engine.

Също така са необходими git и git LFS (Large File Storage) за клониране на директорията на приложението.

4.1.2 Клониране на проекта от GitHub

Проложението може да бъде свалено от GitHub хранилището което се намира на github.com/Yasen-Alchev/MultiplayerFPS_DiplomnaRabota. Преди да можете да клонирате проекта ще е препо-

ръчително да проверите дали имате git LFS. Това става чрез следната команда:

```
$ git lfs version
```

```
git-lfs/3.0.2 (GitHub; windows amd64; go 1.17.2)
```

Фигура 4.1: Примерен изход на командата "git lfs version".

Програмата трябва да покаже нещо подобно на фиг. 4.1.

След потвърждение за инсталирана версия на git LFS, той трябва да бъде инсталиран в папката където смятате да свалите GitHub хранилището. Това става със следната команда:

```
$ git lfs install
```

```
Git LFS initialized.
```

Фигура 4.2: Изход на командата "git lfs install".

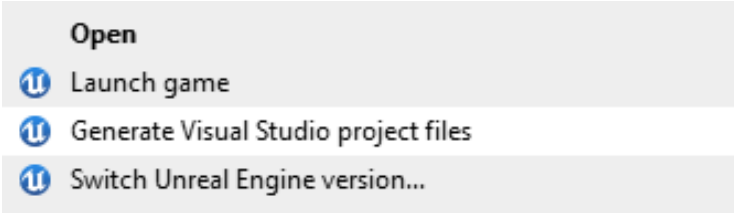
Изходът на тази команда трябва да изглежда така както е на фиг. 4.2. След изпълнение на тези две команди сте готови да изтеглите проекта от GitHub хранилището. Това става със следната команда:

```
$ git clone https://github.com/Yasen-Alchev/MultiplayerFPS_DiplomnaRabota.git
```

След изтегляне на проекта ще намерите папка с името "MultiplayerFPS_DimplomnaRabota"в директорията, в която сте клонирали хранилището. В тази папка ще намерите файловете нужни за работа с приложението.

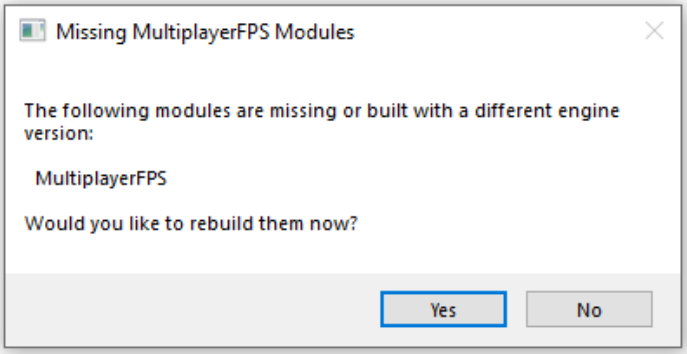
4.1.3 Първо стартиране на проекта

Преди да започенете работа с проекта, има няколко неща които трябва да бъдат изпълненти.

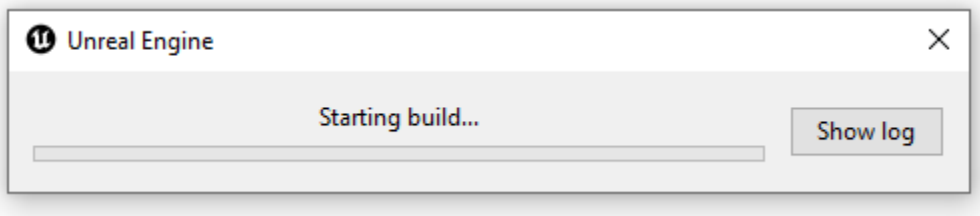


Фигура 4.3: Генериране на файлове нужни на Visual Studio.

Първо трябва да се генерират Visual Studio файлове. Това става чрез натискане на десен бутон върху .uproject файлът в директорията на проекта (фиг. 4.3). Това ще Ви позволи да отворите кода на проекта във Visual Studio.



Фигура 4.4: Прозорец, който пида дали да направи rebuild на MultiplayerFPS.

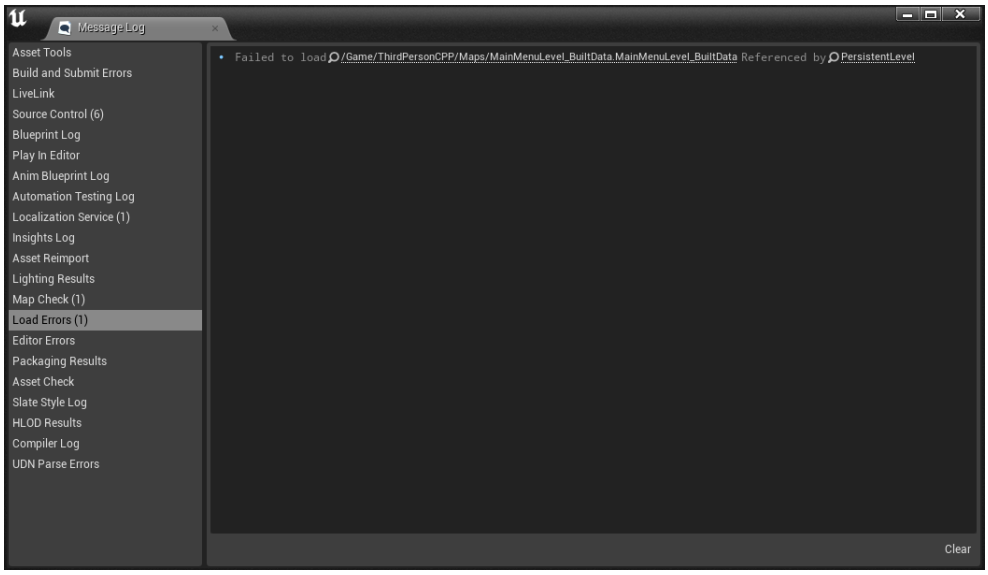


Фигура 4.5: Прозорецът, показващ до къде е стигнал rebuild-ът.

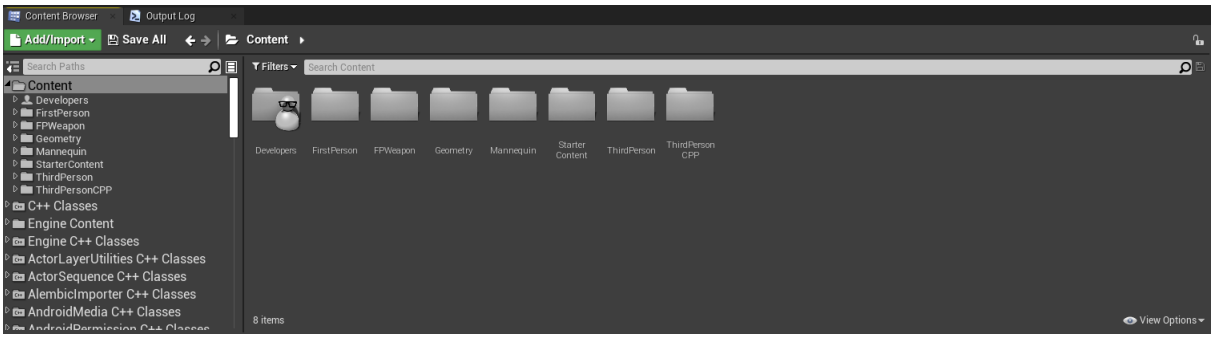
След това трябва да се изпълни файлът с разширение .uproject. След стартирането му ще се появи прозорец (фиг. 4.4), който ще Ви пита дали искате да направите rebuild на MultiplayerFPS модула. Натиснете да (Yes). След това ще се появи нов прозорец, който ще показва до къде този rebuild е стигнал (фиг. 4.5). След приключване на rebuild-ът, Unreal Editor ще се стартира. Преди да можете да започнете работа с проектът, ще е нужно да направите още няколко неща.

Генериране на BuildData за нивата

Преди да генерирате BuildData за нивата, ще е хубаво да се запознаете с интерфейса редактора - Unreal Editor. Това може да стане със статията в документацията на Unreal Engine относно инерфейса на редактора. След това ще знаете кое къде се намира и ще можете да навигирате редакторът по-лесно.

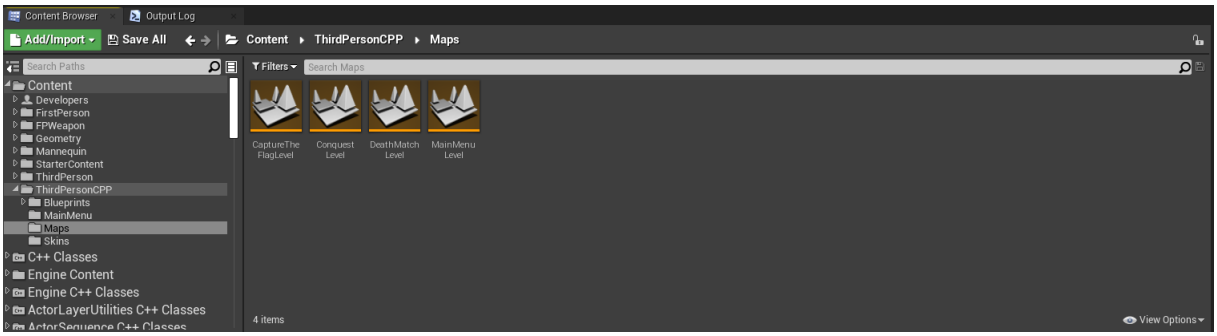


Фигура 4.6: Грешка относно зареждането на BuildData за началното ниво.



Фигура 4.7: Началото на Content папката.

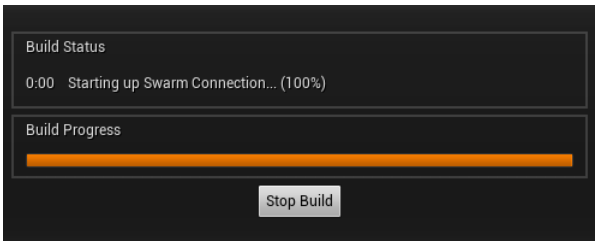
Фигура 4.8: Навлизане в ThirdPersonCPP папката.



Фигура 4.9: Навлизане в Maps папката.

На фиг. 4.6 е показана грешката, която ще видите когато стартирате проекта в редактора. Това се случва, защото git игнорира _BuildData.uasset файловете. Те могат да бъдат генерирани лесно и не отнема много време. Първо трябва да навигирате в папката където се намират нивата, както е показано на фиг. 4.7, 4.8, 4.9.

Фигура 4.10: Лентата с инструменти.



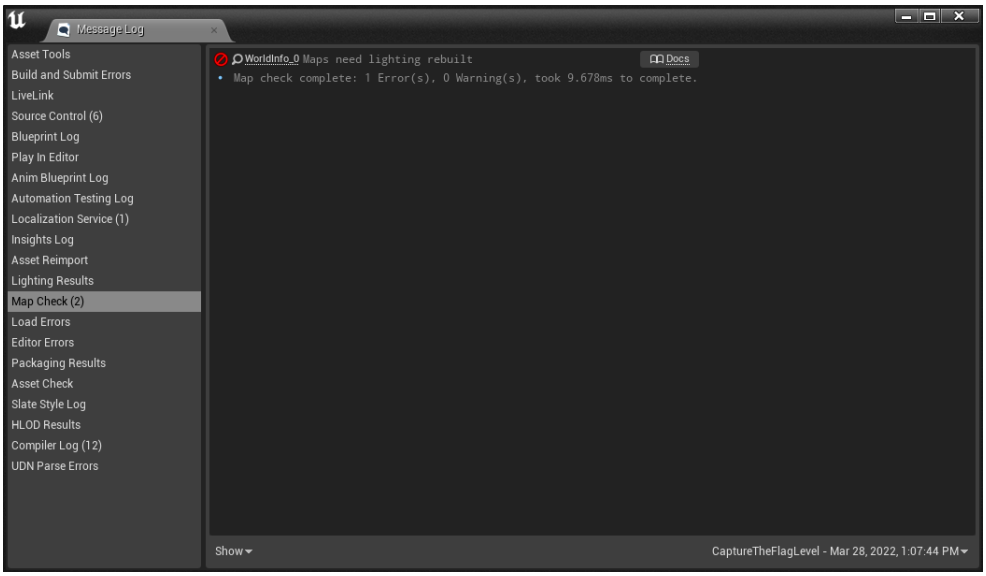
Фигура 4.11: Стартиране на генерирането на BuildData за нивото MainMenuLevel.

Фигура 4.12: Генериран BuildData за MainMenuLevel нивото.

Тъй като MainMenuLevel се зарежда стандартно при стартиране на проекта в редактора, е най-добре да започнете с него. За да се генерира BuildData за това ниво, трябва да му се направи build. Това става с бутна "Build"на лентата с инструменти (фиг. 4.10), след което ще се появи прозорец, показващ началото на генерирането (фиг. 4.11). След изчакване на генерирането да приключи, натиснете Ctrl + Shift + S за да запазите промените на всичко. След запазване в папката на нивата ще се появи файл с името "MainMenuLevel_BuildData". Това означава, че генерирането на BuildData за това ниво е готово (фиг. 4.12).

След това можете да преминете на следващото. Това става с две натискания върху нивото, което искате да заредите.

Фигура 4.13: Грешка относно генерирането на светлината на нивото CaptureTheFlagLevel.



Фигура 4.14: Грешка по време на build на нивото относно светлината.

На фиг. 4.13 можете да видите грешка, която ще Ви се появи когато заредите някое от другите нива, като в този пример това ниво е CaptureTheFlagLevel. За да оправите тази грешка и да генерирате BuildData за това ниво, трябва да направите build на нивото. Това става по същия начин както и с MainMenuLevel. Генерирането на BuildData за това ниво, както и за другите би трябвало да отнеме повече време, поради това че се и генерира информация за светлината на нивото. По време на генерирането ще се появи прозорец, който показва грешка, но той може да бъде игнориран, тъй като тази грешка ще се оправи в процеса на генериране на BuildData и светлината на нивото. След като генерирането приключи запазете промените с Ctrl + Shift + S и ще видите генерираната BuildData за нивото с името {името на нивото}_BuildData.

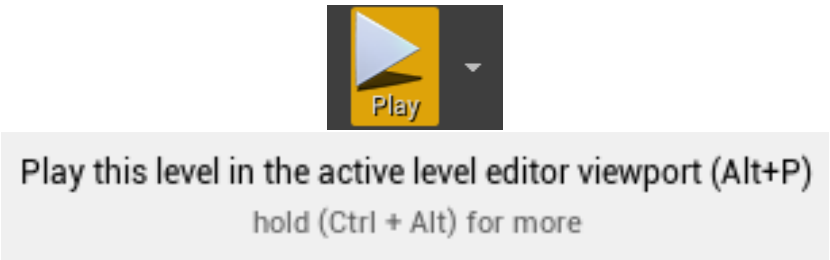
Повторете за всички останали нива.

Фигура 4.15: Финален резултат от генерирането на BuildData за всички нива.

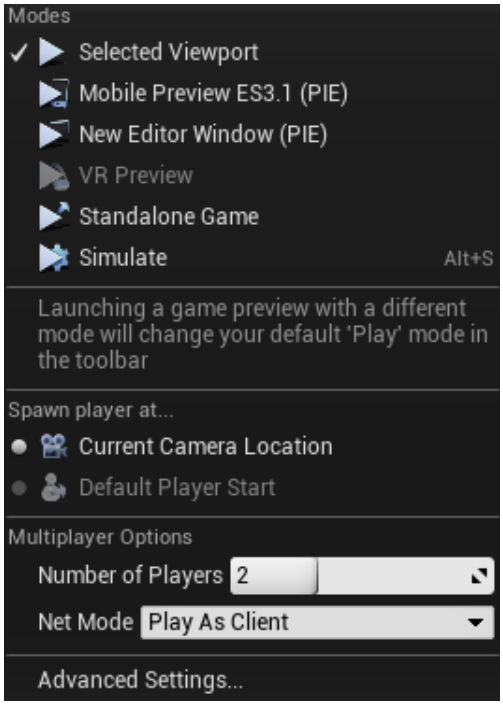
След завършване на генерирането сте готови да работите с проекта.

4.2 Стартиране на проекта

Проектът може да бъде стартиран по два начина. Той може да бъде отворен в прозореца на редактора или като пакетиран продукт.

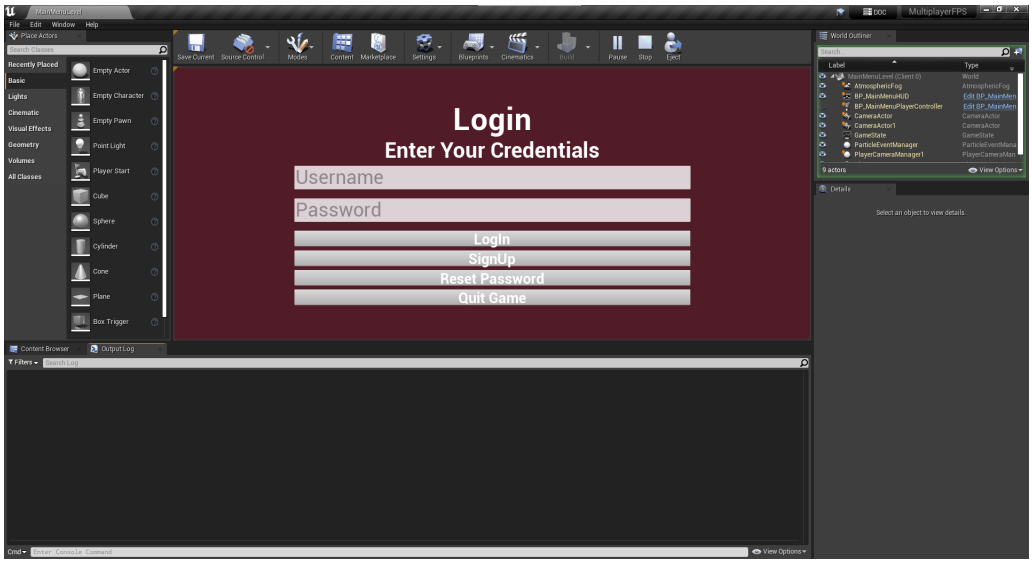


Фигура 4.16: Бутонът за стартиране на проект от Unreal Editor.

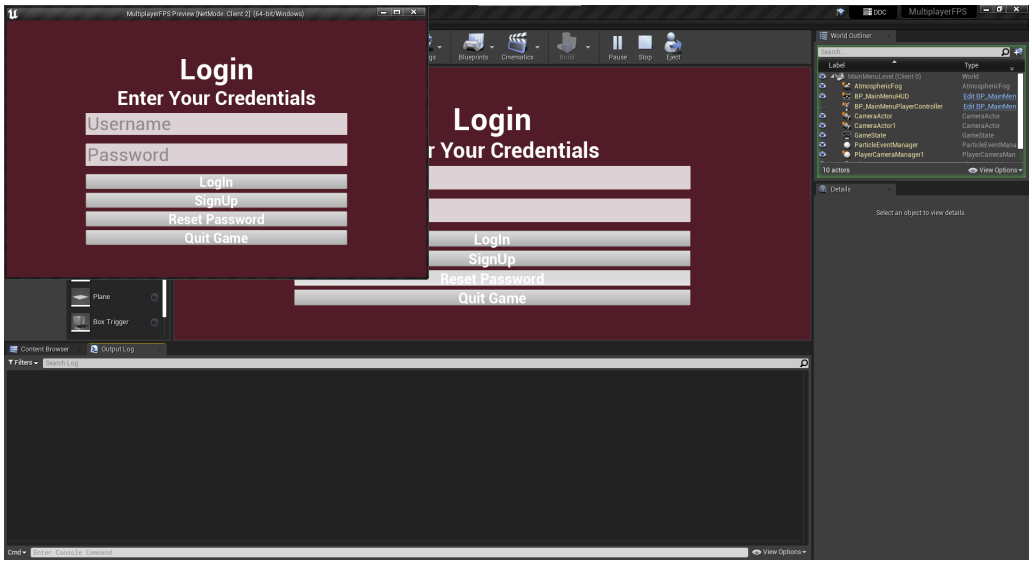


Фигура 4.17: Допълнителни опции за стартиране на проекта в редактора.

Стартирането в редактора става чрез натискане на бутона "Play"(фиг. 4.16) в лентата с инструменти (фиг. 4.10). Това ще стартира играта в прозореца за изглед. Ако сте избрали повече от един играч от допълнителни опции (фиг. 4.17), останалите прозорци ще бъдат отворени като отделни прозорци (фиг. 4.18, 4.19).

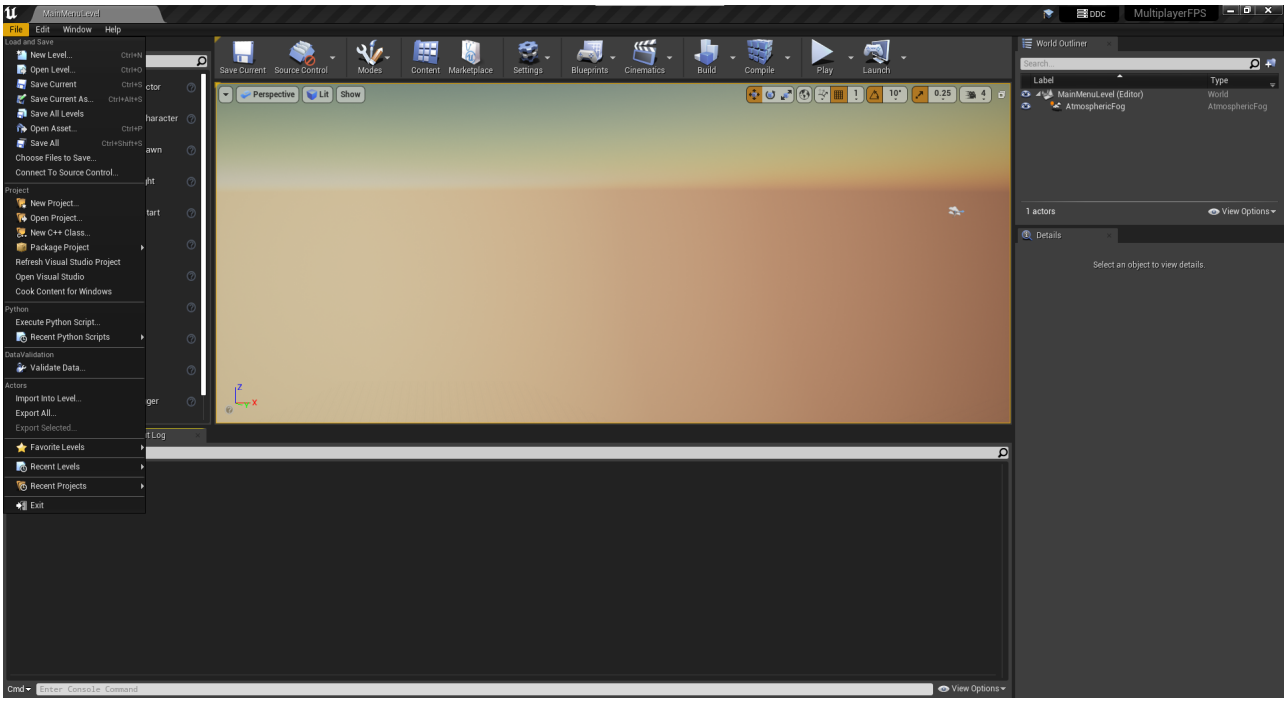


Фигура 4.18: Играта, отворена в редактора с един играч.

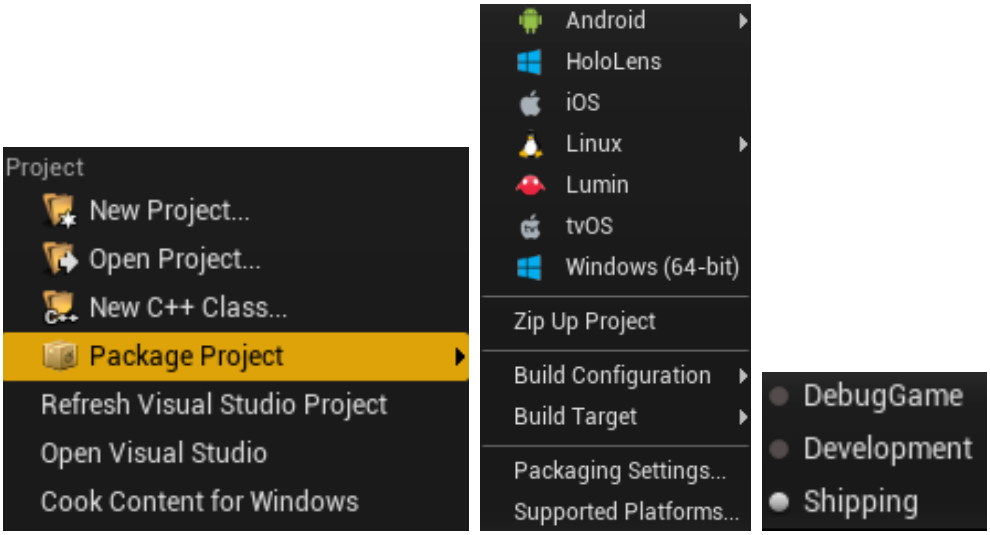


Фигура 4.19: Играта, отворена в редактора с два играча.

Другият начин за стартиране на играта е чрез пакетирана версия на проекта. Проектът може да бъде пакетиран през редактора, чрез опцията "Package Project"(4.21), която се намира във File опциите в горния ляв ъгъл на редактора (4.20).



Фигура 4.20: Къде опцията за пакетиране се намира

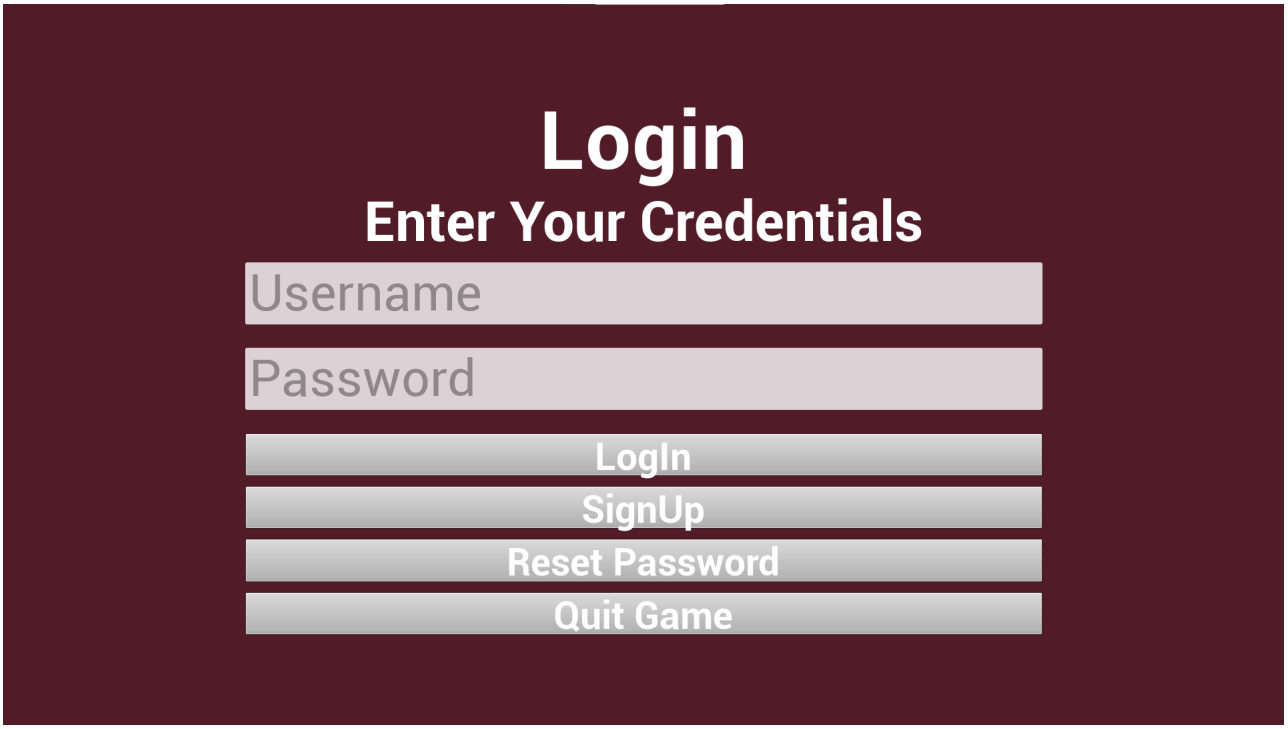


Фигура 4.21: Опции за пакетиране на проекта.

За да пакетирате проектът за Windows операционната система, изберете опцията "Windows (64-bit) което ще стартира пакетирането. Ако пакетирате за някоя друга операционна система изберете нея от менюто. Преди обаче да пакетирате проекта, първо трябва да се избере

как ще бъде пакетиран. Показано на фиг. 4.21 на фигурата в дясно, изберете от "Build Configuration"конфигурация за Shipping, което ще пакетира проектът като за финализиран продукт. След започване на пакетирането, редакторът ще Ви попита в коя папка искате да пакетирате проекта. Изберете папката така както на Вас ще Ви е удобно.

След завършване на пакетирането, в директорята която се избрали ще се намира папка WindowsNoEditor или с името на операционната която се избрали. В папката ще има .exe файл, който ще се казва MultiplayerFPS.exe. След стартиране на този файл, играта ще започна на пълен прозорец (фиг. 4.22).

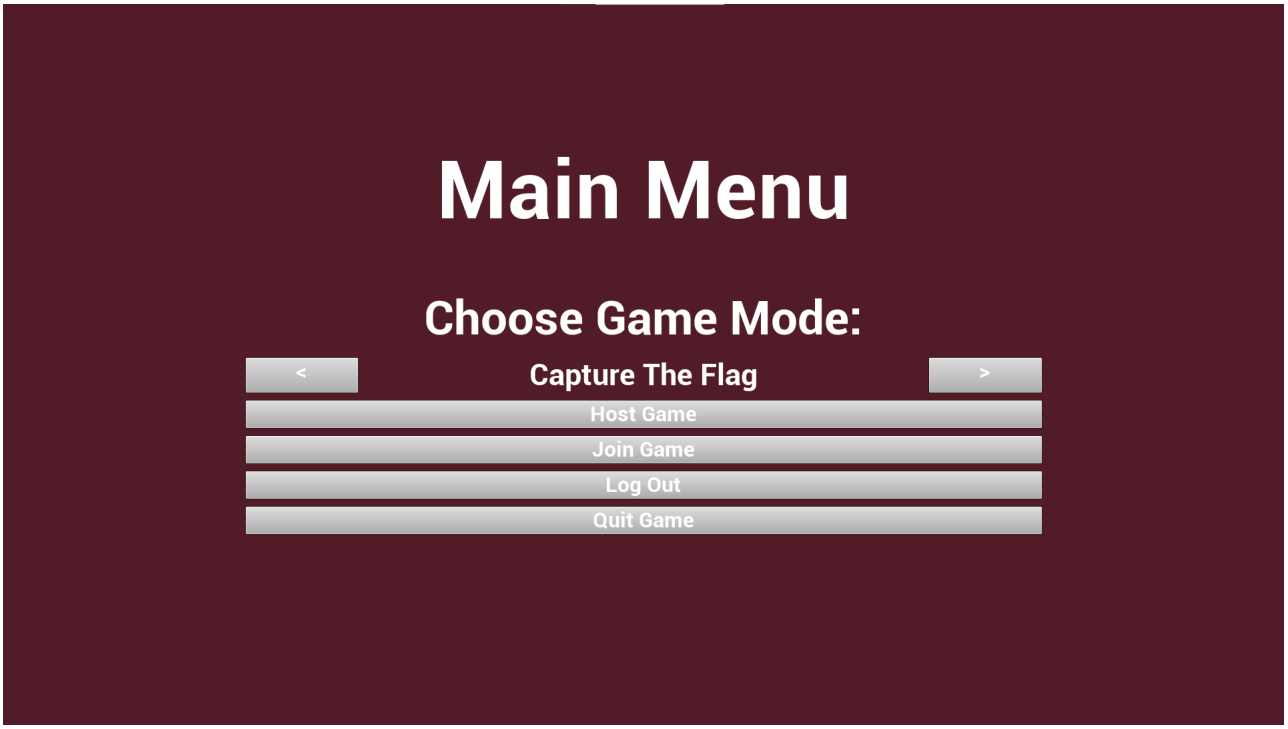


Фигура 4.22: Играта, отворена от пакетирания проект.

4.3 Игране на играта

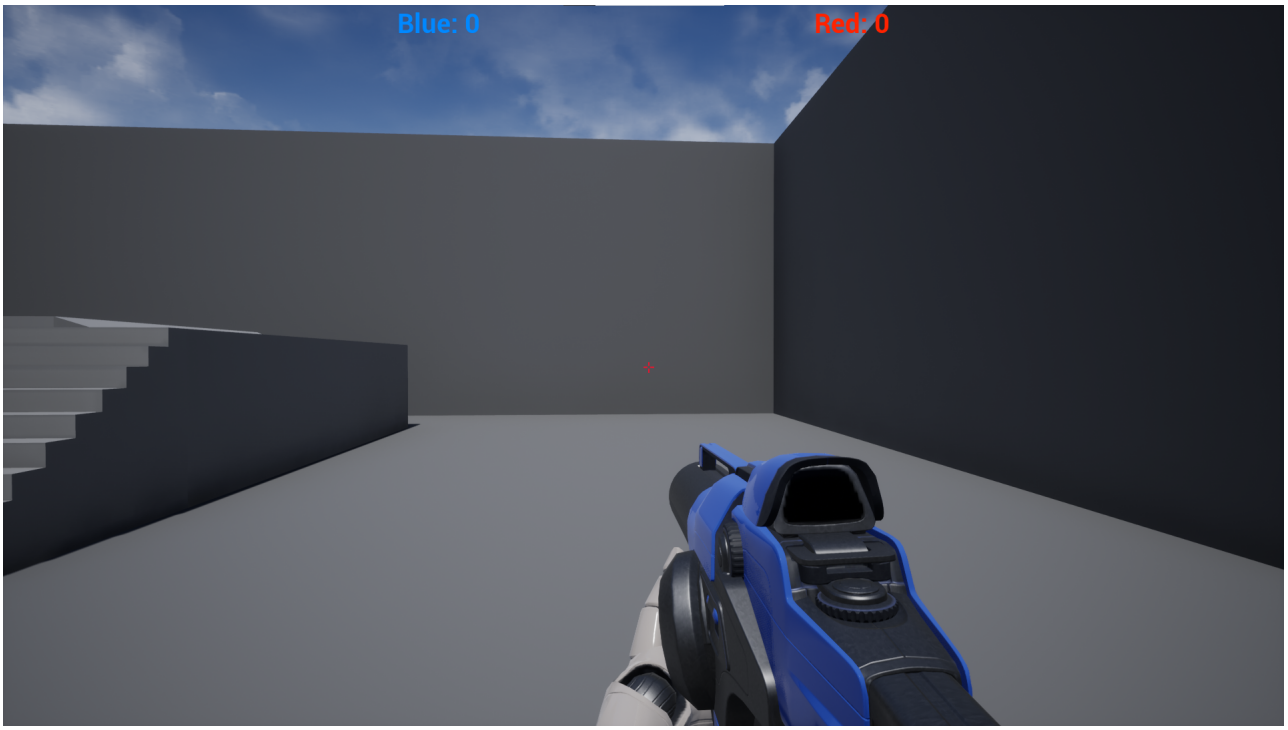
След стартиране на играта ще видите екран, който ще ви пита за потребителско име и парола, както на фиг. 4.22. Като потребителско име вече има регистрирано "AAA"и парола "123123". Можете да влезете с бутона "LogIn". След успешно влизане ще трябва да изебрете режим на игра, който се избира най-горе на екрана със стрелките - фиг. 4.23.

След избран режим можете или Вие да сте хост на играта чрез бутона "Host Game"или да се присъедините чрез бутона "Join Game". Ако изберете да се присъедините, ще трябва да въведете IP адресът на хоста на играта, към която искате да се включите.



Фигура 4.23: Екранът за избиране на режим на игра.

След започване или присъединяване към игра, ще видите Вашия играч, държащ оръжие в ръцете си и мерник в центъра на екрана Ви (фиг. 4.24).



Фигура 4.24: Погледът на играча след започване на игра.

Можете да контролирате играча си с клавиатурата и мишката, тъй като играта е направена за настолен компютър. Как се контролира играчът можете на видите в табл. 4.1. Там са описани всички клавиши нужни за да операрате Вашия герой, какво правят и коя функция извикват в имплементацията на играча.

W, A, S, D	Движение в посоките напред, наляво, назад и надясно
Shift	Тичане.
LMB (Left Mouse Button)	Стрелба с оръжието в ръцете на играча.
RMB (Right Mouse Button)	Приближаване с оръжието в ръцете на играча.
R	Презареждане на оръжието в ръцете на играча.
X	Смяна на режима на стрелба (ако е възможно).
Q	Смяна с другото оръжие на разположение на играча

Таблица 4.1: Таблица с клавишите за контролиране на играча.

Разлчните режими на игра позволяват различни преживявания и също така имат различни цели. Ако искате можете само да стреляте и убивате другите играчи, можете да пазите целите, или можете да

пречите на другите играчи да спечелят. Всичко останало зависи какво Ви се прави в играта. Дадени са Ви инструментите, какво ще правите с тях зависи от Вас.

Глава 5

Заключение

Дипломната работа може да не отговаря на всички изисквания - липсва изкуствения интелект, но това което е направено е до някаква степен опитано да бъде качествено написано.

Оръжията, системата за кръв и слотовете за оръжия са изпъленени и оръжията даже са разработени извън функционалните изисквания, но към тях винаги ще има какво да се добави.

Липсата на изкуствения интелект сваля стойността на диплномната работа, но работата свършена върху другите системи не е малка и не бива да бъде засенчена от липсващия изкуствен интелект.

Библиография

[1] Epic Games Inc. *AActor*. URL: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Actors/.

[2] Epic Games Inc. *Character*. URL: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Framework/Pawn/Character/.

[3] Epic Games Inc. *Components*. URL: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Actors/Components/.

[4] Epic Games Inc. *Delegates*. URL: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Delegates/.

[5] Epic Games Inc. *FHitResult*. URL: https://docs.unrealengine.com/4.27/en-US/API/Runtime/Engine/Engine/FHitResult/.

[6] Epic Games Inc. *Pawn*. URL: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Framework/Pawn/.

[7] Epic Games Inc. *Timers*. URL: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Timers/.

[8] Epic Games Inc. *Tracing*. URL: https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Tracing/Overview/.

[9] Epic Games Inc. *Unreal Engine Terminology*. URL: https://docs.unrealengine.com/4.26/en-US/Basics/UnrealEngineTerminology/.

Съдържание