|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Информационная безопасность

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

***К КУРСОВОМУ ПРОЕКТУ***

***НА ТЕМУ:***

“Разработка компьютерной 3D игры”

Студент ИУ8-33 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Евула А. С.

(группа) (подпись, дата) (Фамилия И. О.)

Руководитель курсового проекта \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Вершинин В. В.

(подпись, дата) (Фамилия И. О.)

*2019 г.*

Оглавление

1. Введение

2. Цель работы и задачи

3. Ход работы

3.1. Unreal Engine 4

3.2. Создание концепции игры

3.3. Создание прототипа

3.4. Создание графического интерфейса игрока

3.5. Тестирование полученной игры

3.6. Подключение сервисов непрерывной интеграции, создание документации и презентации

4. Результат

1. Введение

Предметная область курсового проекта включает в себя теоретический и практический материал из области игровой индустрии, особенности игрового движка Unreal Engine 4 для разработки компьютерных игр.

Для разработки проекта использовалась версия 4.21.0 Unreal Engine. На сегодняшний день игры жанра “Shooter” являются одними из наиболее популярных и востребованных игр игровой индустрии.

По этой причине было принято решение написать игру жанра “First Person Shooter” (далее “FPS”).

2. Цель работы и задачи

Цель работы – разработать компьютерную 3D игру на языке программирования C++ с графическим интерфейсом пользователя, используя игровой движок Unreal Engine 4.

Поставленные задачи:

1. Создание концепции игры.
2. Создание прототипа.
3. Создание графического интерфейса игрока.
4. Тестирование полученной игры.
5. Подключение сервисов непрерывной интеграции, создание документации и презентации.

3. Ход работы

3.1. Unreal Engine 4

Перед описанием игры стоит поговорить об игровом движке, на котором и будет написана сама игра. Unreal Engine – игровой движок, написанный на языке C++, разрабатываемый и поддерживаемый компанией Epic Games. Изначально данный движок был разработан специально для игр жанра “FPS”.

Unreal Engine поддерживает большое количество платформ, технологий, API.

В Unreal Engine присутствует удобная четкая иерархия классов:

Все элементы игрового движка представлены в виде *Объектов*. Все классы в свою очередь являются наследниками данного класса.

В качестве основополагающих классов следует выделить:

*Актер* (Actor) - это базовый класс для объекта, который имеет отношение к игровому процессу, может быть помещен или создан на уровне.

*Пешка* (Pawn) - это базовый класс всех актеров, которыми могут управлять игроки или искусственный интеллект (далее AI).

*Компонент* (Component) - это часть функциональности, которую можно добавить *Актеру*.

*Контроллер* (Controller) - это нефизический *Актер*, который может “обладать” *Актером* для управления ее действиями.

*Режим игры* (Game Mode) – это нефизический класс, имеющий основу правил, которые управляют/корректируют/регулируют игровой процесс.

*Мир* (World) - это объект верхнего уровня, представляющий карту (уровень), в которой будут существовать и визуализироваться *Актеры* и *Компоненты*.

Также следует уточнить:

*Меш* – это геометрический объект, состоящий из набора полигонов, которые можно кэшировать в видеопамяти и визуализировать с помощью видеокарты.

*Скелетный меш* – это геометрический объект, состоящий из двух частей: самого *меша,* и иерархического набора взаимосвязанных костей, которые могут быть использованы для анимации вершин полигонов.

3.2. Создание концепции игры

Игра жанра “FPS” – игра, в которой игрок управляет объектом – *Персонажем*, перемещаясь по игровому *Полю*, уничтожая *Врагов*.   
В качестве дополнения в проекте будут реализованы

*Система очков здоровья* – *Персонаж* игрока и *Враги* имеют свое количество здоровья, уменьшаемое при получении *Урона*. Разные *Пули* (или *Снаряды*) наносят разный *Урон*. Если здоровье опускается до нуля, персонаж умирает;

*Инвентарная система* – игрок может поместить в свой инвентарь и использовать или выбросить *Предмет*;

*Дверная система* – игрок может открывать *Двери* для прохождения в различные локации. Для открытия некоторых Дверей потребуется специальные *Ключ-Карты*.

3.3. Создание прототипа

Первоначально был создан класс персонажа игрока *ADoomfallCharacter* (*A* – Actor, *Doomfall* –название проекта*, Character* –класс *Персонажей*).

Персонаж игрока состоит из:

1. Двух *Скелетных мешей* – модель персонажа и *Оружия.*
2. *Капсулы столкновений.*
3. Полного и текущего здоровья.
4. Полного и текущего количества боеприпасов (далее *Амуниция*).
5. Урон *Пуль* текущего *Оружия.*
6. Функций передвижения и поворота.
7. Функций стрельбы и перезарядки.
8. Функции проверки на наличие *Взаимодействующих* *Объектов* перед игроком (далее “*текущий Предмет”*)
9. Компонентов *Пули*, Звука, Анимаций, Ввода, перекрытия *Объектов.*
10. Множество “setter-ов” и ”getter-ов”.
11. Множество переменных состояний.

Пример части исходного кода класса *Персонажа* игрока:

/// Класс Персонажа игрока

UCLASS(config = Game)

class ADoomfallCharacter : public ACharacter

{

    GENERATED\_BODY()

    UPROPERTY(VisibleDefaultsOnly, Category = Mesh)

        /// Скелетный Меш Персонажа

        class USkeletalMeshComponent\* Mesh1P;

    UPROPERTY(VisibleDefaultsOnly, Category = Mesh)

        /// Скелетный Меш Оружия

        class USkeletalMeshComponent\* FP\_Gun;

...

После описания логики Персонажа, была написана функция *CheckForInteactables()*:

void ADoomfallCharacter::CheckForInteractables() {

    FHitResult HitResult;

    /// Начало и конец “Луча”

    FVector StartTrace = FirstPersonCameraComponent->GetComponentLocation();

    FVector EndTrace = (FirstPersonCameraComponent->GetForwardVector() \* 400) + StartTrace;

    /// Установка параметров пересечения

    FCollisionQueryParams QueryParams;

    QueryParams.AddIgnoredActor(this);

    AGameplayController\* Controller = Cast<AGameplayController>(GetController());

    if (GetWorld()->LineTraceSingleByChannel(HitResult, StartTrace, EndTrace, ECC\_Visibility, QueryParams) && Controller) {

        /// Если Луч пересекся с Взаимодействующим Объектом

        if (AInteractable\* Interactable = Cast<AInteractable>(HitResult.GetActor())) {

            Controller->CurrentInteractable = Interactable;

            return;

        }

    }

    /// Иначе

    if (Controller) {

        Controller->CurrentInteractable = nullptr;

    }

}

Для привязки ввода к действиям используется класс *UInputComponent:*

void ADoomfallCharacter::SetupPlayerInputComponent(class UInputComponent\* PlayerInputComponent) {

    /// Проверка наличия компонента ввода

    check(PlayerInputComponent);

    /// Бинд Прыжка

    PlayerInputComponent->BindAction("Jump", IE\_Pressed, this, &ACharacter::Jump);

    PlayerInputComponent->BindAction("Jump", IE\_Released, this, &ACharacter::StopJumping);

    /// Бинд Перезарядки

    PlayerInputComponent->BindAction("Reload", IE\_Pressed, this, &ADoomfallCharacter::Reload);

...

Для включения *Анимаций*, к примеру выстрела, используется *UAnimInstance*:

if (FireAnimation != NULL) {

       UAnimInstance\* AnimInstance = Mesh1P->GetAnimInstance();

       if (AnimInstance != NULL) {

             AnimInstance->Montage\_Play(FireAnimation, 1.f);

       }

}

Для проигрывания Звука, к примеру выстрела, используется метод библиотеки Kismet/UGameplayStatics:

if (FireSound != NULL) {

       UGameplayStatics::PlaySoundAtLocation(this, FireSound, GetActorLocation());

}

Для объемного и позиционного проигрывания звуков используются *Объекты* *SoundAttenuation*.

Для игрока доступны 2 вида стрельбы – Авто (игрок зажимает клавишу, и выстрелы производятся автоматически с определенным промежутком), Одиночная (игрок нажимает на клавишу, и производится одиночный выстрел). Если Амуниция станет равна нулю – персонаж автоматически перезарядится.

Методы библиотеки Kismet часто используются в коде, например:

*CreateTransform(), CreateRotator(), GetPlayerCharacter(), GetPlayerController()* и т.п.

Итак, у нас есть Персонаж, который может передвигаться, поворачиваться, прыгать, стрелять, перезаряжаться, выполнять проверку на наличие *текущего Предмета*. Анимации всех действий создаются в самом Editor’е, а переход анимаций сопровождается простой логикой, использующей переменные булевого типа. Например, если переменная *bIsInAir* равна *true* (присвоение происходит при вызове метода прыжка), следует включить анимацию начала прыжка, далее после окончания данной анимации включается циклическая анимация падения, и как только игрок коснется пола или другого физического *Объекта,* переменной *bIsInAir* будет присвоено значение *false*, и в свою очередь проиграна анимация окончания прыжка.

Теперь создадим класс *AGameplayController* – *Контроллер* игрока (так называемый “мозг” *Персонажа*).

Данный класс содержит:

1. *Инвентарь* игрока.
2. Метод добавления *Предмета* в *Инвентарь, Взаимодействия,* а также методы открытия/закрытия *Меню* и *Инвентаря.*
3. *Текущий Предмет*, если он существует.

Методы открытия/закрытия *Инвентаря*/*Меню* вызывают соответствующую функцию класса *ADoomfallHUD* – графического интерфейса пользователя.

Метод *Interact() –* вызывает метод *Interact()* у *текущего Предмета.* Например, если это *Предмет Инвентарного* вида, то в нем будет вызываться метод *AddItemToInventory()* класса *AGameplayController.* Вот его содержание:

bool AGameplayController::AddItemToInventoryByID(FName ID) {

    ADoomfallGameMode\* GameMode = Cast<ADoomfallGameMode>(GetWorld()->GetAuthGameMode());

    if (GameMode) {

        /// Получение Предметной Базы Данных

        class UDataTable\* ItemTable = GameMode->GetItemDB();

        if (ItemTable) {

            /// Поиск текущего Предмета в БД по ID

            FInventoryItem\* ItemToAdd = ItemTable->FindRow<FInventoryItem>(ID, "");

            /// Если Предмет найден

            if (ItemToAdd) {

                Inventory.Add(\*ItemToAdd);

                return true;

            }

        }

    }

    /// Если Предмет не найден

    return false;

}

Теперь *Персонаж* игрока полностью готов.

Перейдем к описанию класса *AInteractable*. К данному классу относятся все *Объекты*, с которыми может взаимодействовать игрок (предметы, двери, NPC и т.п.). Данный класс будет базовым для многих других классов и содержать базовую логику взаимодействий.

Каждый *Взаимодействующий Предмет* (далее просто *“Предмет”*) состоит из:

1. *Меша*.
2. Имени.
3. Называния взаимодействия.
4. Методов *Взаимодействия*, а также 3 “*getter-ов*”.

Метод *Interact* является абстрактным и для каждого класса-наследника он свой.

Из *“getter-ов”* стоит выделить метод *GetActionText()*, необходимый для вывода информации на экран игрока:

FString AInteractable::GetActionText() const {

    return FString::Printf(TEXT("%s: Press E to %s"), \*Name, \*Action);

}

Класс *APickup* – класс *Предметов Инвентарного* вида и наследник класса *AInteractable*, т.е. игрок может положить данный *Предмет* в свой *Инвентарь* и после его использовать/выбросить.

В данном классе добавлены:

1. ID *Предмета.*
2. Метод *Use().*
3. Переопределенный метод *Interact().*

Для данного класса следует настроить параметры столкновений, т.е. любые *Актеры* (игрок, *Враг*, *Пули*) могут проходить сквозь *Предмет.* Для этого используется методы *Меша*, например

*Mesh->SetCollisionResponseToAllChannels(ECollisionResponse::ECR\_Ignore)*

*Mesh->SetCollisionResponseToChannel(ECollisionChannel::ECC\_WorldStatic,*

*ECollisionResponse::ECR\_Block)*

Метод *Use()* вызывает соответствующий по имени *Предмета* метод в классе *Персонажа.* Например, получение здоровья *Аптечкой:*

if (Name == FString("Med Kit Small")) {

    Player->ReceiveDamage(50);

}

Класс *ADoor* – класс дверей, открывающихся при взаимодействии с игроком. Как и *APickup* содержит свой ID, а также преопределенный метод взаимодействия – *Interact()* – если у *Двери* нет ID (“None”), то дверь открывается без *Ключ-Карты*, иначе в *Инвентаре* игрока должна присутствовать *Ключ-Карта* с соответствующим ID.

Для *Врагов* игрока используется класс *AAI\_Bot* – наследник класса *ACharacter.*

Данный класс содержит:

1. Два *Скелетных меша* – модель бота и *Оружия.*
2. *Капсулу столкновений.*
3. Полного и текущего здоровья.
4. Урон *Пуль* текущего *Оружия.*
5. Функций стрельбы.
6. Компонентов *Пули*, Звука, Анимаций, перекрытия *Объектов.*
7. Множество “setter-ов” и ”getter-ов”.
8. Множество переменных состояний.
9. Следующая *Точка патруля.*

В отличие от *Персонажа* игрока бот не имеет функции передвижения и поворота. За передвижение бота отвечает его *Контроллер* класса *AAI\_BotController*.

Класс *AAI\_BotController* – класс *Контроллера* бота и состоит из:

1. Методов поиска игрока, выстрела.
2. Множества “*setter-ов*” и “*getter-ов*”.
3. Множества переменных состояний.
4. *Конфига* бота.

*Конфиг* бота состоит из Радиуса поиска и потери игрока, Времени памяти последней локации игрока, Угла обзора, а также Радиусов поиска и потери игрока во время сражения, т.е. бот имеет возможность обнаруживать игрока только с помощью *зрения.*

Метод *CheckForPlayer*() вызывается каждый *тик* и имеет незамысловатую логику:

Если бот не гонится за врагом, то игрок не замечен, а, следовательно, следует направится к следующей *Точке патруля.*

Если игрок замечен, то бот гонится за игроком и следует передвигаться к нему и стрелять в него.

Если игрок не замечен, но бот гонится за игроком, следует передвигаться к последней видимой локации игрока.

Если бот получен *Урон,* то игрок теперь замечен, и бот гонится за ним, а, следовательно, следует передвигаться к нему и стрелять в него.

Если игрок вне Радиуса потери, то игрок не замечен (если не в сражении).

Для перемещения бота используется метод класса *AActor* *MoveToLocation()*, а для остановки используется метод *StopMovement().*

Для правильной работы Системы стрельбы для ботов и игрока созданы отдельные классы *Пуль*: *AEnemyProjectile и APlayerProjectile* соотвественно*.* Оба класса – наследники класса *ADoomfallProjectile* и служат лишь для различия при перекрытии *Объектов.*

Для точной работы Системы патрулированиянеобходимо описатькласс *AWayPoint* – класс *Точек патрулирования.*

Состоит данный класс из

1. Компонента сцены.
2. Нефизического куба – триггера.
3. Указателя на следующую *Точку патрулирования*.

Если бот входит в триггер *Точки патруля*, полю бота *NextWayPoint* присваивается значение данной *Точки патруля*. Фрагмент исходного кода:

void AWayPoint::OnPlayerEnter(UPrimitiveComponent \* OverlapComponent, AActor \* OtherActor,

UPrimitiveComponent \* OtherComponent, int32 OtherBodyIndex,

 bool bFromSweep, const FHitResult & SweepResult) {

    if (OtherActor) {

        AAI\_Bot\* Bot = Cast<AAI\_Bot>(OtherActor);

       /// Если вошел именно Объект класса AAI\_Bot

       if (Bot) {

           Bot->SetNextWayPoint(NextWayPoint);

       }

    }

}

*ADoomfallGameMode* – класс *Режима* игры, т.е. определенной совокупности правил и состояний игры.

Данный класс содержит:

1. Базу Данных Предметов.
2. Текущее состояние игры.
3. Текущую сложность игры.

Возможные состояния игры – *Playing, GameOver, Unknown*.

Возможные сложности игры – Easy, Medium, Hard.

Также существует класс *ALevelEnd* - при пересечении игрока с *Объектом* этого класса *Уровень* считается пройденным, *UGamePlayInstance* – класс, содержащий последнюю сохраненную информацию о состоянии игры, игрока и необходимый для сохранения и загрузки данной информации, *ADoomfallHUD –* класс, отвечающий за графический интерфейс игрока.

3.4. Создание графического интерфейса игрока

Для создания графического интерфейса используются методы класса *ADoomfallHUD* – наследника класса *AHUD*, а также *BlueprintWidget*’ы (создаются и редактируются в самом Editor’е).

Фрагмент исходного кода установки *Геймплейного* интерфейса игрока.

ADoomfallHUD::ADoomfallHUD() {

    /// Установка Интерфейса Игрока

    static ConstructorHelpers::FClassFinder<UUserWidget> UIObj(TEXT("/Game/UI/UserInterface"));

    HUDWidgetClass = UIObj.Class;

}

void ADoomfallHUD::DrawHUD() {

    Super::DrawHUD();

}

void ADoomfallHUD::BeginPlay() {

    Super::BeginPlay();

/// Если текущий уровень – НЕ Меню

    if (GetWorld()->GetMapName() != "UEDPIE\_0\_MainMenu") {

        /// Если существует Класс Интерфейса Игрока

        if (HUDWidgetClass) {

            UWorld\* const World = GetWorld();

            if (World != NULL) {

                /// Если НЕ существует Интерфейс Игрока

                if (!UIWidget) {

                    UIWidget = CreateWidget<UUserWidget>(World, HUDWidgetClass);

                    UIWidget->AddToViewport();

                }

            }

        }

    }

}

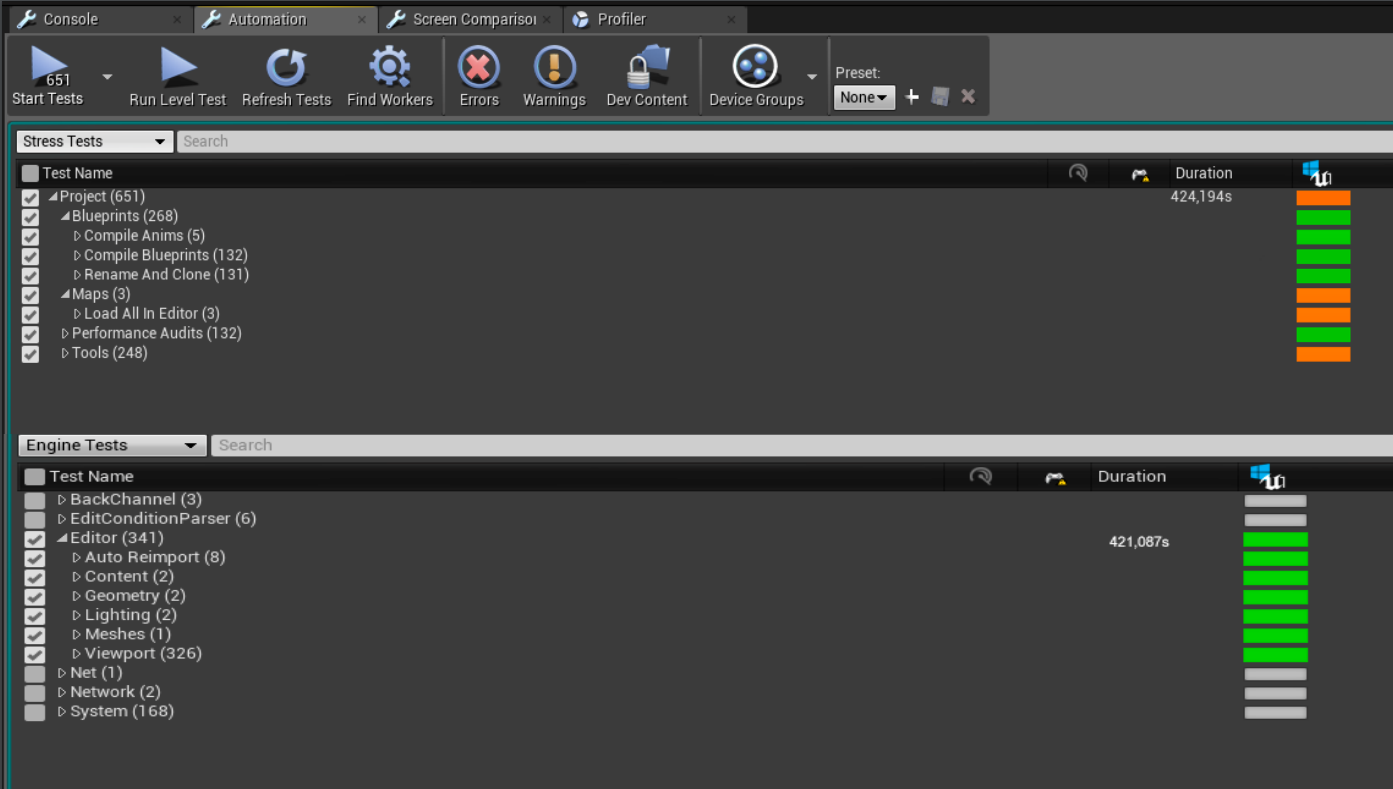
Вот так выглядит *Геймплейный* интерфейс игрока и его состовляющие:



3.5. Тестирование полученной игры

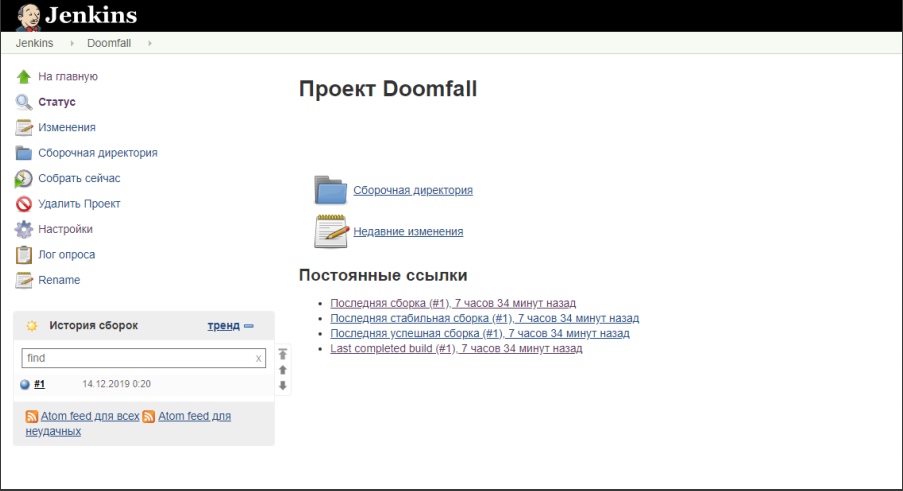
Для Тестирования проекта использовались специальные плагины Unreal Engine для тестирования.

Были проведены стресс-тесты, а также тесты кода проекта и движка. Как видно большинство тестов были успешно пройдены, а пара тестов (Load all in Editor для Maps и Tools) была пройдена с предупреждением, т.к. в процессе создания проекта были удалены некоторые используемые в предыдущих версиях проекта *Asset*’ы, что никак не влияет на проект.

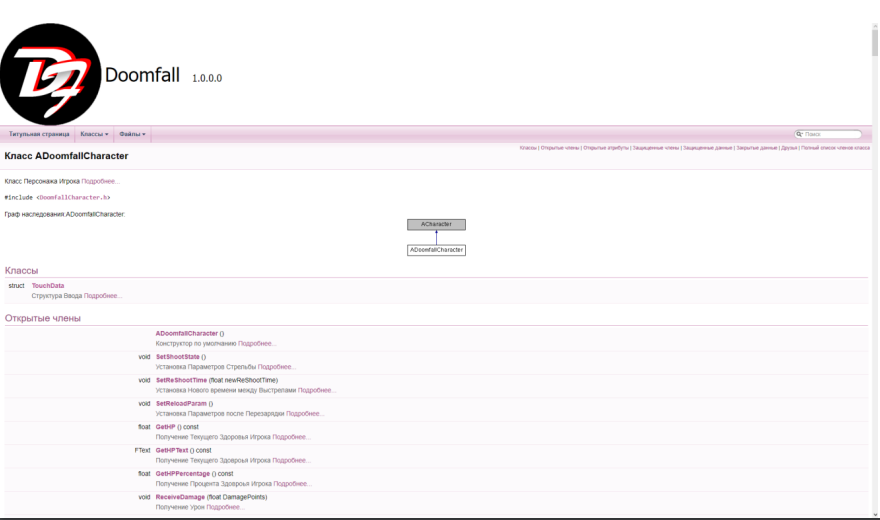


3.6. Подключение сервисов непрерывной интеграции, создание документации и презентации.

В качестве сервиса непрерывной интеграции был выбран Jenkins из-за объемности проекта и удобства сервиса. Как видно, все сборки были проведены успешно.



Для создания документации использовался Doxygen:





Также для данного проекта был создан свой Логотип, пара загрузочных экранов и фон *Меню*.



4. Результат

В качестве результата данной курсовой работы, мною была получена богатая база знаний о создании игр в особенности с помощью движка Unreal Engine и на языке программирования C++, о тестировании игр, была написана 3D игра жанра “First Person Shooter”, прошедшая тесты, представленные Unreal Engine, был подключен к проекту сервис непрерывной сборки Jenkins.

Исходные файлы проекта – <https://github.com/BootyAss/Doomfall>

Документация проекта - <https://disk.yandex.ru/client/disk/Doomfall/html>

Презентация проекта - <https://disk.yandex.ru/client/disk/Doomfall/Presentation>