 Технически Университет – София

Факултет по Компютърни Системи и Технологии

Катедра Компютърни Системи

**Дипломна работа**

на

**Пламен Пламенов Коканов, фак.№ 121209161**

Специалност: Компютърни системи и технологии

Образователно-квалификационна степен: бакалавър

**Тема:**

**Разработване на мултиплатформена тримерна компютърна игра**

**Дипломен ръководител:**

доц. д-р Милена Лазарова

Съдържание

[**1.** **Въведение** 3](#_Toc399995865)

[**2.** **Постановка на дипломната работа, цели и задачи** 5](#_Toc399995866)

[2.1. *Проблеми на модерните видео игри* 5](#_Toc399995867)

[2.2. *Цели и спецификации на разработваната игра* 6](#_Toc399995868)

[2.2.1. Тип на играта 6](#_Toc399995869)

[2.2.2. Тип на използваната компютърна графика 6](#_Toc399995870)

[2.2.3. Разположение на камерата спрямо модела управляван от играча 8](#_Toc399995871)

[2.2.4. Възможност за игра в мрежа 8](#_Toc399995872)

[2.2.5. Интуитивно меню с достатъчно широк набор от опции 10](#_Toc399995873)

[2.2.6. Поддръжка на множество платформи 10](#_Toc399995874)

[2.3. *Задачи* 11](#_Toc399995875)

[2.3.1. Основни процеси при проектирането на игри 11](#_Toc399995876)

[2.3.2. Добиване на техническа експертиза 11](#_Toc399995877)

[2.3.3. Създаване на тримерни модели, анимации,текстури и озвучаване 12](#_Toc399995878)

[2.4. *Подобни продукти* 13](#_Toc399995879)

[2.4.1. Counter Strike: Global Offensive 13](#_Toc399995880)

[2.4.2. Dead Nation 14](#_Toc399995881)

[2.4.3. Lara Croft and the Guardian of Light 15](#_Toc399995882)

[**3.** **Функционално описание на приложението** 16](#_Toc399995883)

[3.1. *Меню* 16](#_Toc399995884)

[3.1.1. Стартово меню 17](#_Toc399995885)

[3.1.2. Главно меню 17](#_Toc399995886)

[3.2. *Режим на игра* 18](#_Toc399995887)

[3.2.1. Ниво 18](#_Toc399995888)

[3.2.2. Виртуален аватар на играч 18](#_Toc399995889)

[3.2.3. Графичен Интерфейс 19](#_Toc399995890)

[3.3. *Игра в мрежа* 20](#_Toc399995891)

[**4.** **Програмна реализация** 21](#_Toc399995892)

[4.1. *Използвани програмни средства* 21](#_Toc399995893)

[4.1.1. Език за програмиране 21](#_Toc399995894)

[4.1.2. Game Engine 21](#_Toc399995895)

[4.1.3. Софтуер за тримерно моделиране 22](#_Toc399995896)

[4.1.4. OpenGL 23](#_Toc399995897)

[4.2. *Архитектура на Unity 3D проект* 23](#_Toc399995898)

[4.3. *Управление на състоянието на играта* 26](#_Toc399995899)

[4.3.1. Основна концепция 26](#_Toc399995900)

[4.4. *Управление на графичния аватар на играча* 28](#_Toc399995901)

[4.5. *Реализация на графичния интерфейс* 29](#_Toc399995902)

[4.5.1. Менюта 29](#_Toc399995903)

[4.5.2. Интерфейс по време на игра 29](#_Toc399995904)

[4.6. *Реализация на сървър и клиент* 30](#_Toc399995905)

[4.6.1. Методи за сървърна реализация в Unity 30](#_Toc399995906)

[**5.** **Ръководство за използване** 33](#_Toc399995907)

[5.1. *Работа с менютата* 33](#_Toc399995908)

[5.1.1. Стартово меню на Unity (само при персонални компютри) 33](#_Toc399995909)

[5.1.2. Главно Меню на играта 34](#_Toc399995910)

[5.1.3. Работа с менюто по време на игра 37](#_Toc399995911)

[5.2. *Начин на игра след зареждане на ниво* 39](#_Toc399995912)

[5.2.1. Управление на тримерния аватар на играча 39](#_Toc399995913)

[5.2.2. Стрелба , муниции и точки здраве 39](#_Toc399995914)

[**6.** **Заключение** 40](#_Toc399995915)

[**7.** **Използвана литература** 41](#_Toc399995916)

[**8.** **Приложение 1 – извадка от сорс кода на разработката** 42](#_Toc399995917)

1. **Въведение**

Видео игрите представляват електронни игри, които включват човешко взаимодействие с потребителски интерфейс с цел генериране на визуална обратна връзка върху видео устройство, което може да бъде всякакъв вид електронен дисплей, притежаващ възможността да изрисува две- или триизмерни изображения. Електронните системи, използвани, за да се играят видео игри, се наричат платформи; примери за това са: персонални компютри, конзоли за видеоигри и мобилни устройства (телефони, таблети и т.н.) и операционните системи работещи върху тях. В наши дни на видео игрите не се гледа само като на парче софтуер, но и като форма на изкуство и цялостно развита индустрия.

Управлението на видео игрите се осъществява чрез устройства наречени контролери и варира между различните платформи. При персоналните компютри най-често се използват клавиатура и мишка, при конзолите – джойстик, при мобилните устройства – сензорен дисплей (върху който могат да са изобразени графично всякакъв вид контролери) или вградената клавиатура. През последните години с бързи темпове се развиват и методи за управление чрез видео камери и софтуер за разпознаване на движения и изображения.

Видео игрите обикновено използват допълнителни средства за осигуряване на интерактивност и информация за играча. Озвучаването е универсално, имплементирано посредством звукови устройства, като високоговорители и слушалки. Чрез хаптични периферни устройства може да се получи и друг вид обратна връзка, която може да бъде силова или стимулирани вибрации в контролера.

Всичко това е в резултат от дългогодишна еволюция. Зората на видео игрите датира от 50-те години на XX век, като главната им цел е била свързана с развитието на изкуствения интелект, интерфейса между компютрите и хората и компютърните симулации (най-вече за военни цели). Игрите се използвали рядко за развлечение поради големите ресурси необходими за операцията на тогавашните компютри. Първите комерсиални видео игри със стриктно развлекателна цел се започват да се произвеждат едва след 1971 година, като с най-голям успех се отличава играта „Pong“ създадена от „Atari“ през 1972 г. От тогава насам игрите добиват все по-голяма популярност като през 2014 г. международните годишни приходи от видео игри (достигащи огромни суми от порядъка на 81,5 милиарда долара) надвишават двойно тези на филмовата индустрия. Близо 1.2 милиарда души по целия свят прекарват свободното си време играейки компютърни игри, като средната възраст на играещите е 35 години. Нашумяват все повече и повече т.нар. eSports – electronic sports. За част от съвременните игри за множество хора дори има и световни шампионати, като благодарение на streaming технологии е възможно състезанията да се гледат от запалянковци от вкъщи без необходимостта от закупуването на билети. Световното първенство на играта League of Legends през 2013 бе наблюдавано от близо 38 милион зрители по целия свят. Тази популярност е добра предпоставка за развитието на бизнес свързан с разработката на игри.

От икономическа гледна точка разработването на компютърни игри изглежда доста привлекателно. Множество от необходимите инструменти за тази цел се предлагат с безплатен лиценз и готов продукт може да се получи дори и от „екип“ състоящ се от 1 човек само за няколко месеца, а печалбата може да е огромна – пример за това е “Flappy Bird”, разработена от Нгуиен Ха Донг за мобилни устройства през 2013. В нея играчът управлява кълбовидна птица, която се движи постоянно напред и трябва да избягва препятствия. Състой се от 1 безкрайно ниво. Играта набира популярност чак в началото на 2014 година, но Нгуиен печели по $50000 на ден от реклами вградени в нея. Дори след като той я сваля от магазина на Apple, в Amazon започват да се продават iPhone4 за цени от порядъка на $600 с инсталиран Flappy Bird. Макар и точно такива случай да са рядко срещани, броя на т.нар. „независими“ разработчици и издатели на видео игри нараства. Това става благодарение на доброто развитие на интернет и социалните мрежи (създаване на групи от геймъри, лесен начин за даване на обратна връзка, комуникация между разработчици работещи в отдалечени точки на света) както и на сайтове за набиране на средства като например “Kickstarter”.

Що се отнася до ефекта на игрите върху хората – мненията са доста противоречиви. Според някой хора те имат вредно влияние върху подрастващите, а дори и върху по-възрастните играчи. Някой дори твърдят че причината за много от престъпленията по света е наличието на насилие в игрите за по-големи. Факт е, че доказателства липсват и всички тез твърдения се базират върху спекулации. От друга страна, научните изследвания на Изабела Гранич и нейните колеги показват че компютърните игри имат положително влияние: те развиват мозъчната функция – екшън ориентираните игри подобряват рефлексите, периферното зрение и вниманието, докато игрите със загадки подпомагат взимането на правилни и бързи решения. Играещите се учат да решават забавни пъзели. Съществува и важен социален аспект. За разлика от стереотипа на затвореният геймър, който използва компютърните игри, за да избягва всякакъв вид връзка с отвъдния свят, в момента повечето хора играят „multi-player” точно с идеята да общуват с други индивиди.

1. **Постановка на дипломната работа, цели и задачи**
   1. *Проблеми на модерните видео игри*

Едни от основните проблеми на модерните видео игри са следствие именно от възможността за бърза печалба без полагането на много труд. Това води разработване на игри с много ниско качество, малко съдържание, липса на оригиналност (“Flappy Bird” – описана по-горе, заимства дизайна на нивата си изцяло от известната игра “Super Mario”) и най-лошото – банален gameplay (взаимодействието между играчите и играта). Наблюдава се тенденция към намаляване на трудността, като все повече се появяват игри, който „водят играча за ръка“, показвайки му във всеки един момент кой бутон трябва да бъде натиснат, за да бъде спечелена играта, с прекалено дълго време за реакция (в крайни случаи – от порядъка на 30 секунди). Това ги превръща по-скоро в интерактивни филми, премахвайки от тях една основна човешка функция – решаването на проблеми, намалявайки възможности за каквото и да е мозъчно развитие, съревнованието между играчите и чувство на удовлетвореност при победа. Игри, който преди години биха се играли само от деца, в наши дни са насочени към възрастни, като единственото различно е по-високото ниво на насилие. Главна роля в това проблематично развитие играе растящата употребата на мобилните устройства като платформа за игри. По-ниската производителност на хардуера им и ограниченията, който това налага върху графичните възможности, както и големият и лесно достъпен пазар (благодарение на Google Play и iPhone App Store) привличат по-малки и нови фирми или независими разработчици, които се стремят да произвеждат игри за възможно най-кратко време.

Всичко това от своя страна поражда ново поколение геймъри, които са привързани към леснотата. От мобилните устройства тези тенденции постепенно се прехвърлят и върху други платформи, а дори и извън сферата на видео игрите.

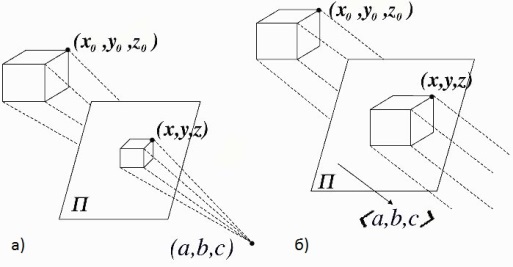
* 1. *Цели и спецификации на разработваната игра*

Основната цел е разработване на видео игра, при която споменатите горе проблеми липсват или са минимално изразени. Като се има това в предвид могат да бъдат формулирани следните спецификации:

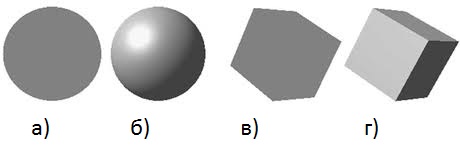
* + 1. Тип на играта

Екшън shooter от трето лице – това представлява компютърна игра, в която играчът контролира виртуален аватар разполагащ с различни видове оръжия, които се използват за стрелба по определени мишени. Най-често съществува система за точкуване свързана с мишените и играчът, събрал най-много точки, побеждава. В голяма част от shooter игрите предназначени за възрастни мишените са противникови играчи управлявани от други хора или изкуствен интелект като целта е противниците да бъдат убити възможно най-бързо и ефективно. Макар и да звучи брутално, тези игри са добър начин за разпускане на напрежението, натрупано през работния ден. Подтикват стратегическо мислене поради нуждата да бъде надхитрен противника като се избират стратегически места за стрелба, различни методи на атака или защита, обмислят се необходимите оръжия за изпълняване на определена цел с възможно най-висока ефективност и преценка на възможните ходове на другите играчи.

* + 1. Тип на използваната компютърна графика

Тримерната компютърна графика спомага за по-доброто потапяне в обстановката на играта, обаче е много по-трудна за реализация от двумерната и изисква повече компютърни ресурси. И при двата вида, изображението, което се получава, се разполага върху двумерен екран, което налага прилагането на допълнителна обработка при използването компютърна графика с три измерения. В основата й стоят математически представяния на тримерни модели (фигури) съставени от триъгълници и разположени в 3D поле (световни координати). В това поле се намира и камера, като всичко, което попадне в обхвата й, евентуално ще бъде изобразено върху екрана. От положението и настройките на камерата зависи използваната проекция като съществуват два основни типа: перспективна (фиг. 1а) и паралелна (фиг. 1б). При първия тип моделите се изобразяват върху екрана така както биха се видели от човешкото око. Това има предимството по-далечни обекти да изглеждат по-малки. При втория – всички линии на моделите, които са били перпендикулярни при математическата репрезентация, остават перпендикулярни и в крайното изображение.

фиг. 1

Осветлението и сенките също оказват влияние върху крайния резултат. Без тях изображението би изглеждало двумерно. На фиг. 2а) и фиг. 2в) са изобразени съответно сфера и куб при липса на осветление, докато на фиг. 2б) и фиг. 2г) са изобразени същите обекти, но осветени.

фиг. 2

Наличието на светлина и сенки усложнява графичната постановка и допринася към изискването за по-голяма производителност на използваната платформа, но както се забелязва, без тях всичко в играта би било неразличимо.

* + 1. Разположение на камерата спрямо модела управляван от играча

Камерата трябва да бъде разположена зад играча и над него в световни координати и насочена към модела му, сключвайки ъгъл от 45о с равнината на движение. Целта е постигане на достатъчно голям обхват на зрение, който да включва възможно най-много предмети заобикалящи играча.

* + 1. Възможност за игра в мрежа

Всяка игра е по-забавна, когато може да се играе с други хора, най-вече приятели, в реално време. Включването на multiplayer се съпътства от доста технически проблеми. Всеки един клиент (инстанция на играта) трябва да комуникира през интернет мрежа със останалите клиенти и сървъра, обменяйки данни за това какво е текущото състояние на съответния играч (текуща позиция, кръв, оръжие, брой точки до победата и т.н.) и текущото състояние на играта като цяло, например кое е текущото ниво, дали е стигнат края на играта. Освен това тези състояния трябва да са синхронизирани между отделните клиенти. Това е проблематично, поради крайната скорост с която горепосочените данни се предават по мрежата.Получава се забавяне което се нарича лаг и трябва да бъдат имплементирани технологии, който го ограничават до определени допустими граници, които за shooter и екшън-базирани игри са от порядъка на 50 ms. При стойности надвишаващи тази граница започват да се наблюдават всякакви нежелани ефекти, като най очевидния е моделите на отдалечените играчи всъщност да не са точно там където са изобразени в локалната игра и при опит за взаимодействие с тях те не реагират.

Друг аспект на режима multiplayer е сигурността и защитата от хакери. Включването на защити може да доведе до проблеми при бързодействието на играта, затова е по-добре тези неща да се оставят като опции, от които потребителят на играта да избира – дали да бъде включено криптиране на връзката, и типът на сървъра, който може да бъде една от 3 опции – не-авторитарен, полу-авторитарен или авторитарен.

* ***Не-авторитарен сървър*** – сървърът играе ролята на посредник на данни между отделните клиенти без да извършва каквито и да било други логически изчисления или проверки. Това повишава производителността, но отваря врата на хакерите, тъй като цялата логика на играта се извършва на самите клиенти всеки играч би могъл да си подправи играта, така че да има безкрайни животи, муниции, да убива всичко от един удар, да се движи с огромна скорост и т.н., след което да изпрати тази информация на сървъра, който просто ще я препрати към другите клиенти. Този тип е подходящ за локални мрежи, където играещите се познават или си вярват и също така позволява един от играчите да стартира сървъра върху собствената си машина поради ниските системни изисквания.
* ***Полу-авторитарен сървър*** – част от логиката на играта (най-вече по простите изчисления като преместване на модела на играча, генериране на изстрели, отнемане на количество кръв, опростени алгоритми за откриване на колизии) се преместват върху сървъра. На клиентите се оставя само генерирането на изображение и по-сложните физически симулации. Това ограничава до някаква степен хакерите. Този тип сървъри е предпочитан за масивни мрежови игри (такива в който на едно място от света на играта и по едно и също време могат д се съберат над 200 човека)
* ***Авторитарен сървър* –** от гледна точка на сигурността, това е сървърът с най-високото ниво на сигурност, но и най-високи системни изисквания, понеже цялата логика на играта се пренася върху него, докато единственото задължение на клиентите е да приемат командите на играчите, да ги изпращат към сървъра, и да генерират изображение на екрана в зависимост от командите получени като отговор от сървъра. Програмната реализация при този вариант е най-сложна. След натискането на бутон от потребителя, времето необходимо на съответната команда да бъде изпратена на сървъра, обработена и отговор от обработката върнат към клиента, може да е значително дълго. Това налага клиента да използва техники за предвиждане на резултата от обработката направена на сървъра и след това сравнение между предвидените и получените от сървъра стойности. Този подход е идеален при игри с малък брой хора и когато защитата от хакери е трябва да е много добра. Повечето модерни игри, на който се гледа като eSport са приели този подход.
  + 1. Интуитивно меню с достатъчно широк набор от опции

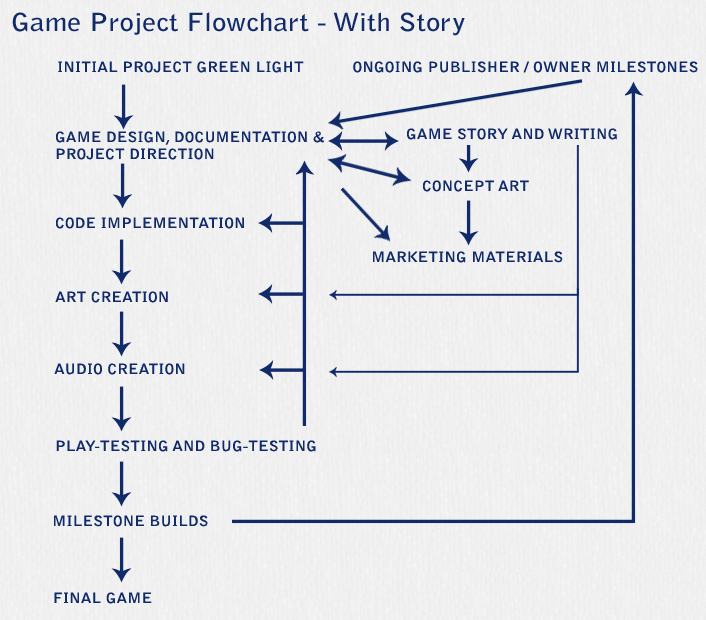
Повечето игри могат да бъдат разделени на две части – главно меню, чрез което потребителят може да регулира различни параметри на играта и втора част, в която се изпълнява самата игра. Главното меню често е съвкупност от под-менюта, едно от които съдържа бутона за старт на игра. Текущата разработка трябва да разполага с менюта за регулация на: настройки на качеството на изображението, смяна на бутоните/командите за контрол над играта, козметични промени на аватара на играча, смяна на името на играча, настройка на сървърните опции (име на сървър, брой играчи, начално ниво, лимит на точките необходими за победа). Налага се предоставяне на интерфейс бутони за стартиране на сървър и присъединяване към активен такъв, за излизане и спиране на приложението. Тези бутони и опции трябва да са достъпни както в началото на играта, така и след зареждане на определено ниво (по време на игра). Необходимо е достатъчно яснота за това за какво точно се отнася всеки един от бутоните в менютата.

* + 1. Поддръжка на множество платформи

Играта трябва да може да работи на следните платформи: Mac, персонални компютри с операционни системи Windows и Linux, iOS, Android версии над 2.3, PlayStation 3, Xbox 360. Съвременните мобилни устройства разполагат с достатъчно добър хардуер и производителност, за да поддържат компютърна графика и анимация, която би изглеждала достатъчно добре както на персонални компютри така и на конзоли. Това може да се види от игри като “Heroes of Order and Chaos” и “Order and Chaos” на Gameloft, които се доближават по естетическо качество до “League of Legends” на RIOT, която от своя страна е една от най-известните игри от последните 2 години. “Asphalt 8” и “Modern Combat 5” (също на Gameloft) притежават по-хубава графика и анимации от голяма част от игрите предназначени за компютри. Обаче споменатите до тук игри са предназначени само за мобилни устройства, което ограничава хората не-разполагащи с подходящи платформи или тези който не харесват мобилните устройства като платформа за видео игри. Мултиплатформените игри спомагат за обединението на общностите от геймъри, привързани към определени платформи. Част от реализирането на тази цел е и разработването на графичен интерфейс и начин на управление на играта, които да имат сходно поведение върху различните платформи. Потребителите не трябва да печелят предимство (или да губят такова) в зависимост от платформата.

* 1. *Задачи*
     1. Основни процеси при проектирането на игри

Проектирането на една игра минава през 7 етапа (фиг. 3)

1. Дизайн
2. Програмна реализация
3. Създаване на графични модели, текстури и анимации
4. Създаване на озвучението
5. Тестване
6. Публикуване на междинни билдове
7. Издаване на играта

фиг. 3

* + 1. Добиване на техническа експертиза

Една от основните задачи е запознаването с добри практики за дизайн на програмната реализация на видео игрите. При нужда от добавяне на нова функционалност или промяна на стара, това да става възможно най бързо и да не се налага коренна промяна във всички файлове на играта. Софтуерната архитектура трябва да бъде проектирана по такъв начин, че да е лесна за разбиране от евентуални нови хора, които ще работят по проекта.

В наши дни съществува голям избор от програми спомагащи създаването на компютърни игри - game engines. Те биват специализирани програматорски рамки улесняващи процеса на разработване, понеже се грижат за изобразяването на графичните примитиви (няма нужда от директна работа със специфични графични библиотеки от по-ниско ниво), управляват физичните симулации, светлинните източници и сенките и откриването на колизии. Така се дава възможност на разработчиците на игри да се фокусират главно върху логиката на самата игра. Изборът на подходящ game engine e много важен, понеже определя начинът на програмиране, важен фактор е в бързодействието на играта, качеството на компютърната графика и физичните симулации. Поради тези причини е хубаво да бъде отделено достатъчно време за проучване на достъпните вариантите за game engine.

Възможността за игра в мрежа е може би най-трудният аспект на проекта. Изисква се направата на проучване за това какъв протокол е най-добре да бъде използван при трансфера на данни и дали няма да е необходимо за различните типове данни да се използват различни протоколи. Трябва да се потърсят вече готови програмни рамки и практики за мрежова комуникация, като например remote procedure calls. Необходимо е запознаване с видовете сървърни архитектури използвани при multiplayer игрите, както и начините на връзка с отделните клиенти.

Ще е необходимо и запознаване с начина на работа с избрания game engine, което може да включва научаване на нов език за програмиране. Трябва време да се свикне с използването на интерфейса и научаване на повечето функции.

* + 1. Създаване на тримерни модели, анимации,текстури и озвучаване

Тези задачи обикновенно се извършват от хора занимаващи се с изкуства, но при текущата разработка липсват такива. В такъв случай по-простите модели, анимации, текстури и озвучение могат да бъдат създадени с помощта на програмни средства предназначени за това, докато по-сложните могат да бъдат намерени в интернет.

* 1. *Подобни продукти*
     1. Counter Strike: Global Offensive

„Counter Strike: Global Offensive“ (съкратено като CS: GO) е онлайн тактически shooter от първо лице, разработен от Valve Corporation и Hidden Path Entertainment. Това е четвъртата игра от поредицата „Counter Strike“.

„Counter Strike: Global Offensive“ е издадена на 21 август 2012 г. и е предназначен за следните платформи: Microsoft Windows и OS X, Xbox и PlayStation. Бета версия е издадена за Linux през септември 2014 г. Между-платформен multiplayer бе планиран между Windows, OS X и PlayStation, но в крайна сметка бе ограничен до Windows и OS X, поради разликите в обновителната честота необходима за синхронизация между клиентите на отделните системите.

Подобно на предишните игри от поредицата всеки играч се присъединява към 1 от два отбора – Терористи или Контра-Терористи и се опитва да изпълни определени задачи или да елиминира противниковия отбор. Играта работи в кратки кръгове, които завършват, когато всички играчи на една страна са мъртви или задачата на екипа е изпълнена. За повечето режими на игра, след като играчът умре, трябва да изчака, докато рунда свърши, за да се съживи. Играчите купуват оръжия и оборудване в началото на всеки кръг с пари, броят на които зависи от производителността на съответния отбор от предходните рундове. В допълнение, когато един кръг завърши всички играчи получават някаква сума пари, като играчи от отбора победител получават значително повече.

„Counter Strike: Global Offensive“ получава положителни отзиви от професионалните критици. Сайтът Metacritic дава на PC версията обща оценка 83% на базата на 38 ревюта от професионални критици.

Снимки от играта са показани на фиг.5 и фиг.6

фиг. 5 фиг. 6

* + 1. Dead Nation

„Dead Nation“ представлява Shoot ‘Em Up игра с камера от тип горе/долу (камерата е разположена над играча и насочена към него) предназначена за PlayStation 3, разработена от финландското студио Housemarque и е издадена през 2010 г. Shoot ‘Em Up означава, че играчът разполага с набор от оръжия, с помощта на които трябва да убие орди от постоянно прииждащи врагове. Играта се развива в измислен свят, изпаднал в зомби-апокалипсис. Играчът влиза в ролята на мъж или жена и трябва да се отбранява от прииждащите различни видове зомбита. „Dead Nation” получава предимно позитивни отзиви, като сайтовете GameRankings и Metacritics й дават съответно 78% и 77%. През 2013г дори е включена в топ 25 от класацията на IGN (гейминг сайт) за игри предназначени за PlayStation Network, като заема 13-то място.

Примери от играта са показани на фиг.7 и фиг. 8

фиг. 7



фиг. 8

* + 1. Lara Croft and the Guardian of Light

„Lara Croft and the Guardian of Light”е платформена (голяма част от начина на игра се базира на изпълнението на трудни скокове от една платформа към друга) екшън видео игра, разработена от Crystal Dynamics и публикувана от Square Enix за Microsoft Windows, PlayStation 3, Xbox 360 и IOS. За разлика от предишните игри от поредицата, играта е с тежък акцент върху кооперацията с други играчи. При игра в мрежа, играчите поемат ролята или на Лара Крофт или на 2000-годишният майски войн на име Тотек. Те трябва да работят заедно, за да спрат злия дух Ксолотл и да върнат „Огледалото на пушека“.

За разлика от предишните части на поредицата, които са приключенски игри с перспективна проекция на камерата, „The Guardian of Light” е нелинейна платформена екшън игра с фиксирана изометрична камера, подобна на „Tomb Raider: The Prophecy” за Game Boy Advance . Всеки от двата героя притежава уникални оръжия и умения. Както при някои предишни заглавия от поредицата „Tomb Raider”, Лара запазва нейните двойна пистолети с безкрайни боеприпаси и захващаща се кука, която може да се използва, за преминаване на пропасти. Totec носи копия, които могат да се използват както за оръжие, така и върху околната среда, за да може Лара да се изкачва върху тях. И двата героя носят неограничено количество бомби, които могат да бъдат взривявани. Гробници могат да бъдат претършувани и някои от тях имат сложни пъзели, които трябва да се решат. „The Guardian of Light” може да се играе с един играч, и втори играч може да се включи по всяко време. Необходими са от 8 до 9 часа игра, за да бъдат минати всички нива.

Играта получава много добри оценки след излизането на пазара. Сайтът Metacritic й дава 85% за Xbox версията, 84% за PlayStation версията и 82% за PC версията. От GameRankings получава съответно 86%, 82% и 81%. Според някой хора дори е една от най-добрите игри от поредицата „Tomb Raider”. (фиг. 9 и фиг 10)

 фиг. 9

 фиг. 10

1. **Функционално описание на приложението**

Приложението може да бъде разделено на 2 основни части: меню, в което се извършват настройки и режим на игра, в който група играчи са се свързали към един съврър и е заредено първоначално избрано ниво. Сървърът може да бъде стартиран на платформата използвана от един от играчите, както и на отдалечена сървърна машина (dedicated server)

* 1. *Меню*

Менютата в една игра предоставят на потребителя опции, чрез които играта може да бъде персонализирана. Текущата разработка улеснява това като групира тези опции в отделни под-менюта в зависимост от нещата, който променят.

* + 1. Стартово меню

Стартовото меню се появява веднага след стартиране на играта. То предоставя възможнст за общи начални настройки, които включват:

**Графични настройки**

1. Настройка на резолюцията на играта
2. Дали играта да бъде подкарана в режим на прозорец или на цял екран
3. Избор на качеството на графиката, като са налични следните варианти: най-бързо; бързо; прсто; добро; красиво; фантастично;

(под най-бързо и бързо се разбира степента на бързодействие на самата игра – колкото по-бързо, толкова по-лоша е графиката)

1. Избор на монитор (дисплей), върху който да бъде показана играта.

**Настройки на клавишите за управление на играта**

(дава се възможност за специфициране на главен и вторичен бутон (при натискане на кой да е от двата бутона се извършва една и съща функция)

1. Бутон за движение нагоре
2. Бутон за движение надолу
3. Бутон за движение наляво
4. Бутон за движение надясно
5. Бутон за главна стрелба
6. Бутон за вторична стрелба
7. Бутон за третична стрелба
8. Бутон за скок

Вариантите за избор на бутон включват както клавиши от клавиатура за компютър, така и бутони от джойстик за конзоли.

Стартовото меню съдържа и два бутона, единият от тях пуска играта, а другият спира процеса.

Това меню е налично при всички проекти реализирани на Unity 3D.

* + 1. Главно меню

Главното меню се появява след натискане на бутона старт от стартовото меню и зареждане на самата игра. То представлява начална точка, от която могат да бъдат стартирани набор от други под-менюта, които включват:

1. Меню за стартиране на сървър – позволява избор на име за свървъра, максимален брой на вързаните клиенти, избор на ниво за играта и стартиране на сървъра.
2. Меню за връзка към съврър – дава възможност за избор на свърър, към който да се направи връзка
3. Меню с опции – позволява промяна на силата на звука, настройки на качеството на графиката, персонализация на играча (избор на модел и име), настройка на бутоните за управление.

Също така е наличен бутон за изход от играта и спиране на приложението

* 1. *Режим на игра*
     1. Ниво

Нивата представляват тримерни модели с много по-големи размери от тези на аватарите на играчите. В общия случай са съвкупност от графични симулации на сгради, стаи и помещения с разнообразна архитектура, в които играчите могат да се движат.

В различни точки от едно ниво са разположени специални предмети, които могат да се събират от играчите. Определено време след като един предмет бъде събран, той се появява отново на същото място. Те представляват муниции (за всяко оръжие има точно определен вид муниции) и аптечки, които възстановяват изгубени точки здраве (описани в 3.2.2)

* + 1. Виртуален аватар на играч
* *Оръжия*

В режим на игра играчите контролират виртуален аватар, който представлява тримерен модел на човек разполагащ за набор от оръжия. Възможна е смяна между отделните оръжия, като всяко има различна функция. В текущия момент са налични три типа – нож, пистолет и автомат. Ножът се използва за близък бой и не е много ефективен, пистолетът стреля бавно, докато автоматът е бърз и нанася най-големи щети. Всяко от оръжията разполага с пълнител, като при изхабяването му, оръжието трябва да бъде презаредено. Не се позволява стрелба по време на презареждане. Играчите разполагат с определн брой муниции и при изхабяването им единствената опция остава използването на ножът

* *Точки здраве*

Аватарите разполагат с определено количество здраве представено чрез точки. Всеки точен изстрел нанася щети върху уцеления играч и намалява здравето, което притежава. Когато то стигне 0 точки, аватарът на съответния играч умира, на противниковия играч успял да нанесе последния удар се дават точки, а след определен период от време аватарът се съживява, като моделът му се поставя на случайно място от нивото.

* *Управление*

Аватарите могат да се движат напред, назад, наляво и надясно и да се въртят около себе си. Посоката на стрелба зависи от това на къде е завъртян аватарът. Възможно е както бавно така и бързо ходене. При бавното ходене изстрелите са по-точни, докато при бързото – по-неточни.

* + 1. Графичен Интерфейс

Основната функция на графичния интерфейс е да покаже на потребителя възможно най-много информация за текущата игра. Включва няколко основни групи от изображения:

* *Оръжие*

Показва на потребителя текущото оръжие, колко патрони има в пълнителя и колко патрони за това оръжие са му останали като цяло.

* *Точки здраве*

Показва текущите точки здраве, с който разполага играча, както и максималния брой точки, който може да има.

* *Посоката на стрелба*

Показва точно къде се е прицелил играча.

* *Меню в режим на игра*

Представлява меню, даващо възможност за промяна на настройките на играта. Представлява съвкупност от под-менютата описани в точка 3.1. с изключение на менюто за създаване на сървър и това за връзка към сървър. Съдъжа бутони за изход от играта и спиране на приложението или връщане към главното меню.

* *Таблица за текущ резултат*

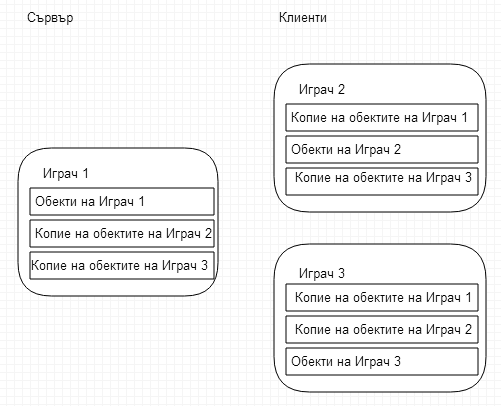
Таблица, която показва броя на убииствата, умиранията и лага на всеки един от играчите за текущата игра.

* 1. *Игра в мрежа*

Възможността за игра в мрежа, като множество хора играят на еедин сървър е една от основните функционалности на разработваната игра.

Предлага се връзка и между разнородни платформи – например към сървър стартиран на Android може да се върже клиент, използващ Windows.

Играта в мрежа се осъществява като един от потребителите стартира сървър на собствената си платформа от менюто за стартиране на сървър. Така неговата игра се превръща в сървър и отговаря за осъществяването на комуникацията между отделните клиенти. На фиг. 11 е показана примерна топология на мрежова игра с трима участника.



фиг. 11

В този случай играч 1 е стартирал сървъра, а играч 2 и 3 са клиентите, които са се свързали към него. Всяка една от игрите разполага както с локални обекти, контролирани от съответния играч, така и с копия на отдалечени обекти, които се контролират от другите играчи. Тези копия се инициализират при връзка към сървъра, и главната му цел е да осигури комуникационен канал, през който игрите да могат да синхронизират състоянието на копията на обектите с оригиналните.

1. **Програмна реализация**
   1. *Използвани програмни средства*
      1. Език за програмиране

C# е обектно-ориентиран език за програмиране, разработен от Microsoft, като част от софтуерната платформа .NET. Тя предоставя програмен модел, библиотека от класове (FCL, Framework Class Library) и среда за изпълнение на написан специално за нея програмен код (CLR, Common Language Runtime). Платформата .NET е предназначена за различни версии на Windows. Целта на проекта с отворен код (подход в проектирането, разработката и дистрибуцията на софтуер, позволяващ свободен достъп до софтуерния код) Mono, спонсориран от Novell, е да даде възможност програми, написани на .NET Framework, да работят и на други операционни системи, като Linux, Solaris, Mac OS, BSD и др. Основна концепция на C# са класовете, които се дефинират чрез свойствата (атрибутите) и поведението (методите) на обектите. Няма глобални променливи и функции - всичко се дефинира в класове, които съдържат в себе си методи, а в методите е разположена програмната логика. Липсва и множественото наследяване от [C++](http://bg.wikipedia.org/wiki/C_Plus_Plus), но то в някаква степен е заместено от интерфейсите. Кодът написан на C# се компилира до платформено-независим междинен език, наречен CIL (Common Intermediate Language). По време на изпълнение CIL кодът (т. нар. „управляван код“) бива автоматично компилиран от CLR за конкретната хардуерна платформа и операционна система, с която работи потребителят.

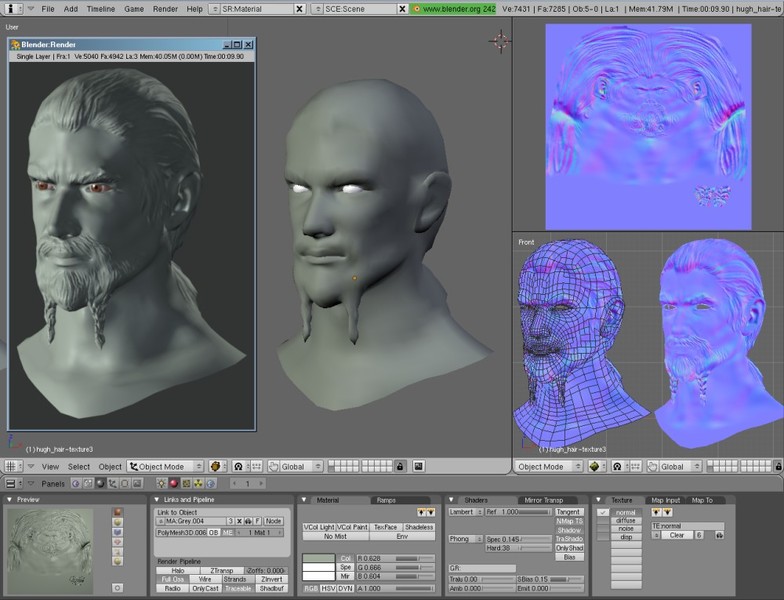
* + 1. Game Engine

Unity 3D представлява мултиплатформена система за създаване на игри разработена от Unity Technologies и включваща game engine и интегрирана среда за разработка (IDE). Използва PhysX на Nvidia за физични симулации. В един проект разработчиците имат контрол върху доставките за мобилни устройства, уеб браузъри, настолни компютри и конзоли. Поддържаните платформи включват: BlackBerry 10, Windows Phone 8, Windows, OS X, Linux (главно Ubuntu), Android, IOS, Unity Web Player (включително Facebook), Adobe Flash, PlayStation 3, PlayStation 4, PlayStation Vita, Xbox 360, Xbox One, Wii U и Wii. Unity Web Player представлява plugin за браузъри, поддържан само под Windows и OS. В зависимост от платформата и настройките графичният engine използва следните приложно-програмни интерфейси (APIs): OpenGL и Direct3D. Unity позволява специфициране на компресията и настройките на резолюцията на текстурите за всяка платформа която поддържа разработваната игра. Осигурява подкрепа за bump mapping, reflection mapping, parallax mapping, Screen Space Ambient Occlusion (SSAO), динамични сенки използващи shadow maps, render-to-texture и full-screen post-processing ефекти. Графичният engine на Unity разполага с множество варианти за shader и декларитивна резервна спецификация, което позволява на Unity да засече най-добрият вариант за използвания хардуер и ако не бъде открит съвместим, да се използва алтернативен резервен shader. Програмирането в Unity е скриптово базирано и изградено върху Mono проекта обяснен в 4.1.1. Програмистите могат да използват UnityScript (персонализиран език с ECMAScript вдъхновени синтаксис, наречен JavaScript от софтуера), C #, или Boo (който има Python вдъхновен синтаксис).

Unity Asset Store е дом на нарастваща библиотека от свободни и търговски активи, създадени както от Unity Technologies, така и от членове на Unity общността. Голямо разнообразие от активи е на разположение, което обхваща всичко от текстури, модели и анимации, до цели примери за проекти, уроци и разширения за редактора на Unity.

* + 1. Софтуер за тримерно моделиране

Blender е 3D софтуер с отворен код за моделиране, анимация, текстуриране, рендъринг, комозитинг, създаване на интерактивно съдържание, а също така и за създаване на компютърни игри чрез Blender - Game Engine. В момента поддържа 25 езика на интерфейса и е достъпен за всички основни операционни системи под GNU General Public License. На фиг. 12 е показана моделирана човешка глава на Blender с направен рендъринг (в ляво)



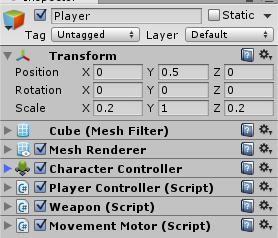
фиг. 12

* + 1. OpenGL

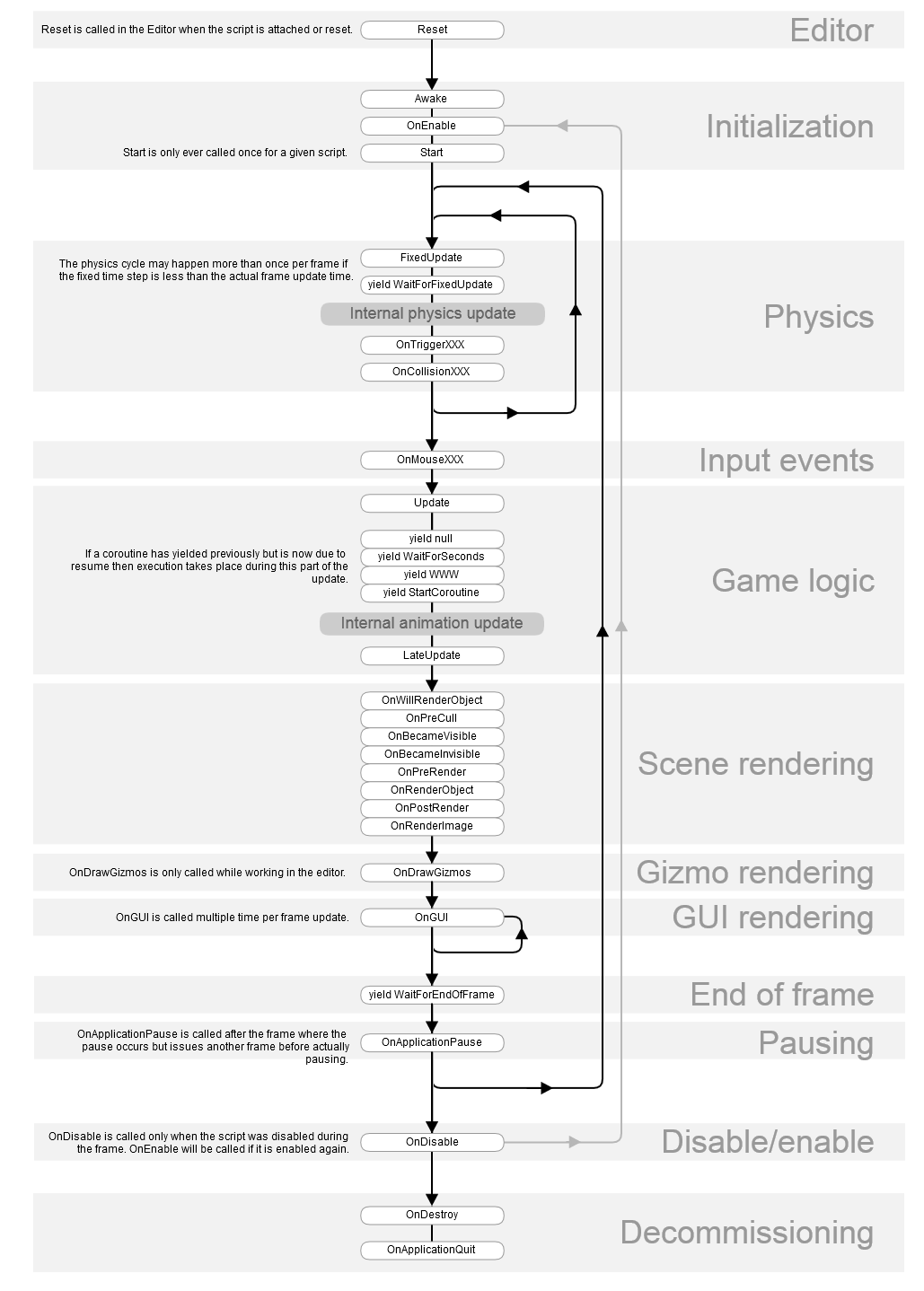
OpenGL (Open Graphics Library) е мощен приложно-програмен интерфейс (API) за реализиране на лесно преносими графични приложения. Създаден през 1992г, OpenGL бързо става един от най-популярните програмни интерфейси за реализиране на 2D и 3D графика. За това допринасят широката му достъпност, съвместимостта му с различни операционни системи и с различни компютърни платформи. Тази библиотека е подходяща за приложения изискващи високо качество на изображението, комбинирано с добра производителност, за да бъде възможно генерирането му в реално време.

* 1. *Архитектура на Unity 3D проект*

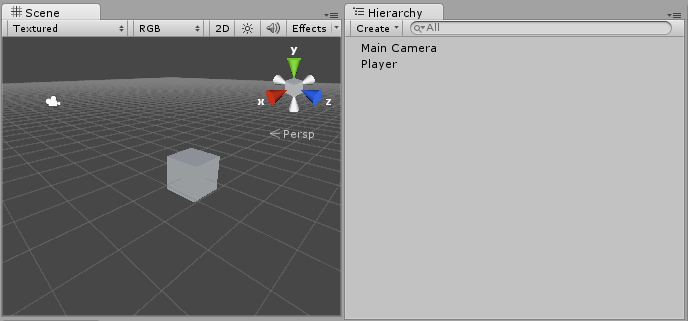
Програмният модел на Unity е базиран върху софтуерната техника Entity-component-system (ECS). Това е техника, която използва композиция вместо наследяване за изгражданете на високо-производителни кодови архитектури, със структура, подобна на релационни бази данни и/или функционално програмиране. Този тип дизайн се използва главно при проектирането на игри от 2000 г. насам. Базиран е на три основни концепта:

1. Entity (обект): Обект с обща цел. Обикновено се състои само от един уникален идентификационен номер.
2. Component (компонент): Съдържа данните за един аспект на обекта, и как той си взаимодейства със света. Реализациите обикновено използват структури, класове, или асоциативните масиви.
3. System (система): Всяка система работи непрекъснато (като че всяка система има своя собствена нишка) и изпълнява глобални действия върху всяко Entity, което притежава Component със същия аспект като тази система.

Примерно Entity от Unity е показано на фиг13. Уникалният идентификатор в случая е „Player”. Kомпонентите на това Entity са: Transform, Cube, Mesh Renderer, Character Controller, Player Controller, Weapon, фиг. 13Movement Motor. Mesh Renderer и Cube (Mesh Filter) се използват от системата за изрисуване на 3D модели (чрез тях се реализира графичното представяне на този игрови обект). Скриптовете са част от системата за контрол на поведението (логиката определяща състоянието) на това Entity. Това се осъществява като класът дефиниран в един скрипт наследява класa MonoBehaviour дефинирaн в UnityEngine namespace. В MonoBehaviour са декларирани множество от виртуални event функции. Тези, които се използват от наследяващият скрипт, трябва да се предефинират в него. На фиг. 14 са показани всички такива event функции и редът, в който се изъпняват

 фиг. 14

Всички тези обекти се разполагат в т.нар. Unity сцена. Тя представлява тримерно пространство зададено чрез правоъгълна координатна система с определено начало. Както се забелязва от фиг. 6 показаното entity притежава компонент наречен Transform. Той определя положението на обекта в сцената, като задава трансформацията му спрямо началото на координатната система, ротацията и мащабирането спрямо собствената координатна система на обекта. На фиг. 15 е показана сцена съдържаща игровият обект с идентификатор „Player” и главната камера, както и иерархията от обекти.

 фиг. 15

* 1. *Управление на състоянието на играта*
     1. Основна концепция

**Класът StateManager –** Статичен клас, който пази информация за текущото активно състояние на играта (описано по-долу).

**Класът** **GameManager -** Основен клас в играта, наследяващ MonoBehaviour и инициализиран в първото ниво, като е поставен върху игрови обект със същото име, като е нагласен по такъв начин, че да не се изтрива при зареждане на следващите нива. Съдържа информация необходима за множество други класове в играта, като например заготовката за изрисуване на Joystick при платформа Android. Предефинира методите наследени от MonoBehaviour: Update() и OnGUI(), които извикват съответно GameState.DoUpdate() и GameState.OnGUI(). Всеки клас от играта може да достъпи GameManager класа по следния лесен начин: GameObject.Find(“GameManager”).GetComponent<GameManager>();

GameObject представлява статичен клас дефиниран в UnityEngine namespace.

**Класът GameState –** Представянето на текущото състоянието в играта се използва архитектура от класове, които са базирани върху Singleton дизайн принципа. Всяко едно състояние е представено чрез отделен singleton клас, наследяващ от абстрактния клас GameState, който съдържа следната функционалност:

public abstract void DoUpdate (); - Извиква се веднъж за всеки кадър на играта и се използва за контрол на състоянието

public abstract void DrawGUI(); - Извиква се няколко пъти за всеки кадър и се грижи за изобразяване и управление на графичния интерфейс за всяко състояние

public abstract void EnterState(); - Използва се при вход към едно състояние при нужда от специфични преходни ефекти

public abstract void LeaveState(); - Използва се при изход от дадено състояние при нужда от специфични преходни ефекти

За следене на текющото състояние се използва статичен клас StateManager, който съдържа инстанция на текущата имплементация на GameState.

public static void DoUpdate () ; - Извиква имплементацията на DoUpdate() на текущото състояние

public static void DrawGUI() ; - Извиква имплементацията на DrawGUI() на текущото състояние.

public static void ChangeState(GameState newState) ; - Използва се за промяна от текущото състояние към newState. Извиква LeaveState() текущото състояние, и EnterState() на новото състояние.

**Класът PlayState –** наследява GameState и се използва за регулиране на част от логиката на играта. В себе си пази референция към локалния играч, както и референции към локалните обекти на всички отдалечени клиенти.

void PlayState.Update() – следи дали играча е мъртъв, и ако е, стартира таймер. При изстичане на определено време съживява играча.

public override void EnterState() – инициализира връзката към сървъра, или стартира свъръра, в зависимост от това дали PlayState е настроен като сървър или клиент (определя се чрез булевата променлива isServer), съответно чрез NetworkManager.StartServer(int maxConnections, string currentLevel) и NetworkManager.JoinServer(string host)

public override void LeaveState() – спира връзкъта към сървъра, или самия сървър, използвайки NetworkManager.Disconnect();

* 1. *Управление на графичния аватар на играча*

**Игови обект Player** – aватарът на играча представлява игрови обект с прикачени графичен модел и други компоненти, които включват: WeaponManager, RigidBody, PlayerController, MovementMotor, HealthManager, PlayerAnimation, Weapon. Всички от които наследяват MonoBehaviour.

**Класът** **WeaponManager** - пази информация за текущото оръжие (Weapon), и методи, чрез които го управлява (смяна на оръжие, начало на стрелба, край на стрелба и презареждане на оръжието):

void Update() - следи дали е натиснато копче за смяна на оръжието и го сменя, като го предава на PlayState класа, за да може да бъде изобразен интерфейса за оръжията.

**Компонентът** **RigidBody –** дава на Player обекта характеристики на физичен обект (такъв, върху който действат физични явления)

**Класът** **MovementMotor** се използва за изчисляване на скоростта на аватара и ротацията и прилага съответната сила върху RigidBody компонента

**Класът PlayerAnmiation –** използва се за анимиране на играча

**Класът** **PlayerController** събира входните команди на потребителите в Updatе() метода и наглася състоянията на класовете MovementMotor и WeaponManager, също така променя положението на камерата, в зависимост от това на къде се е прицелил играчът. Съдържа променливи, които пазят броя на убииствата, броя на умиранията и името на текущия играч. С цел поддържане на вход от множество платформи се използва Factory дизайн принципа: интерфейсът InputManagerIF декларира методи за проверка дали бутон изпълняващ дадена функция е натиснат. Този интерфейс се имплементира от MobileInputManager, PCInputManager и ConsoleInputManager (съответно за мобилни устройства, персонални компютри и конзоли). Тези класове от своя страна имплементират методи на интерфейса в зависимост от това кой бутон на каква функция съответства. В PlayerController класа е декларирано поле, което съдържа референция към имплементация на InputManagerIF. Тази имплементация се взима чрез статичнo извикване на InputManagerFactory.getInputManager() (което връща обект от тип InputManagerIF) в предефинираната Start().InputManagerFactory.getInputManager() използва предефинирани в Unity макрота, които определят за каква платформа е текущата компилация и благодарение на тях се генерира код, който конструира съответната имплементация на InputManagerIF. Съответната InputManagerIF имплементация се грижи за изобразяване на допълнителени, виртуални устройства за вход върху екрана.

**Класът** **Health** – се грижи за текущото здраве на играча, като използва методите OnDamage(float amount) ;

**Класът Weapon –**  представлява текущото оръжие. Стрелбата се извършва в Weapon.Update(), като за реализацията и се използва RayCast функция, генерираща лъч от цевта на оръжието до световните координати на курсора на мишката (при персонални компютри) или в посока, съвпадаща с ротацията на едно от лостчетата на джойстика (в зависимост от настройките). Удар се засича с най-близкия предмет, през който минава лъчът.

public void StartFire() **–** стартира стрелба;

public void StopFire() **–** спира стрелба;

public void Reload() **–** стартира презареждането;

* 1. *Реализация на графичния интерфейс*
     1. Менюта

Менютата се управляват чрез класове, наследяващи GameState, като всички бутони на отделните менюта се изрисуват в съответната имплементация на GameState.DrawGUI() метода.

**Класът MainMenuState** – представлява началното меню на играта.

**Класът CreateServerMenuState** – меню за създаване на сървър.

**Класът JoinServerMenuState** – меню за връзка към сървър

**Класът OptionsMenuState** – дава избор от опции за настройки на играта, като от тях в момента е имплементирано само менюто за настройки на играч.

**Класът PlayerSettingsMenuState** – настройки за аватара на играча

Полезното на тази архитектура е, че за добавяне на новo меню просто трябва да се дефинира нов клас, който наследява GameState(), без да се налага промяна в старата функционалност.

* + 1. Интерфейс по време на игра

Графичният интерфейс по време на игра (заредено ниво) се реализира чрез дефинираната в класa PlayState функция – DrawGUI(). PlayState репрезентира състоянието на игра и наследява GameState. DoUpdate() се използва за засичане на входни команди, като при засичане на команда DrawGUI() изрисува съответния интерфейс върху екрана. В PlayState има поле, което съдържа информация за текущия играч, както и масив, който съдържа информация за всички отдалечени клиенти. При изрисуване на интерфейсите показващи точките здраве, текущото оръжие и броя на муниции се използва информация от полето за текущ играч. При изрисуване на интерфейса показващ текущите убииства/умирания и имена на всички играчи се използва информация от масива за играчите, както и от полето за текущия играч. Полето и масива съдържат обекти от тип PlayerController, които от своя страна притежават свойства за необходимата информация.

* 1. *Реализация на сървър и клиент*
     1. Методи за сървърна реализация в Unity

Сървърът реализиран на този етап представлява не-авторитарен сървър (описан в точка 2.2.4). Осъществява се чрез вградената в Unity мрежова функционалност – статичният клас Network. За стартиране на сървър се използва Network.InitializeServer(int connections, int listenPort, bool useNat), а за връзка към стартиран – Network.Connect(string IP, int remotePort, string password = ""). За комуникация между отделните клиенти и сървъра се използва специален компонент дефиниран в Unity – NetworkView. Всяко NetworkView се идентифицира с уникален номер за една инстанция на играта. За да се осъществи връзката между NetworkView на локалната игра и NetworkView на отдалечен клиент или сървър, те трябва да имат еднакви номера. То предоставя три начина за трансфер на информация, като и трите използват UDP протокол:

1. Remote Procedure Calls – представлява извкиване на функция, която е дефинирана на отдалечената инстанция на играта. Извършва се чрез NetworView.RPC(string name, RPCMode mode, params object[] args), където name е името на отдалечената функция, която трябва да е дефинирана в скрипт на обект, който притежава компонент NetworkView. Отдалечени функции, които могат да се извикват по подобен начин се обозначават с [RPC] анотацията
2. Сериализация – извършва се чрез имплементация на метода OnSerializeNetworkView(BitStream stream, NetworkMessageInfo messageInfo), които се извиква 15 пъти в секунда. Имплементацията трябва да бъде направена в скрипт на игрови обект, който притежава компонент NetworkView. Чрез писане в stream се осъществява пращане на информация към отдалечените NetworkView с еднакъв идентификатор, докато с четене от stream се получава информацията
3. Наблюдаване - NetworkView наблюдава компонента за трансформация или анимация на обекта, към когото е прикачен и ако настъпят промени ги праща на всички отдалечени NetworkView компоненти с еднакъв номер, които от своя страна прилагат промените върху обектите, към които са прикачени.
   * 1. Имплементация

**Класът** **NetworkManager** – основният клас, който се грижи за управлението на връзката и синхронизирането на обектите, прикачен към GameManager игровият обек, с добавен NetworkView към него.:

public void StartServer(int maxConnections, string currentLevel) – стартира сървър със съответно максимален брой играчи и избраното ниво

public void JoinServer(string host) – връзва се към свърър с хост – host

public void Disconnect() – прекъсва връзката (може да се използва както от страна на сървър така и от страна на клиент)

[RPC]

void LoadLevel (string level, int levelPrefix) – remote procedure call за зареждане на нивото (идеята е да се изпълни върху всички текущи и бъдещи клиенти на съвръра.

public void SpawnPlayer() – инициализира аватара на играча (игровият обект Player) върху сървъра и всички клиенти (както текущи така и бъдещи)

public void RespawnPlayer() – съживява играча, след като е умрял. В локалната игра.

[RPC]

void RespawnAll(NetworkPlayer owner) – remote procedure call, който съживява играча върху всички други клиенти и сървъра

**Игровият oбект** Player – за да се синхронизират всички инстанцирани (чрез SpawnPlayer()) в отделните клиенти и сървъра обекти на аватара , към него се добавя компонент NetworkView. То се използва за предаване на движението, промяната на кръвта, промяната на името, стрелбата, дали играча е убил и дали е умрял по мрежата.

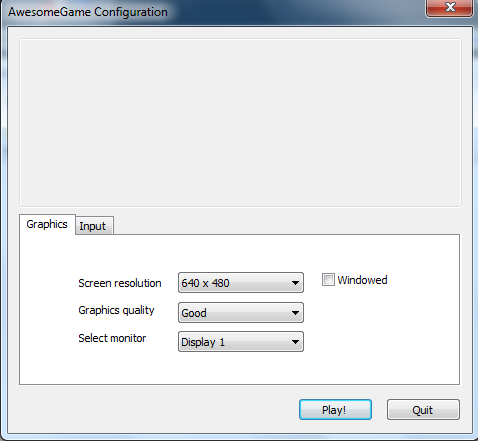
1. Движение – използва се функцията OnSerializeNetworkView(BitStream stream, NetworkMessageInfo messageInfo) дефинирана в MovementMotor класа. Когато е извикана в локалната инстанция на играта записва текущото положение на обекта в stream променливата, премества обекта (в MovementMotor.FixedUpdate()) и след това записва новото положение в stream променливата. Когато OnSerializeNetworkView(BitStream stream, NetworkMessageInfo messageInfo) се извика от инстанция на обекта намираща се на отдалечен клиент се прочитат двете положения от stream променливата и се прави линеарна интерполация между тях в съответния MovementMotor.Update().
2. Стрелбата – използва се OnSerializeNetworkView(BitStream stream, NetworkMessageInfo messageInfo) в Weapon класа, за да предаде (съответно да получи) дали е стартирана стрелбата, дали е спряна или е натиснат бутон за презареждане.
3. Промяна на кръвта – Health.ОnDamage(float amount) извиква чрез remote procedure call функцията DoDamageRemote (float amount), която намалява кръвта на всички отдалечени инстанции на аватара на играча. OnDamage(float amount) се извиква, когато в Weapon.Update() се засече удар.
4. Смяна на име – чрез метода PlayerController.ChangeName(string newName)
5. Добавяне на убииство и смърт – използват се методите

public void PlayerController.KillPlayer() – убива играча, върху който е извикан (прави Player игровия обект не-активен), и увеличава броя умирания с 1.

Public void PlayerController.AddKill() – добавя едно убииство.

1. **Ръководство за използване**
   1. *Работа с менютата*
      1. Стартово меню на Unity (само при персонални компютри)

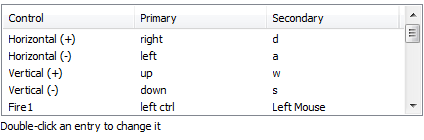
Когато играта се стартира върху персонален компютър, първото меню, което се появява е стартовото меню на unity, което позволява начална настройка на графиката и входните команди на играта (фиг. 16)



фиг. 16

Както се вижда, предлага избор от 2 под-менюта – Graphics и Input. От първото могат да се изберат резолюция, която започва от 640 х 480 и стига до максимално достижимата от екрана. Качество на графиката и екран, върху който да бъде стартирана играта. Ако не е селектиран бутона Windowed играта няма да се стартира на пълния екран.

Input под-менюто е показано на фиг. 17.



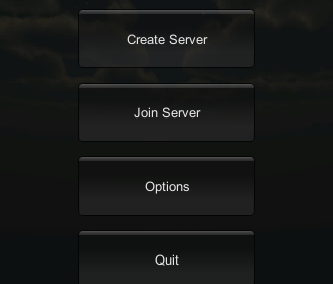
фиг. 17

Позволява се съпоставяне на една команда към два клавиша – Primary и Secondary. Horizontal(+) представлява движение на дясно, Horizontal(-) – на ляво, Vertical(+) движение напред/нагоре, Vertical(-) движение надолу. Всички други бутони съвпадат с имената си, например Fire1 – главна стрелба.

При натискане на Play бутона от фиг 16. стартира същинската част от играта (при платформи различни от персонален компютър се пропуска точка 5.1.1

* + 1. Главно Меню на играта

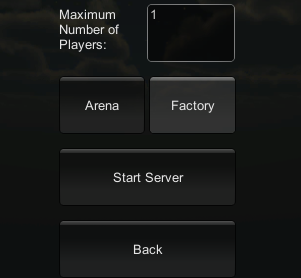
Главното меню може да се види на фиг. 18. Представлява съвкупност от бутони, като всеки един бутон препраща към едно от под-менютата в играта. При натискане на един от бутоните, се скрива предходното меню и се появява новото.



фиг.18

* *Create Server*

При натискане на Create Server бутона играта препраща към под-менюто за създаване на нов сървър изобразено на фиг.19



фиг. 19

В текстовото поле обозначено с “Maximum Number of Players” се задава максималния брой на играчите, който ще могат да играят на сървъра (към този брой вилза и човека, който създава сървъра). По подразбиране е равно на 1;

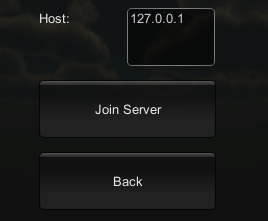
Под него са разположени два радио бутона наречени “Arena” и “Factory”. Те представляват нивата, от които потребителя може да избира за игра. При избор на единия от бутоните, той се осветава (на фиг.14 това е Factory).

Бутонът Start Server стартира сървър с избраните максимални играчи и зарежда избраното ниво.

Бутонът Back връща към предходното меню.

* *Join Server*

При натискане на Join Server бутона играта препраща към под-менюто за създаване на нов сървър изобразено на фиг.20



фиг. 20

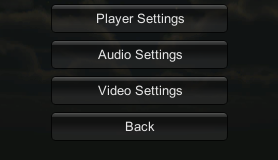
В текстовото поле обозначено с “Host” се попълват хостът и портът на сървъра, към който потребителят иска да се върже, в формат: host:port. Ако стойност за порт не бъде попълнена се използва такава по подразбиране – 12045.

При натискане на Join Server бутона се осъществява връзка със сървара и започва зареждането на избраното в сървъра ниво.

Бутонът Back връща предходното меню.

* *Options*

При натискане на Options бутона играта препраща към под-менюто настройки на играта изобразено на фиг.21

**

фиг. 21

На този етап функционира само бутонът Player Settings. За имплементация на другите е нужен външен софтуер от Unity Asset Store

Бутонът Player Settings препраща към меню за настройки на играча (фиг. 22) , които на този етап включват само промяна на името му.



Фиг 22.

Аpply бутонът трябва да бъде натиснат за да се запази промяната. Името на играча се пази в определен файл, който е различен за различните операционни системи

* Mac OS X – ~/Library/Preferences/unity.PlamenKokanov.AwesomeGame.plist
* Android – /data/data/appname/shared\_prefs/ unity.PlamenKokanov.AwesomeGame.xml
* Windows – в регистрито под HKCU\Software\PlamenKokanov\AwesomeGame
* Linux – ~/.config/unity3d/ PlamenKokanov/AwesomeGame
  + 1. Работа с менюто по време на игра

След стартиране на нивото, чрез натискане на клавиша Esc при платформи с клавиатура, или back при мобилни устройства и конзоли може да се покаже допълнително меню (показано на фиг. 23)



фиг. 23

При натискане на бутона Resume или бутна натиснат за пускане на менюто, то се скрива.

Бутонът Settings отваря допълнително меню, от където може директно да се смени името на играча (фиг.24)



фиг. 24

Това меню действа по същия начин както под-менюто за настройки от главното меню

Бутонът Scores показва текущия резултат във формат име на играч, брой убииства, брой умирания (фиг. 25). При натискането му автоматично се скриват всички досега показани менюта.



фиг. 25

За да се затвори таблицата с резултати се натиска същият бутон, който се ползва за пускане на менюто в игра, но този път самото меню не се показва.

* 1. *Начин на игра след зареждане на ниво*
     1. Управление на тримерния аватар на играча

След зареждане на избраното от потребителя направил сървъра ниво, на играча се дава контрол върху тримерен аватар (фиг. 26), който се създава на произволно място в нивото.



фиг. 26

Камерата на играча се наглася над модела и сочеща към него, така че се получава картина като тази на фиг.23. Този аватар може да бъде местен напред, назад, наляво и надясно със съответните клавиши (нагласени в стартовото меню) или в зависимост от положението на лявото лостче на джойстика при мобилни устройства и конзоли. Моделът може да се върти около себе си, като винаги тялото и оръжието са насочени към положението на курсора при наличие на мишка. Когато се използва джойстик моделът се завърта спрямо положението на десния лост на джойстика.

* + 1. Стрелба , муниции и точки здраве

Стрелбата се извършва при натискане на бутона за стрелба на съответната платформа: за компютри – в зависимост от настройките на стартовото меню (левия бутон на мишката по подразбиране) ; за мобилни приложения – при задържане на десния виртуален лост; за Xbox360 – бутон „А“; за PlayStation – бутон триъгълник. Посоката на стрелба се определя от лазераният мерник на оръжието.

Както се вижда от фиг. 23 в горния ляв ъгъл се намира интерфейс, който показва текущатите точки здраве (зелен правоъгълник, в който са зададени максималната и текущата кръв във формата <текущи точки>/<максимални точки>) и броя на мунициите оставащи в оръжието, както и броя на патроните в текущия пълнител (изобразени са върху прозрачен петоъгълник, като форматът е <всички оставащи патрони>|<патрони оставащи в текущия пълнител>). Когато броят на патроните в пълнителя стане равен на 0, се налага оръжието да бъде презаредено. Това става или като се натисне бутона за презареждане, или автоматично при опит за стрелба с празно оръжие.

Максималните точки здраве на всеки играч са 100, и те намаляват всеки път, когат играчът бъде уцелен. Зеленият правоъгълник, чрез който са изобразени намаля дължината си с намалянето на текущите точки здраве.

Когато кръвта на един играч стане 0, той умира и към текущия му резултат се добавя една смърт, след 5 секунди се съживява на случайно място в нивото. Убииствата на човекът, който го е свалил до 0 точки здраве, се увеличават с 1.

Крайната цел на играта е да се направят възможно най-много убииства и да се умре възможно най-малко пъти.

1. **Заключение**

**Текущо състояние на дипломната работа:**

Играта от текущата разработка изпълнява зададените основни цели – поддържа множество операционни системи; позволява игра в мрежа от няколко играча едновременно, като дори е възможно те да употребяват различни платформи; предоставя менюта, които са лесни за употреба и не затормозяват потребителя с твърде много опции струпани на едно място; с помощта Unity предлага естетически приятна тримерна графика; архитектурата на програмната реализация позволява лесно добавяне на функционалност, като например: нови игрови състояния, добавяне на нови модели за оръжия и аватари, нови нива и поддръжка за допълнителни платформи.

**Насоки за бъдещо развитие:**

Препоръчва се направата на централизиран авторитарен сървър с цел предотвратяване на хакери и популяризиране на играта. Възможно е дори връзка с база данни, която да пази информацията за всеки играч, като например точките, които е направил в последните няколко игри. Това позволява на играчите да следят по-лесно дали стават по-добри или не и също така да сравняват своите резултати с тези на противниците/приятелите си.

1. **Използвана литература**
2. <http://docs.unity3d.com/Manual/net-HighLevelOverview.html>
3. <http://en.wikipedia.org/wiki/Unity_(game_engine)>
4. <http://en.wikipedia.org/wiki/OpenGL>
5. <http://en.wikipedia.org/wiki/Game_engine>
6. <http://en.wikipedia.org/wiki/Video_game>
7. <http://en.wikipedia.org/wiki/Blender_(software)>
8. <http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking>
9. <http://developer.valvesoftware.com/wiki/Lag_Compensation>
10. <http://developer.valvesoftware.com/wiki/Working_With_Prediction>
11. <http://www.theesa.com/facts/pdfs/esa_ef_2014.pdf>
12. <http://www.bigfishgames.com/blog/2014-global-gaming-stats-whos-playing-what-and-why/>
13. <http://forrest.technicat.com/games/unity.pdf>
14. Unity Multiplayer Games by Alan R. Stagner
15. **Приложение 1 – извадка от сорс кода на разработката**

**GameManager.cs**

using UnityEngine;

using System.Collections;

public class GameManager : MonoBehaviour {

public string[] levelNames;

public Texture2D loadingScreen;

public GameObject joystickPrefab;

public GameObject cursorPrefab;

public float respawnTime;

void Start() {

DontDestroyOnLoad(this);

Application.LoadLevel ("MainMenuScene");

}

void Update () {

StateManager.DoUpdate ();

}

void OnGUI() {

StateManager.DrawGUI();

}

}

**StateManager.cs:**

using UnityEngine;

using System.Collections;

public static class StateManager {

private static GameState gameState = MainMenuState.getInstance();

public static void DoUpdate () {

gameState.DoUpdate ();

}

public static void DrawGUI() {

gameState.DrawGUI();

}

public static void ChangeState(GameState newState) {

gameState.LeaveState();

gameState = newState;

gameState.EnterState();

}

}

**GameState.cs**

using UnityEngine;

using System.Collections;

public abstract class GameState {

protected int firstButtonPosY;

protected int firstButtonPosX;

protected int buttonHeight;

protected int buttonWidth;

protected int buttonOffset;

protected GameObject gameManager;

protected GameState() {

firstButtonPosY = Screen.height/20;

firstButtonPosX = Screen.width/3;

buttonHeight = Screen.height/5;

buttonWidth = Screen.width/3;

buttonOffset = Screen.height/4;

gameManager = GameObject.Find("GameManager");

}

public abstract void DoUpdate ()

public abstract void DrawGUI();

public abstract void EnterState();

public abstract void LeaveState();

}

**PlayState.cs:**

using UnityEngine;

using System.Collections;

using System.Collections.Generic;

public class PlayState : GameState {

public static string CURRENT\_LEVEL\_NOT\_SET\_ERROR = "Cannot initialize PlayState. Current level not set!";

public static string CURRENT\_HOST\_NOT\_SET\_ERROR = "Cannot initialize PlayState. Current host not set!";

public bool showMenu;

public bool showScore;

public bool isChatting;

public HudWeapons hudWeapons;

public HudHealth hudHealth;

public float respawnTime;

private PlayerController playerController;

private List<PlayerController> enemyPlayers;

private string currentLevel;

private Rect menuBox;

private Rect playerNameLabel;

private Rect playerNameTextField;

private Rect applyButton;

private Rect closeButton;

private Rect menuExtensionBox;

private Rect resumeGameButton;

private Rect playerSettingsButton;

private Rect showScoresButton;

private Rect mainMenuButton;

private Rect respawnLabelRect;

private Rect scoresRect;

private Texture2D loadingScreen;

private Rect exitGameButton;

private static PlayState instance;

private int maxPlayers;

private NetworkManager networkManager;

private bool isServer;

private string currentHost;

private string playerNameString;

private bool drawSettingsMenu;

private bool loading;

private Rect loadingScreenRect;

public PlayerController PlayerController {

get {

return playerController;

}

set {

playerController = value;

}

}

public List<PlayerController> EnemyPlayers {

get {

return enemyPlayers;

}

set {

enemyPlayers = value;

}

}

public string CurrentLevel {

get {

return currentLevel;

}

set {

currentLevel = value;

}

}

public int MaxPlayers {

get {

return maxPlayers;

}

set {

maxPlayers = value;

}

}

public bool IsServer {

get {

return isServer;

}

set {

isServer = value;

}

}

public string CurrentHost {

get {

return currentHost;

}

set {

currentHost = value;

}

}

public static PlayState getInstance () {

if(instance == null)

instance = new PlayState();

return instance;

}

private PlayState () :base() {

enemyPlayers = new List<PlayerController>();

menuBox = new Rect(5, Screen.height/3, Screen.width/5, Screen.height/2);

resumeGameButton = new Rect(menuBox.x, menuBox.y+menuBox.height/6, menuBox.width - 1, menuBox.height/6);

playerSettingsButton = new Rect(menuBox.x, menuBox.y+2\*menuBox.height/6, menuBox.width - 1, menuBox.height/6);

showScoresButton = new Rect(menuBox.x, menuBox.y+3\*menuBox.height/6, menuBox.width - 1, menuBox.height/6);

mainMenuButton = new Rect(menuBox.x, menuBox.y+4\*menuBox.height/6, menuBox.width - 1, menuBox.height/6);

exitGameButton = new Rect(menuBox.x, menuBox.y+5\*menuBox.height/6, menuBox.width - 1, menuBox.height/6);

respawnLabelRect = new Rect(Screen.height/3, Screen.width/3, Screen.height/3, Screen.width/3);

menuExtensionBox = new Rect(5 + Screen.width/5, menuBox.y, Screen.width/5, Screen.height/2);

playerNameLabel = new Rect(menuExtensionBox.x, menuExtensionBox.y+menuExtensionBox.height/4, menuExtensionBox.width - 1, menuExtensionBox.height/8);

playerNameTextField = new Rect(menuExtensionBox.x, menuExtensionBox.y+menuExtensionBox.height\*(3f/8f), menuExtensionBox.width - 1, menuExtensionBox.height/8);

applyButton = new Rect(menuExtensionBox.x + 1, menuExtensionBox.y+2.2f\*menuExtensionBox.height/4, menuExtensionBox.width - 2, menuExtensionBox.height/6);

closeButton = new Rect(menuExtensionBox.x + 1, menuExtensionBox.y+3\*menuExtensionBox.height/4, menuExtensionBox.width - 2, menuExtensionBox.height/6);

loadingScreenRect = new Rect(0, 0, Screen.width, Screen.height);

scoresRect = new Rect(Screen.width/4, Screen.height/4, Screen.width/1.5f, Screen.height/1.5f);

showMenu = false;

loadingScreen = gameManager.GetComponent<GameManager>().loadingScreen;

networkManager = gameManager.GetComponent<NetworkManager>();

hudHealth = gameManager.GetComponent<HudHealth>();

hudWeapons = gameManager.GetComponent<HudWeapons>();

respawnTime = gameManager.GetComponent<GameManager>().respawnTime;

}

public override void DrawGUI() {

if(!loading) {

if(showMenu) {

GUI.Box (menuBox, "Menu");

if(GUI.Button (resumeGameButton, "Resume")) {

showMenu = false;

} else if(GUI.Button (showScoresButton, "Scores")) {

showScore = true;

showMenu = false;

} else if (GUI.Button (playerSettingsButton, "Settings")){

drawSettingsMenu = true;

} else if (GUI.Button (mainMenuButton, "Main Menu")) {

StateManager.ChangeState(MainMenuState.getInstance ());

} else if (GUI.Button (exitGameButton, "Exit Game")) {

Application.Quit();

}

}

if(drawSettingsMenu) {

GUI.Box (menuExtensionBox, "Settings");

GUI.Label (playerNameLabel, "Player Name:");

playerNameString = PlayerPrefs.GetString("playerName");

playerNameString = GUI.TextField (playerNameTextField, playerNameString);

if (GUI.Button (applyButton, "Apply")) {

PlayerPrefs.SetString("playerName", playerNameString);

PlayerPrefs.Save();

playerController.ChangeName(playerNameString);

} else if (GUI.Button (closeButton, "Close")) {

drawSettingsMenu = false;

}

}

if(showScore) {

GUILayoutOption width = GUILayout.Width(scoresRect.width/3.2f);

GUILayout.BeginArea (scoresRect);

GUILayout.BeginVertical();

GUILayout.BeginHorizontal ();

GUILayout.Box("Name", width);

GUILayout.Box("Kills", width);

GUILayout.Box("Deaths", width);

GUILayout.EndHorizontal();

GUILayout.BeginHorizontal ();

GUILayout.Box(playerController.playerName, width);

GUILayout.Box(playerController.Kills.ToString(), width);

GUILayout.Box(playerController.Deaths.ToString(), width);

GUILayout.EndHorizontal();

foreach(PlayerController enemy in enemyPlayers) {

GUILayout.BeginHorizontal ();

GUILayout.Box(enemy.playerName, width);

GUILayout.Box(enemy.Kills.ToString(), width);

GUILayout.Box(enemy.Deaths.ToString(), width);

GUILayout.EndHorizontal();

}

GUILayout.EndVertical ();

GUILayout.EndArea();

}

if(hudHealth.enabled) {

hudHealth.CurrentHealth = playerController.health.CurrentHealth;

hudHealth.MaxHealth = playerController.health.maxHealth;

hudHealth.DrawGUI ();

}

if(hudWeapons.enabled) {

hudWeapons.currentWeapon = playerController.weaponManager.currentWeapon;

hudWeapons.DrawGUI();

}

if(playerController.Dead) {

GUI.Label (respawnLabelRect, "Respawning in " + Mathf.CeilToInt(respawnTime));

}

}

else {

GUI.DrawTexture(loadingScreenRect, loadingScreen);

}

}

public override void DoUpdate() {

if(!loading) {

if(Input.GetKeyDown ("escape")) {

if(showScore) {

showScore = false;

} else if (drawSettingsMenu){

drawSettingsMenu = false;

} else {

showMenu = !showMenu;

}

}

if(playerController.health.currentHealth <= 0){

playerController.Dead = true;

respawnTime -= Time.deltaTime;

if(respawnTime <= 0) {

networkManager.RespawnPlayer();

playerController.Dead = false;

respawnTime = gameManager.GetComponent<GameManager>().respawnTime;

}

}

}

}

public void Reset() {

showMenu = false;

showScore = false;

currentLevel = null;

playerController = null;

enemyPlayers.Clear();

}

public override void EnterState() {

if(isServer) {

if(currentLevel != null && currentLevel != "") {

loading = true;

networkManager.StartServer(maxPlayers, currentLevel);

} else {

StateManager.ChangeState (CreateServerMenuState.getInstance ());

Debug.LogError(CURRENT\_LEVEL\_NOT\_SET\_ERROR);

}

} else {

if(currentHost != null && currentHost != "") {

loading = true;

networkManager.JoinServer(currentHost);

} else {

Debug.LogError(CURRENT\_HOST\_NOT\_SET\_ERROR);

StateManager.ChangeState (JoinServerMenuState.getInstance ());

}

}

}

public override void LeaveState() {

hudHealth.enabled = false;

hudWeapons.enabled = false;

networkManager.Disconnect();

Screen.showCursor = true;

}

public void OnNetworkLoadedLevel() {

loading = false;

hudHealth.enabled = true;

hudWeapons.enabled = true;

}

}

**MainMenuState.cs:**

using UnityEngine;

using System.Collections;

public class MainMenuState : GameState{

private Rect createServerButton;

private Rect joinServerButton;

private Rect optionsButton;

private Rect quitButton;

private static MainMenuState instance;

public static MainMenuState getInstance () {

if(instance == null)

instance = new MainMenuState();

return instance;

}

private MainMenuState() : base() {

createServerButton = new Rect(firstButtonPosX, firstButtonPosY, buttonWidth, buttonHeight);

joinServerButton = new Rect(firstButtonPosX, firstButtonPosY + buttonOffset, buttonWidth, buttonHeight);

optionsButton = new Rect(firstButtonPosX, firstButtonPosY + 2\*buttonOffset, buttonWidth, buttonHeight);

quitButton = new Rect(firstButtonPosX, firstButtonPosY + 3\*buttonOffset, buttonWidth, buttonHeight);

}

public override void DrawGUI() {

if(GUI.Button(createServerButton, "Create Server")) {

StateManager.ChangeState(CreateServerMenuState.getInstance ());

} else if (GUI.Button(joinServerButton, "Join Server")) {

StateManager.ChangeState(JoinServerMenuState.getInstance ());

} else if (GUI.Button(optionsButton, "Options")) {

StateManager.ChangeState(OptionsMenuState.getInstance ());

} else if (GUI.Button (quitButton, "Quit")) {

Application.Quit();

}

}

public override void DoUpdate() {

}

public override void EnterState() {

if(Application.loadedLevelName != "MainMenuScene") {

Application.LoadLevel ("MainMenuScene");

}

}

public override void LeaveState() {

}

}

**HudHealth.cs**

using UnityEngine;

using System.Collections;

public class HudHealth : MonoBehaviour {

public Texture2D fullHealth;

public Texture2D emptyHealth;

public GUIStyle currentHealthStyle;

private float maxHealth;

private float currentHealth;

private Rect background;

private Rect healthRect;

private Rect backgroundBox;

private Rect healthLabel;

public float CurrentHealth {

get {

return currentHealth;

}

set {

currentHealth = value;

}

}

public float MaxHealth {

get {

return maxHealth;

}

set {

maxHealth = value;

}

}

void Start() {

background = new Rect(5, 5, Screen.width/5, Screen.height/15);

healthRect = new Rect(5, 5, background.width, background.height);

healthLabel = new Rect(background.width/3, 5 + Screen.width/110, background.width/3, background.height);

}

public void DrawGUI() {

float healthPercent = currentHealth/maxHealth;

healthRect.width = healthPercent\*(background.width - 2);

GUI.DrawTexture(background, emptyHealth, ScaleMode.StretchToFill);

GUI.DrawTexture (healthRect, fullHealth, ScaleMode.StretchToFill);

GUI.Label (healthLabel, currentHealth.ToString () + "/" + maxHealth.ToString (), currentHealthStyle);

}

}

**HudWeapons.cs**

using UnityEngine;

using System.Collections;

public class HudWeapons : MonoBehaviour {

public Weapon currentWeapon;

public GUIStyle totalAmmoStyle;

public GUIStyle ammoRemainingStyle;

private Vector2 startCorner;

private float alphaAmmo;

private Color auxColor;

private Color cColor;

private int currentAmmo;

void Start() {

alphaAmmo = 1.0f;

}

public void DrawGUI() {

auxColor = cColor = GUI.color;

startCorner = new Vector2(Screen.width, Screen.height) - new Vector2(5, (Screen.height-45));

ShowAmmunition();

GUI.color = cColor;

}

void ShowAmmunition() {

auxColor.a = alphaAmmo;

GUI.color = auxColor;

GUI.DrawTexture(currentWeapon.ammunitionRect, currentWeapon.ammunitionBackground);

float delta = Mathf.Clamp(currentWeapon.ClipAmmoAmount, 0, currentWeapon.clipSize);

delta /= currentWeapon.clipSize;

delta \*= currentWeapon.maxIcons;

int length = Mathf.CeilToInt(delta);

for(int i = 0; i < length; i++) {

GUI.DrawTexture(new Rect(currentWeapon.ammunitionRect.x + 40 + (i \* (currentWeapon.ammunitions.width - 1)), currentWeapon.ammunitionRect.y + 28, currentWeapon.ammunitions.width, currentWeapon.ammunitions.height), currentWeapon.ammunitions);

}

var auxRect = new Rect(currentWeapon.ammunitionRect.x + 40, currentWeapon.ammunitionRect.y + 2, 20, 20);

GUI.Label(auxRect, currentWeapon.AmmoAmount.ToString(), totalAmmoStyle);

auxRect.x += 23;

auxRect.y -= 1;

GUI.Label(auxRect, "|", totalAmmoStyle);

auxRect.x += 5;

auxRect.y += 4;

GUI.Label(auxRect, currentWeapon.ClipAmmoAmount.ToString(), ammoRemainingStyle);

}

}

**InputManagerFactory.cs**

using UnityEngine;

using System.Collections;

public class InputManagerFactory {

public static InputManagerIF GetInputManager(Transform playerTransform, Camera camera) {

#if UNITY\_IPHONE || UNITY\_ANDROID

return new MobileInputManager(playerTransform, camera);

#elif UNITY\_XBOX360 || UNITY\_PS3

return new ConsoleInputManager(playerTransform, camera);

#else

return new PCInputManager(playerTransform, camera);

#endif

}

}

**InputManagerIF.cs**

using UnityEngine;

using System.Collections;

public interface InputManagerIF {

Vector3 GetHorizontalAxis();

Vector3 GetVerticalAxis();

bool GetFireButtonDown();

bool GetFireButtonUp();

bool GetBackButtonPressed();

bool GetWalkButton();

bool GetReloadButton();

Vector3 GetFacingDirection();

Vector3 GetCameraAdjustment();

}

**PCInputManager.cs**

using UnityEngine;

using System.Collections;

public class PCInputManager : InputManagerIF {

private GameObject cursorObject;

private Transform playerTransform;

private Camera camera;

private Plane playerMovementPlane;

private float cursorPlaneHeight;

private Quaternion screenMovementSpace;

private Vector3 screenMovementForward;

private Vector3 screenMovementRight;

private Vector3 cameraAdjustmentVector;

public PCInputManager(Transform playerTransform, Camera camera) {

GameObject cursorPrefab = GameObject.Find ("GameManager").GetComponent<GameManager>().cursorPrefab;

cursorObject = (GameObject)GameObject.Instantiate (cursorPrefab);

this.playerTransform = playerTransform;

this.camera = camera;

cursorPlaneHeight = 0.15f;

playerMovementPlane = new Plane (playerTransform.up, playerTransform.position + playerTransform.up \* cursorPlaneHeight);

screenMovementSpace = Quaternion.Euler (0, camera.transform.eulerAngles.y, 0);

screenMovementForward = screenMovementSpace \* Vector3.forward;

screenMovementRight = screenMovementSpace \* Vector3.right;

}

public Vector3 GetHorizontalAxis() {

return Input.GetAxis ("Horizontal") \* screenMovementRight;

}

public Vector3 GetVerticalAxis(){

return Input.GetAxis ("Vertical") \* screenMovementForward;

}

public bool GetFireButtonDown(){

return Input.GetButtonDown("Fire1");

}

public bool GetFireButtonUp(){

return Input.GetButtonUp("Fire1");

}

public bool GetBackButtonPressed(){

return Input.GetKeyDown("escape");

}

public bool GetWalkButton() {

return Input.GetKey(KeyCode.LeftShift);

}

public bool GetReloadButton() {

return Input.GetKey (KeyCode.R);

}

public Vector3 GetCameraAdjustment() {

HandleCursorAlignment(ScreenPointToWorldPointOnPlane (Input.mousePosition, playerMovementPlane, camera));

Vector3 cursorScreenPosition = Input.mousePosition;

float halfWidth = Screen.width / 2.0f;

float halfHeight = Screen.height / 2.0f;

float maxHalf = Mathf.Max (halfWidth, halfHeight);

Vector3 posRel = cursorScreenPosition - new Vector3 (halfWidth, halfHeight, cursorScreenPosition.z);

posRel.x /= maxHalf;

posRel.y /= maxHalf;

cameraAdjustmentVector = posRel.x \* screenMovementRight + posRel.y \* screenMovementForward;

cameraAdjustmentVector.y = 0.0f;

return cameraAdjustmentVector;

}

public Vector3 GetFacingDirection() {

Vector3 cursorScreenPosition = Input.mousePosition;

playerMovementPlane.normal = playerTransform.up;

playerMovementPlane.distance = -playerTransform.position.y + cursorPlaneHeight;

Vector3 cursorWorldPosition = ScreenPointToWorldPointOnPlane (cursorScreenPosition, playerMovementPlane, camera);

return (cursorWorldPosition - playerTransform.position);

}

private static Vector3 PlaneRayIntersection (Plane plane, Ray ray) {

float dist;

plane.Raycast (ray, out dist);

return ray.GetPoint (dist);

}

private static Vector3 ScreenPointToWorldPointOnPlane (Vector3 screenPoint, Plane plane, Camera camera) {

Ray ray = camera.ScreenPointToRay (screenPoint);

return PlaneRayIntersection (plane, ray);

}

private void HandleCursorAlignment(Vector3 cursorWorldPosition) {

cursorObject.transform.position = cursorWorldPosition;

Screen.showCursor = (Input.mousePosition.x < 0 || Input.mousePosition.x > Screen.width || Input.mousePosition.y < 0 || Input.mousePosition.y > Screen.height);

Quaternion cursorWorldRotation = cursorObject.transform.rotation;

if (GetFacingDirection() != Vector3.zero)

cursorWorldRotation = Quaternion.LookRotation (GetFacingDirection());

Vector3 cursorScreenspaceDirection = Input.mousePosition - camera.WorldToScreenPoint (playerTransform.position + playerTransform.up \* cursorPlaneHeight);

cursorScreenspaceDirection.z = 0;

Quaternion cursorBillboardRotation = camera.transform.rotation \* Quaternion.LookRotation (cursorScreenspaceDirection, -Vector3.forward);

cursorObject.transform.rotation = Quaternion.Slerp (cursorWorldRotation, cursorBillboardRotation, 1);

float compensatedScale = 0.1f \* Vector3.Dot (cursorWorldPosition - camera.transform.position, camera.transform.forward);

float cursorScaleMultiplier = Mathf.Lerp (0.7f, 1.0f, Mathf.InverseLerp (0.5f, 4.0f, GetFacingDirection().magnitude));

cursorObject.transform.localScale = Vector3.one \* Mathf.Lerp (compensatedScale, 1, 0) \* cursorScaleMultiplier;

}

}

**MobileInputManager.cs:**

using UnityEngine;

using System.Collections;

public class MobileInputManager : InputManagerIF {

private GameObject joystickPrefab;

private Joystick joystickLeft;

private Joystick joystickRight;

private Transform playerTransform;

private Camera camera;

private Plane playerMovementPlane;

private float cursorPlaneHeight;

private Quaternion screenMovementSpace;

private Vector3 screenMovementForward;

private Vector3 screenMovementRight;

private Vector3 cameraAdjustmentVector;

public MobileInputManager(Transform playerTransform, Camera camera) {

joystickPrefab = GameObject.Find ("GameManager").GetComponent<GameManager>().joystickPrefab;

GameObject joystickLeftGO = (GameObject) GameObject.Instantiate (joystickPrefab);

joystickLeftGO.name = "Joystick Left";

joystickLeft = joystickLeftGO.GetComponent<Joystick> ();

GameObject joystickRightGO = (GameObject) GameObject.Instantiate (joystickPrefab);

joystickRightGO.name = "Joystick Right";

joystickRight = joystickRightGO.GetComponent<Joystick> ();

if(joystickRightGO!=null){

// Move to right side of screen

GUITexture guiTex = joystickRightGO.GetComponent<GUITexture> ();

guiTex.pixelInset = new Rect(Screen.width - guiTex.pixelInset.x - guiTex.pixelInset.width, guiTex.pixelInset.y, guiTex.pixelInset.width, guiTex.pixelInset.height);

}

this.playerTransform = playerTransform;

this.camera = camera;

cursorPlaneHeight = 0.15f;

playerMovementPlane = new Plane (playerTransform.up, playerTransform.position + playerTransform.up \* cursorPlaneHeight);

screenMovementSpace = Quaternion.Euler (0, camera.transform.eulerAngles.y, 0);

screenMovementForward = screenMovementSpace \* Vector3.forward;

screenMovementRight = screenMovementSpace \* Vector3.right;

}

public Vector3 GetHorizontalAxis(){

return joystickLeft.position.x\*screenMovementRight;

}

public Vector3 GetVerticalAxis(){

return joystickLeft.position.y\*screenMovementForward;

}

public bool GetFireButtonDown(){

return joystickRight.tapCount > 0;

}

public bool GetFireButtonUp(){

return joystickRight.tapCount >= 0;

}

public bool GetBackButtonPressed(){

return Input.GetKeyDown("escape");

}

public bool GetWalkButton() {

return false;

}

public bool GetReloadButton(){

return false;

}

public Vector3 GetFacingDirection(){

Vector3 direction = joystickRight.position.x \* screenMovementRight + joystickRight.position.y \* screenMovementForward;

return direction;

}

public Vector3 GetCameraAdjustment(){

return GetFacingDirection();

}

}

**Health.cs**

using UnityEngine;

using System.Collections;

public class Health : MonoBehaviour {

public int maxHealth;

public int currentHealth;

private float lastDamageTime;

public int CurrentHealth {

get {

return currentHealth;

}

}

void OnEnable() {

currentHealth = maxHealth;

}

public void OnDamage(float amount) {

if (amount <= 0)

return;

networkView.RPC ("DoDamageRemote", RPCMode.AllBuffered, (int)amount);

}

[RPC]

void DoDamageRemote(int amount) {

currentHealth -= amount;

}

}

**MoveAnimation.cs**

using UnityEngine;

using System.Collections;

public class MoveAnimation : MonoBehaviour {

public AnimationClip clip;

public Vector3 velocity ;

private float weight;

private bool currentBest;

public float speed;

public float angle;

public void Init () {

velocity.y = 0;

speed = velocity.magnitude;

angle = PlayerAnimation.HorizontalAngle (velocity);

}

}

**MovementMotor.cs**

using UnityEngine;

using System.Collections;

public class MovementMotor : MonoBehaviour {

public float walkingSnappyness;

public float turningSmoothing;

private Vector3 movementDir;

private Vector3 facingDirection;

private float maxMovementSpeed;

private float maxRotationSpeed;

public Vector3 MovementDir {

get {

return movementDir;

}

set {

movementDir = value;

}

}

public Vector3 FacingDirection {

get {

return facingDirection;

}

set {

facingDirection = value;

}

}

public float MaxMovementSpeed {

get {

return maxMovementSpeed;

}

set {

maxMovementSpeed = value;

}

}

public float MaxRotationSpeed {

get {

return maxRotationSpeed;

}

set {

maxRotationSpeed = value;

}

}

void FixedUpdate () {

Vector3 targetVelocity = movementDir \* maxMovementSpeed;

Vector3 deltaVelocity = targetVelocity - rigidbody.velocity;

if (rigidbody.useGravity)

deltaVelocity.y = 0;

rigidbody.AddForce (deltaVelocity \* walkingSnappyness, ForceMode.Acceleration);

// Setup player to face facingDirection, or if that is zero, then the movementDirection

Vector3 faceDir = facingDirection;

if (faceDir == Vector3.zero)

faceDir = movementDir;

// Make the character rotate towards the target rotation

if (faceDir == Vector3.zero) {

rigidbody.angularVelocity = Vector3.zero;

}

else {

float rotationAngle = AngleAroundAxis (transform.forward, faceDir, Vector3.up);

rigidbody.angularVelocity = (Vector3.up \* rotationAngle \* turningSmoothing);

}

}

float AngleAroundAxis (Vector3 dirA, Vector3 dirB,Vector3 axis) {

// Project A and B onto the plane orthogonal target axis

dirA = dirA - Vector3.Project (dirA, axis);

dirB = dirB - Vector3.Project (dirB, axis);

// Find (positive) angle between A and B

float angle = Vector3.Angle (dirA, dirB);

// Return angle multiplied with 1 or -1

return angle \* (Vector3.Dot (axis, Vector3.Cross (dirA, dirB)) < 0 ? -1 : 1);

}

void OnSerializeNetworkView (BitStream stream, NetworkMessageInfo info)

{

if (stream.isWriting) {

Vector3 dir = transform.position;

Quaternion rot = transform.rotation;

stream.Serialize (ref dir);

stream.Serialize (ref rot);

}

else {

stream.Serialize (ref correctPlayerPos);

stream.Serialize (ref correctPlayerRot);

}

}

private Vector3 correctPlayerPos = Vector3.zero;

private Quaternion correctPlayerRot = Quaternion.identity;

void Update()

{

if (!networkView.isMine)

{

if(correctPlayerPos==Vector3.zero) return;

if (Vector3.Distance(correctPlayerPos, transform.position) < 4)

{

transform.position = Vector3.Lerp(transform.position, correctPlayerPos, Time.deltaTime \* 5);

transform.rotation = Quaternion.Lerp(transform.rotation, correctPlayerRot, Time.deltaTime \* 5);

}

else

{

transform.position = correctPlayerPos;

transform.rotation = correctPlayerRot;

}

}

}

}

**PlayerAnimation.cs**

using UnityEngine;

using System.Collections;

public class PlayerAnimation : MonoBehaviour {

public Rigidbody rigid;

public Transform rootBone;

public Transform upperBodyBone;

public float maxIdleSpeed = 0.5f;

public float minWalkSpeed = 2.0f;

public AnimationClip idle;

public AnimationClip turn;

public MoveAnimation[] moveAnimations;

private Transform tr;

private Vector3 lastPosition = Vector3.zero;

private Vector3 velocity = Vector3.zero;

private Vector3 localVelocity = Vector3.zero;

private float speed = 0;

private float angle = 0;

private float lowerBodyDeltaAngle = 0;

private float idleWeight = 0;

private Vector3 lowerBodyForwardTarget = Vector3.forward;

private Vector3 lowerBodyForward = Vector3.forward;

private MoveAnimation bestAnimation = null;

private float lastFootstepTime = 0;

private float lastAnimTime = 0;

public Animation animationComponent;

void Awake () {

tr = rigid.transform;

lastPosition = tr.position;

foreach (MoveAnimation moveAnimation in moveAnimations) {

moveAnimation.Init ();

animationComponent[moveAnimation.clip.name].layer = 1;

animationComponent[moveAnimation.clip.name].enabled = true;

}

animationComponent.SyncLayer (1);

animationComponent[idle.name].layer = 2;

animationComponent[turn.name].layer = 3;

animationComponent[idle.name].enabled = true;

}

void FixedUpdate () {

velocity = (tr.position - lastPosition) / Time.deltaTime;

localVelocity = tr.InverseTransformDirection (velocity);

localVelocity.y = 0;

speed = localVelocity.magnitude;

angle = HorizontalAngle (localVelocity);

lastPosition = tr.position;

}

void Update () {

idleWeight = Mathf.Lerp (idleWeight, Mathf.InverseLerp (minWalkSpeed, maxIdleSpeed, speed), Time.deltaTime \* 10);

animationComponent[idle.name].weight = idleWeight;

if (speed > 0) {

float smallestDiff = Mathf.Infinity;

foreach (MoveAnimation moveAnimation in moveAnimations) {

float angleDiff = Mathf.Abs(Mathf.DeltaAngle (angle, moveAnimation.angle));

float speedDiff = Mathf.Abs (speed - moveAnimation.speed);

float diff = angleDiff + speedDiff;

if (moveAnimation == bestAnimation)

diff \*= 0.9f;

if (diff < smallestDiff) {

bestAnimation = moveAnimation;

smallestDiff = diff;

}

}

animationComponent.CrossFade (bestAnimation.clip.name);

}

else {

bestAnimation = null;

}

if (lowerBodyForward != lowerBodyForwardTarget && idleWeight >= 0.9f)

animationComponent.CrossFade (turn.name, 0.05f);

if (bestAnimation != null && idleWeight < 0.9f) {

var newAnimTime = Mathf.Repeat (animationComponent[bestAnimation.clip.name].normalizedTime \* 2 + 0.1f, 1f);

lastAnimTime = newAnimTime;

}

}

void LateUpdate () {

float idle = Mathf.InverseLerp (minWalkSpeed, maxIdleSpeed, speed);

if (idle < 1) {

Vector3 animatedLocalVelocity = Vector3.zero;

foreach (MoveAnimation moveAnimation in moveAnimations) {

if (animationComponent[moveAnimation.clip.name].weight == 0)

continue;

if (Vector3.Dot (moveAnimation.velocity, localVelocity) <= 0)

continue;

animatedLocalVelocity += moveAnimation.velocity \* animationComponent[moveAnimation.clip.name].weight;

}

float lowerBodyDeltaAngleTarget = Mathf.DeltaAngle (

HorizontalAngle (tr.rotation \* animatedLocalVelocity),

HorizontalAngle (velocity)

);

lowerBodyDeltaAngle = Mathf.LerpAngle (lowerBodyDeltaAngle, lowerBodyDeltaAngleTarget, Time.deltaTime \* 10);

lowerBodyForwardTarget = tr.forward;

lowerBodyForward = Quaternion.Euler (0, lowerBodyDeltaAngle, 0) \* lowerBodyForwardTarget;

}

else {

lowerBodyForward = Vector3.RotateTowards (lowerBodyForward, lowerBodyForwardTarget, Time.deltaTime \* 520 \* Mathf.Deg2Rad, 1);

lowerBodyDeltaAngle = Mathf.DeltaAngle (

HorizontalAngle (tr.forward),

HorizontalAngle (lowerBodyForward)

);

if (Mathf.Abs(lowerBodyDeltaAngle) > 80)

lowerBodyForwardTarget = tr.forward;

}

Quaternion lowerBodyDeltaRotation = Quaternion.Euler (0, lowerBodyDeltaAngle, 0);

rootBone.rotation = lowerBodyDeltaRotation \* rootBone.rotation;

upperBodyBone.rotation = Quaternion.Inverse (lowerBodyDeltaRotation) \* upperBodyBone.rotation;

}

public static float HorizontalAngle (Vector3 direction) {

return Mathf.Atan2 (direction.x, direction.z) \* Mathf.Rad2Deg;

}

}

**PlayerController.cs**

using UnityEngine;

using System.Collections;

public class PlayerController : MonoBehaviour {

public MovementMotor motor;

public WeaponManager weaponManager;

public Health health;

public string playerName;

public float runSpeed;

public float walkSpeed;

public float maxRotationSpeed;

public Vector3 initOffsetToPlayer;

private Vector3 moveDir;

private float cameraSmoothing;

private float cameraPreview;

private bool dead;

private bool walk;

private Vector3 cameraVelocity;

private bool canFire;

private Camera mainCamera;

private Transform mainCameraTransform;

private Transform character;

private Plane playerMovementPlane;

private float cursorPlaneHeight;

private InputManagerIF inputManager;

private int kills;

private int deaths;

private bool state;

public int Kills {

get {

return kills;

}

set {

kills = value;

}

}

public int Deaths {

get {

return deaths;

}

set {

deaths = value;

}

}

public bool Dead {

get {

return dead;

}

set {

dead = value;

}

}

void Awake() {

mainCamera = Camera.main;

mainCameraTransform = mainCamera.transform;

if (!character)

character = transform;

if(networkView.isMine) {

mainCameraTransform.position = character.position + initOffsetToPlayer;

PlayState.getInstance ().PlayerController = this;

inputManager = InputManagerFactory.GetInputManager(character, mainCamera);

}

else {

if(!PlayState.getInstance ().EnemyPlayers.Contains(this))

PlayState.getInstance ().EnemyPlayers.Add(this);

}

deaths = 0;

kills = 0;

}

void Start() {

walk = false;

cameraSmoothing = 0.01f;

cameraPreview = 2.0f;

state = false;

cameraVelocity = Vector3.zero;

}

void OnEnable() {

moveDir = Vector3.zero;

walk = false;

canFire = true;

motor.MovementDir = Vector2.zero;

motor.FacingDirection = Vector2.zero;

cameraVelocity = Vector3.zero;

state = false;

}

void OnDisable() {

moveDir = Vector3.zero;

walk = false;

canFire = false;

}

void OnDestroy() {

if(!networkView.isMine)

PlayState.getInstance ().EnemyPlayers.Remove(this);

}

void Update() {

if(networkView.isMine) {

walk = (inputManager.GetWalkButton() && !dead) || moveDir == Vector3.zero;

if(!dead) {

moveDir = inputManager.GetHorizontalAxis() + inputManager.GetVerticalAxis();

} else {

moveDir = Vector3.zero;

}

if(canFire){

HandleFiring();

if(inputManager.GetReloadButton()) {

weaponManager.Reload();

}

}

if (moveDir.sqrMagnitude > 1) {

moveDir = moveDir.normalized;

}

motor.MaxMovementSpeed = walk ? walkSpeed : runSpeed;

motor.MovementDir = moveDir;

motor.FacingDirection = inputManager.GetFacingDirection();

Vector3 cameraAdjustmentVector = inputManager.GetCameraAdjustment();

Vector3 cameraTargetPosition = character.position + initOffsetToPlayer + cameraAdjustmentVector \* cameraPreview;

mainCameraTransform.position = Vector3.SmoothDamp (mainCameraTransform.position, cameraTargetPosition, ref cameraVelocity, cameraSmoothing);

}

}

void HandleFiring() {

if (state == false && inputManager.GetFireButtonDown ()) {

weaponManager.StartFire();

state = true;

}

else if (state == true && inputManager.GetFireButtonUp ()) {

weaponManager.StopFire();

state = false;

}

}

public void ChangeName(string name) {

networkView.RPC ("ChangeRemoteName", RPCMode.AllBuffered, name);

}

[RPC]

void ChangeRemoteName(string name) {

playerName = name;

}

public void KillPlayer() {

networkView.RPC ("KillRemotePlayer", RPCMode.AllBuffered);

}

[RPC]

void KillRemotePlayer() {

deaths++;

this.gameObject.SetActive(false);

}

public void AddKill() {

networkView.RPC ("AddRemoteKill", RPCMode.AllBuffered);

}

[RPC]

void AddRemoteKill() {

kills++;

}

}

**SimpleBullet.cs**

using UnityEngine;

using System.Collections;

public class SimpleBullet : MonoBehaviour {

public float speed;

public float lifeTime;

public float dist;

private float spawnTime;

private Transform tr;

void OnEnable () {

tr = transform;

spawnTime = Time.time;

}

void Update () {

tr.position += tr.forward \* speed \* Time.deltaTime;

dist -= speed \* Time.deltaTime;

if (Time.time > spawnTime + lifeTime || dist < 0) {

GameObject.Destroy (gameObject);

}

}

}

**Weapon.cs**

using UnityEngine;

using System.Collections;

public class Weapon : MonoBehaviour {

public string keyToActivate;

public AudioSource shootSound;

public AudioSource reloadSound;

public bool autoReload;

public Texture2D ammunitionBackground;

public Texture2D ammunitions;

public int maxIcons;

public Rect ammunitionRect;

public float frequency;

public float smallConeAngle;

public float largeConeAngle;

public float damagePerSecond;

public float reloadTime;

public GameObject bulletPrefab;

public Transform spawnPoint;

public GameObject muzzleFlashFront;

public int maxAmmo;

public int clipSize;

private bool fire;

private bool hasToReload;

private bool reloading;

private int ammoAmount;

private int clipAmmoAmount;

private bool isHarderToAim;

private float initialReloadTime;

private float lastFireTime;

private RaycastHit hitInfo;

public bool Reloading {

get {

return reloading;

}

}

public int ClipAmmoAmount {

get {

return clipAmmoAmount;

}

}

public bool Fire {

get {

return fire;

}

set {

fire = value;

}

}

public int AmmoAmount {

get {

return ammoAmount;

}

}

void Start() {

initialReloadTime = reloadTime;

if(spawnPoint == null) {

spawnPoint = transform;

}

ammoAmount = maxAmmo;

clipAmmoAmount = clipSize;

ammunitionRect = new Rect(ammunitionRect.x , ammunitionRect.y, Screen.width/ammunitionRect.width, Screen.height/ammunitionRect.height);

muzzleFlashFront.SetActive(false);

isHarderToAim = true;

}

void OnEnable() {

ammoAmount = maxAmmo;

clipAmmoAmount = clipSize;

}

void Update () {

if(networkView.isMine) {

LocalFire ();

}

else {

RemoteFire();

}

}

void RemoteFire() {

if(fire && !hasToReload && !reloading) {

if (Time.time > lastFireTime + 1 / frequency)

{

if (!shootSound.isPlaying) {

shootSound.Play ();

}

float coneAngle = isHarderToAim ? largeConeAngle : smallConeAngle;

Quaternion coneRandomRotation = Quaternion.Euler (Random.Range (-coneAngle, coneAngle), Random.Range (-coneAngle, coneAngle), 0);

GameObject go = (GameObject)GameObject.Instantiate (bulletPrefab, spawnPoint.position, spawnPoint.rotation \* coneRandomRotation);

SimpleBullet bullet = go.GetComponent<SimpleBullet> ();

lastFireTime = Time.time;

Physics.Raycast (spawnPoint.position, spawnPoint.forward, out hitInfo);

if (hitInfo.transform) {

bullet.dist = hitInfo.distance;

}

else {

bullet.dist = 1000;

}

}

} else {

if (shootSound.isPlaying) {

shootSound.Stop ();

}

}

if(reloading) {

if(!reloadSound.isPlaying){

reloadSound.Play();

}

} else {

if(reloadSound.isPlaying){

reloadSound.Stop();

}

}

}

void LocalFire ()

{

if (fire && !hasToReload && !reloading) {

if (clipAmmoAmount == 0) {

if (autoReload) {

reloading = true;

}

hasToReload = true;

muzzleFlashFront.SetActive (false);

fire = false;

}

else if (Time.time > lastFireTime + 1 / frequency) {

float coneAngle = isHarderToAim ? largeConeAngle : smallConeAngle;

Quaternion coneRandomRotation = Quaternion.Euler (Random.Range (-coneAngle, coneAngle), Random.Range (-coneAngle, coneAngle), 0);

GameObject go = (GameObject)GameObject.Instantiate (bulletPrefab, spawnPoint.position, spawnPoint.rotation \* coneRandomRotation);

SimpleBullet bullet = go.GetComponent<SimpleBullet> ();

clipAmmoAmount--;

lastFireTime = Time.time;

hitInfo = new RaycastHit ();

Physics.Raycast (spawnPoint.position, spawnPoint.forward, out hitInfo);

if (hitInfo.transform) {

Health targetHealth = hitInfo.transform.GetComponent<Health> ();

if (targetHealth) {

targetHealth.OnDamage (damagePerSecond / frequency);

if(targetHealth.currentHealth <= 0) {

hitInfo.transform.GetComponent<PlayerController>().KillPlayer();

PlayState.getInstance().PlayerController.AddKill();

}

}

bullet.dist = hitInfo.distance;

}

else {

bullet.dist = 1000;

}

}

}

if (reloading) {

reloadTime -= Time.deltaTime;

if (shootSound.isPlaying) {

shootSound.Stop ();

}

if (!reloadSound.isPlaying) {

reloadSound.Play ();

}

if (reloadTime <= 0) {

hasToReload = false;

reloading = false;

if (ammoAmount >= clipSize) {

int neededAmmo = clipSize - clipAmmoAmount;

clipAmmoAmount += neededAmmo;

ammoAmount -= neededAmmo;

}

else {

clipAmmoAmount = ammoAmount;

ammoAmount = 0;

}

if (reloadSound.isPlaying) {

reloadSound.Stop ();

}

reloadTime = initialReloadTime;

}

}

}

public string GetKeyToActivate() {

return keyToActivate;

}

public void StartFire () {

fire = true && !hasToReload && !reloading;

muzzleFlashFront.SetActive (fire);

if (shootSound != null)

shootSound.Play ();

}

public void StopFire () {

fire = false;

muzzleFlashFront.SetActive (false);

if (shootSound != null)

shootSound.Stop ();

}

public void Reload() {

if(ammoAmount > 0) {

reloading = true;

fire = false;

muzzleFlashFront.SetActive (false);

if(shootSound != null) {

shootSound.Stop ();

}

}

}

public int GetRemainingClips() {

int remainingClips = ammoAmount/clipSize + ((ammoAmount%clipSize == 0) ? 0 : 1);

return remainingClips;

}

void OnSerializeNetworkView(BitStream stream, NetworkMessageInfo messageInfo) {

if(stream.isWriting) {

stream.Serialize(ref fire);

stream.Serialize(ref reloading);

stream.Serialize (ref hasToReload);

stream.Serialize(ref isHarderToAim);

} else {

stream.Serialize(ref fire);

stream.Serialize (ref reloading);

stream.Serialize(ref hasToReload);

stream.Serialize(ref isHarderToAim);

}

}

}

**WeaponManager.cs**

using UnityEngine;

using System.Collections;

public class WeaponManager : MonoBehaviour {

public Weapon[] weapons;

public Weapon currentWeapon;

void Update() {

for(int i=0; i<weapons.Length; i++) {

if(Input.GetKey (weapons[i].GetKeyToActivate())) {

currentWeapon = weapons[i];

PlayState.getInstance ().hudWeapons.currentWeapon = weapons[i];

}

}

}

public void StartFire() {

if(currentWeapon != null) {

currentWeapon.StartFire ();

}

}

public void StopFire() {

if(currentWeapon != null) {

currentWeapon.StopFire ();

}

}

public void Reload() {

if(currentWeapon != null && !currentWeapon.Reloading) {

currentWeapon.Reload ();

}

}

}