ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

ДИПЛОМНА РАБОТА

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

Тема: Игра за състезание с коли с подобрения

Дипломант:
Александър Андреев Асса

Дипломен ръководител:
Борис Димитров

СОФИЯ

2 0 2 3

ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

Дата на заданието: 22.11.2022 г. Утвърждавам:.....
Дата на предаване: 22.02.2023 г. /проф. д-р инж. П. Якимов/

ЗАДАНИЕ
за дипломна работа

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

на ученика Александър Андреев Асса от 12 Б клас

1. Тема: Игра за състезание с коли с подобрения

2. Изисквания:
- Превключване между няколко камери по време на състезание;
- Система за съхранение на най - добро време;
- Една писта за състезания;
- Режим на игра, при който играчът се стреми да подобри зададеното време;
- Режим на игра, при който играчът се състезава с друг играч, играещ на същото устройство

3. Съдържание 3.1 Теоретична част
3.2 Практическа част
3.3 Приложение

Дипломант :.....
/ Александър Асса /
Ръководител:.....
/ Борис Димитров /

Директор:.....
/ доц. д-р инж. Ст. Стефанова /

Становище

на дипломна работа

Дипломант: **Александър Андреев Асса**
Ръководител: **Борис Руменов Димитров**
Тема на дипломна работа: **Игра за състезание с коли с подобрения**

...Дипломантът работи консистентно и с желание
...при проекта си през учебната година.....
..Определям проекта като достатъчно завършен и
..готов за допускане до защита, с постигна-
..тите изискуеми от заданието на дипломната
..работа.....
.....
.....
.....
.....

Предлагам за рецензент: Александър Галинов Ангелов
тел: 0878954530
e-mail: aleksandar.angelovv@gmail.com

Подпис на ръководител:

Увод

Създаването на видеоигри започва през 60-те години на XX. век. Тогава игрите не са имали комерсиален характер и са били достъпни за ограничен кръг потребители, най-често за самия разработчик. Смята се, че играта Spacewar!, първоначално разработена от Стив Ръсел и негови колеги от Масачузетския технологичен институт (MIT), е първата игра, която е била постоянно надграждана и съответно достъпна за сравнително широк кръг играчи.

Разработването и разпространението на игрите за комерсиални цели започва през 70-те години, когато са създадени първите аркадни и конзолни игри. Нарастващата популярност, заедно с по-добрите възможности в мощността и графиката на хадуера, стават причина за развитието на изцяло ново направление в индустрията. В създаването на една видеоигра вече участват огромни екипи, наброяващи по над 100 души.

В началото на XXI. век с все по-широкото разпространение, усъвършенстване и достъпността на персонални компютри и мобилни телефони видеоигрите стават ежедневие за потребителите на глобално ниво.

Основната цел на този проект е създаването на игра, в която играчът контролира триизмерна кола. Целта на играта е да се финишира за възможно най-кратко време. Проектът включва разработването на два режима на игра:

- режим, при който играчът изминава определеното трасе за по – малко от зададеното време;
- режим, при който играчът може да се състезава срещу друг играч, който играе на същата клавиатура, от същото устройство.

Първа глава

Преглед на подобни продукти и на известни развойни среди

1.1. Преглед на подобни продукти

1.1.1. Mario Kart

Марио Карт е известна поредица за аркадни състезания. Първата игра от поредицата излиза през 1992г. Последната игра, излязла само за Nintendo switch, е Mario kart 8 deluxe. В нея, както в предишните играчът се състезава срещу няколко противника, които могат да бъдат или хора, или ботове. По време на игра състезателите могат да взимат подобрения, които могат да им правят колата по – добра или могат да забавят колите на противниците. Всеки играч може да държи в себе си до 2 подобрения, като когато играчът взима подобрния не знае какво точно ще му се падне. Играчът има избор от шофьор, шаси на колата и гуми, всяко от които определя колко бързо набира, как завива, и колко е максималната скорост. Играта може да се играе срещу конзолата, други хора локално(до 4 човека) или онлайн(до 12 човека). Има няколко режима на игра, които са:

Grand Prix – при който играчът се състезава в няколко поредни състезания, всяко от които даващо му точки в зависимост от позицият на финиширане.

Time Trials – при който играчът се стреми да направи възможно най – бързото време. При този режим на игра има само един състезател.

Battle Mode – където играчите са в затворена арена без начало или край и за да спечелят трбва да изпълнят предварително определено действие.

Въпреки че играта се продава само за Nintendo switch, играта бива купена близо до 48 миолиона и 400 хиляди пъти, най -много в поредицата. Според някои тя е най – добрата игра в поредицата след оригиналният Mario kart.

Фиг. 1.1.1.1. Процес на игра на Mario kart 8 deluxe (Grand Prix).

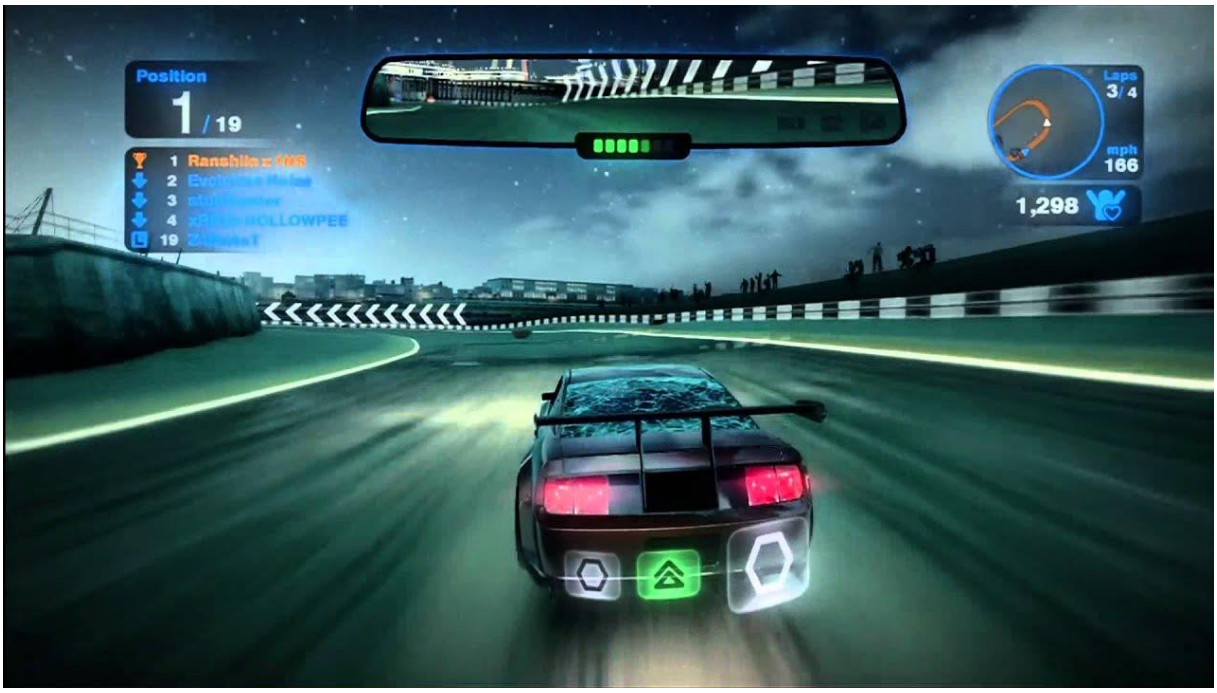
1.1.2. Blur

Blur е състезателна игра, разработена от Bizarre Creations и излязла през 2010 г. Излиза за Windows, PlayStation3, Xbox 360. В нея играчът има избор дали да се състезава срещу хора или срещу изкуствен интелект. По време на състезание играчите могат да взимат подобрения, които могат да ги забързат, да забавят противниците си по няколко различни начина или да се защитят от възпрепятствие от противниците. В играта съществуват няколко различни начина на игра, всеки с различни правила. Те биват:

- Обикновено състезание, където до 20 коли се състезават;
- Състезание срещу часовника, където играчът се опитва да подобри предварително зададено време (само за 1 играч);
- Състезание за точност, където играчът се стреми възможно най- много контролирани от играта коли с забавящи ги подобрения (само за 1 играч);
- Отборно състезание, където играчите биват разделени по равно в два отбора по максимум 10 човека, като завършващата позиция на всеки от играчите носи определен брой точки на отбора му. Отборът с най-много точки печели. (само за повече от 1 играч).

Освен това играта има и кампания за 1 играч, която добавя нов начин на игра-дуел. По време на дуел се състезаваш срещу само един противник, като за да спечелиш трябва или да финишираш първи, или да уцелиш колата му достатъчно пъти с забавящи подобрения. За да напредне в кампанията играчът трябва да извършва

предварително определени предизвикателства като да завърши 5 пъти на някое от първите 3 места.



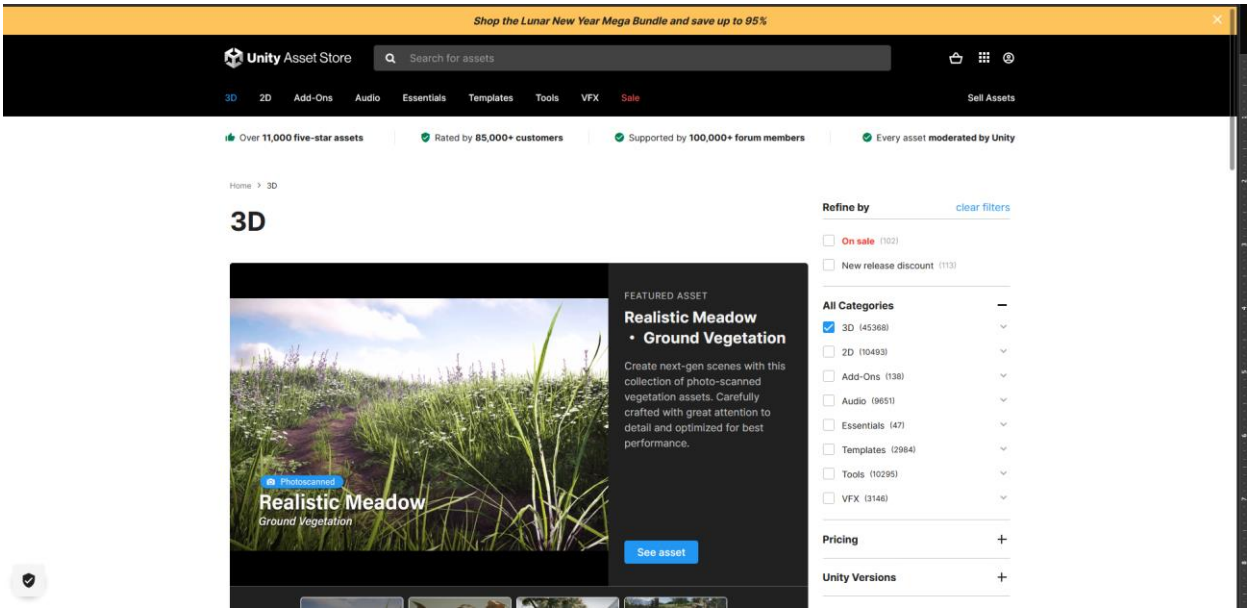
Фиг. 1.1.2.1. Процес на игра на Blur (обикновено състезание).

Колите в Blur се различават по максимална скорост, ускорение, начин на завиване и устойчивост, която определя колко пъти може да бъде ударена колата, преди да трябва да почне от последната премината точка за съхранение на прогреса. Освен това колите са разделени в няколко класа, така че колите в един клас да са с еднакви възможности. Играта продава само 500 хиляди копия. Тя е предпоследната издадена игра на Bizarre Creations, които закриват февруари 2011 г.

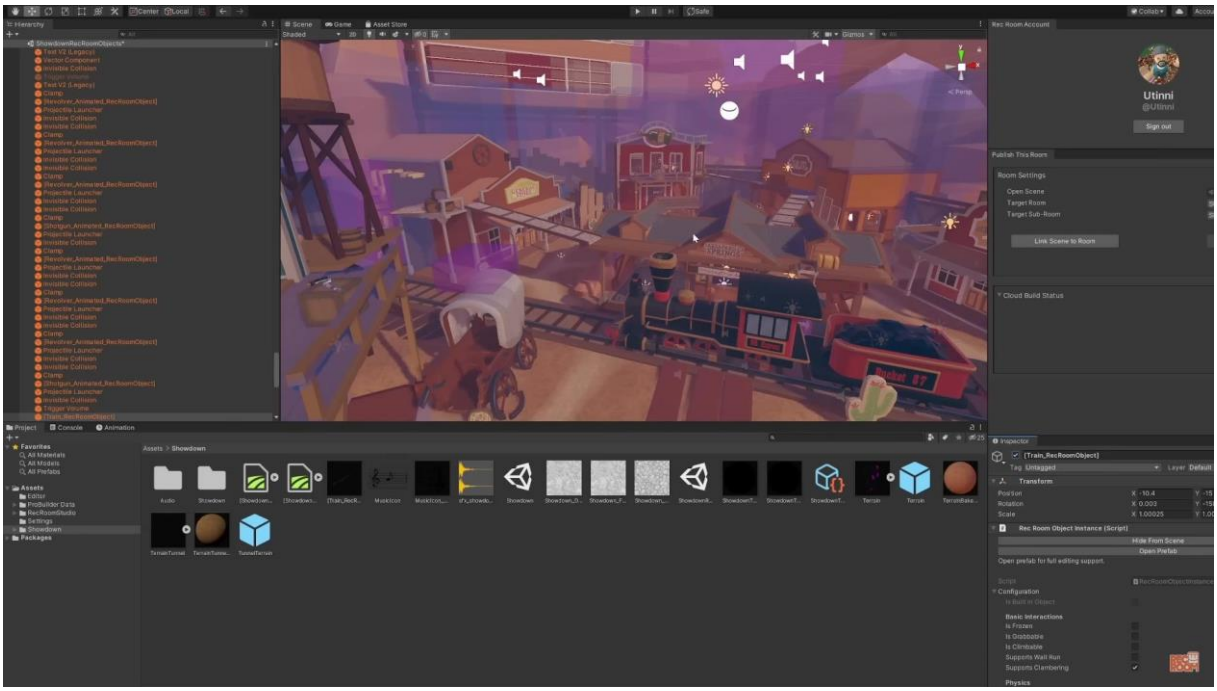
1.2. Технологии за създаване на игра

1.2.1. Unity

Unity е многоплатформен игрови двигател, поддържащ Windows, Linux, macOS, WebGL, Android, IOS, Nintendo Switch, PlayStation, Google Stadia, Xbox и много други. Той се използва главно за създаването на Indie игри и игри за IOS и Android. Поддържа правенето на 2D и 3D игри. Освен за игри Unity се използва в архитектурата, правенето на филми и automotive индустрията. Unity поддържа като език за програмиране само C#, откакто спряха поддръжката на UnityScript, техен език базиран на JavaScript, през август 2017г. Главните плюсове на Unity са че той е безплатен за проекти с малък годишен доход, активна общност, Unity Asset Store, от където могат лесно да се намерят и добавят модели и текстури за какъвто и да е проект. Има по – голяма поддръжка за двуизмерни игри спрямо Unreal Engine. Друг плюс на игровия двигател е че е лесен за научаване. Освен това човек може да закупи допълнително опция да се добавят плащания към игрите, сървъри ако играта ти ползва такива. Популярни игри правени с Unity са Escape from Tarkov, Outer Wilds.



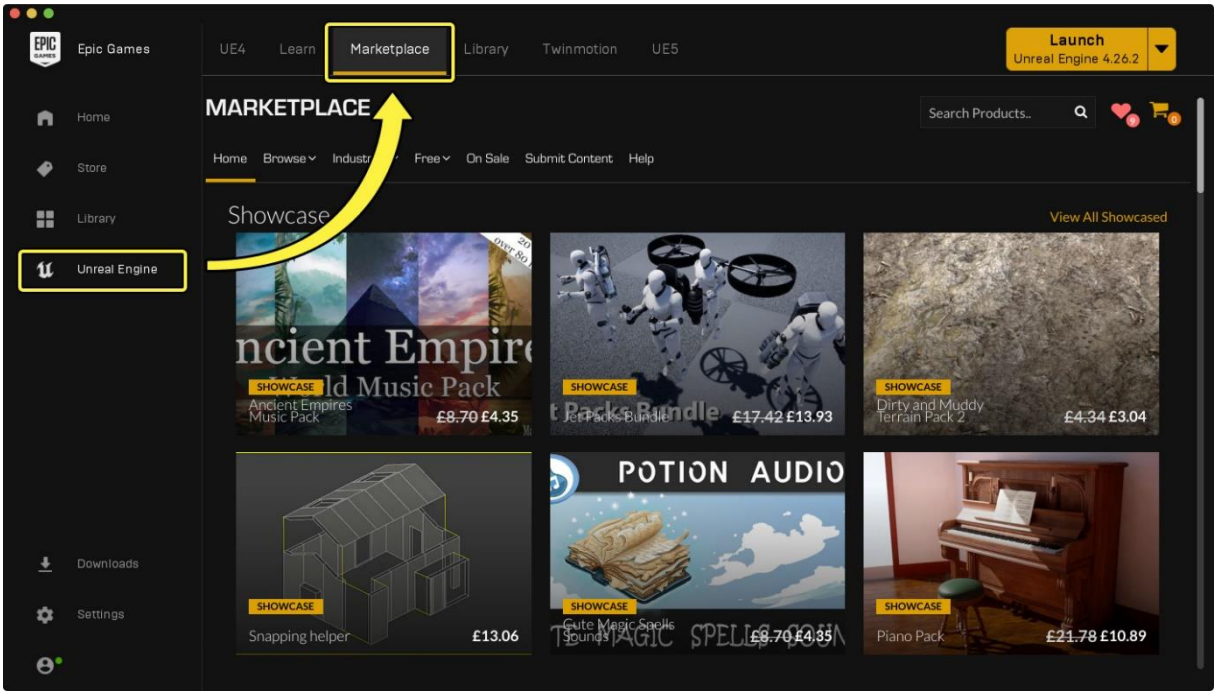
Фиг. 1.2.1.1. Unity Asset Store.



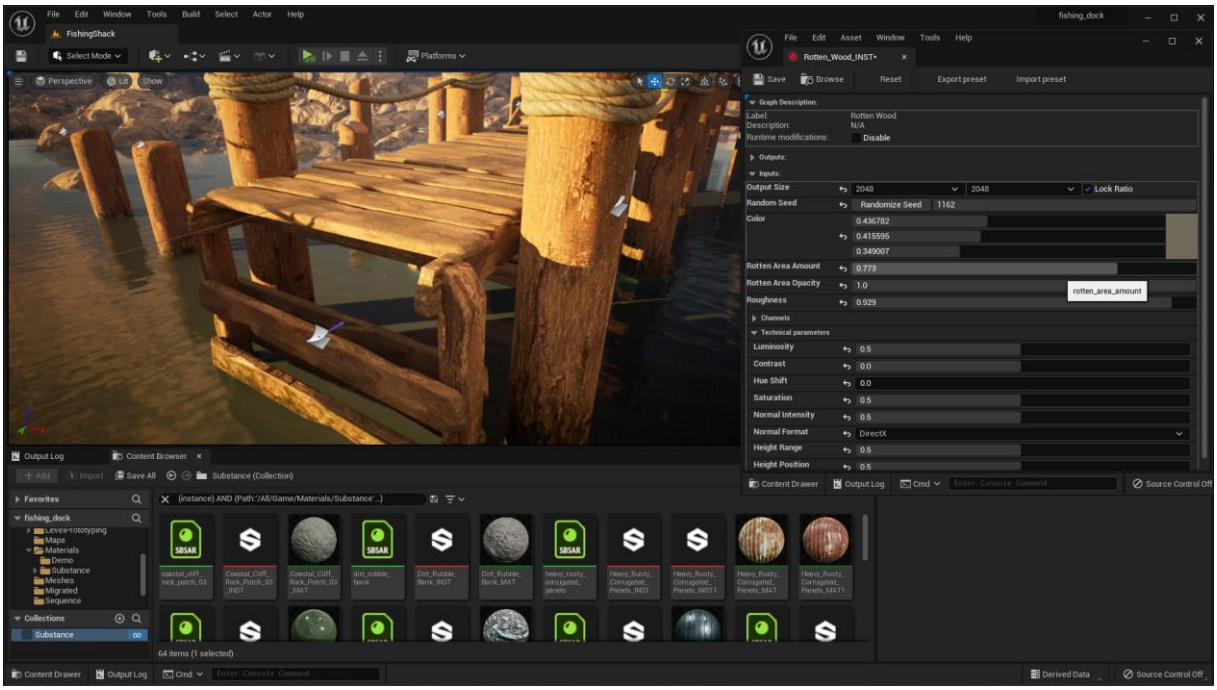
Фиг. 1.2.1.2. Unity.

1.2.2 Unreal engine

Unreal Engine е игрови двигател, които поддържа Windows, Linux, macOS, Android, IOS, Nintendo Switch, PlayStation, Google Stadia, Xbox и други. Започнал е като двигател за игри от шутер жанъра, като в последствие бива използван за визуализация в архитектурата, създаването на филми, за визуализация в automotive индустрията и всякакви видео игри. За програмиране има избор между C++ и програмиране на Blueprints, които са подобни на блокови схеми. Има интеграция с Microsoft Visual Studio за програмиране на C++. Въпреки че игровият двигател поддържа създаването на двуизмерни игри, той е специално направен за триизмерни игри. Използва логически дървета за имплементацията на изкуственият интелект на героите, контролирани от компютъра. Предимства на Unreal engine са че е безплатен, стига доходът ти да по – малък от 1 милион американски долара, активна общност с много обучителни статии и видеа, предварително създадени ситеми за съхраняване на прогреса на играча и игра он няколко играча, по – развити технологии за триизмерна графика спрямо Unity. Съществува и Unreal Marketplace, където могат да бъдат закупени и сложени за продажба триизмерни модели, анимации и звукови ефекти. Популярни игри направени с него са Fortnite, Mortal Kombat 11, Borderlands 3, Sea of Thieves.



Фиг. 1.2.2.1. Unreal Marketplace.



Фиг. 1.2.2.2. Unreal Engine.

Втора глава

Функционални изисквания и аргументиран избор на развойна среда

2.1. Функционални изисквания

- Да се създаде игра, в която играчът контролира триизмерна кола. Колата ще разполага с предварително зададени точки живот.
- Да може играчът да превключва между няколко изгледа по време на игра, без да отива в менюта.
- Да се създаде режим на игра, при който играчът изминава определеното трасе за по-малко от зададеното време.
- Да се създаде режим на игра, при който играчът може да се състезава срещу друг играч, който играе на същата клавиетура, от същото устройство.
- Система за съхранение на най-бързите времена и имената на извършилителите им.

- Подобрения за колата, които играчът получава от предварително разположени на пистата spawner-и. Те биват 4 типа:
    - засилка, която еднократно ускорява колата в посоката, към която колата е насочена;
    - изстрел, който бива изстрелван от играчите и при удар в друг играч намалява точките им живот;
    - ремонт, който връща играча на максималното количество точки живот;
    - щит, който предотвратява намаляването на точките на живот на играчът за определено време.
- Подобренията трябва да могат и да се оставят на земята, и да могат да бъдат вземани от който и да е играч еднократно.
- Когато точките живот на един от играчите свършат той да бъде върнат по-назад по пистата с максимален брой точки живот.
- Система, която да предотвратява играчът да се движи в грешна посока и да прогресира по пистата.
- Система, която да върне играча на пистата, ако той излезе извън границите ѝ.

2.2. Аргументация за избора на развойната среда

2.2.1. Unreal Engine или Unity

За моята дипломна работа използвам Unreal Engine 5. Причини за това са:

- По-добрата поддръжка на триизмерни игри от страна на Unreal Engine.
- Системите за игра от няколко играча и съхраняването на прогреса на игрчът олесняват процесът на разработка.
- Възможността за по-хубави графики в Unreal Engine.
- Наличие на предишен опит с C++, който се използва в Unreal Engine.

2.2.2. Unreal Engine 4 или Unreal Engine 5

В Unreal Engine 5 има много нови техонологии, които подобряват триизмерните графики, осветлението и сенките на света. Основната промяна за това задание е промяната на предварително зададеният двигател за генериране на физики от PhysX на Chaos Physics, като ще бъде спряна поддръжката на PhysX в някоя от бъдещите версии. Това променя цялата структура на класовете за колата. Избирам Unreal Engine 5, за да може проектът да бъде поддържан за по-дълъг период от време. За развойна среда използвам препоръчаният от Unreal, Visual Studio 2019, защото има интеграция с Unreal Engine 5.

Трета глава

Реализация на проекта

3.1. Колата

Колата наследява клас AWheeledVehiclePawn, който съдържа UChaosVehicleMovementComponent и USkeletalMeshComponent. В UChaosVehicleMovementComponent се пазят данни, които се използват за изчисляването на движението на колата. Пример за такива данни са теглото на колата, броят на гумите, кои от гумите задвижват колата, на колко градуса могат да завиват гумите, кои от гумите могат да завиват, крива на въртящият момент, дали автоматично да се сменят предавките, на колко оборота да сменя предавките автоматично. В USkeletalMeshComponent се задават анимациите на колата, как изглежда колата, колко тежи колата къде е центърът на тежестта на колата.

```
UCLASS(abstract, config=Game, BlueprintType)
class CHAOSVEHICLES_API AWheeledVehiclePawn : public APawn
{
    GENERATED_UCLASS_BODY()

private:
    /**  The main skeletal mesh associated with this Vehicle */
    UPROPERTY(Category = Vehicle, VisibleDefaultsOnly, BlueprintReadOnly, meta = (AllowPrivateAccess = "true"))
    class USkeletalMeshComponent* Mesh;

    /** vehicle simulation component */
    UPROPERTY(Category = Vehicle, VisibleDefaultsOnly, BlueprintReadOnly, meta = (AllowPrivateAccess = "true"))
    class UChaosVehicleMovementComponent* VehicleMovementComponent;
public:

    /** Name of the MeshComponent. Use this name if you want to prevent creation of the component (with ObjectInitializer.DoNotCreateDefaultSubobject).
    static FName VehicleMeshComponentName;

    /** Name of the VehicleMovement. Use this name if you want to use a different class (with ObjectInitializer.SetDefaultSubobjectClass). */
    static FName VehicleMovementComponentName;
```

Фиг. 3.1.1. Променливите в класа AWheeledVehiclePawn.

Колата, която се използва в играта, заменя даденият от AWheeledVehiclePawn UChaosVehicleMovementComponent компонент, с написан от мен компонент, наследяващ UChaosVehicleMovementComponent. Добавените функции в него се ползват за промяната на променливи, които не могат да бъдат достъпени директно от други класове.

За движение на колата се използват функциите Accelerate и Steer, които приемат входна стойност от -1 до 1 и викат нужната функция, която изчислява как трвбва да се придвижи колата на следващият ход, като преди това обработват входа, за да се подаде верен вход на функциите, изчисляващи следващата позиция на колата.

```
61
62  void AMyWheeledVehiclePawn::Accelerate(float AxisValue)
63  {
64      if (AxisValue >= 0) {
65          GetVehicleMovementComponent()->SetThrottleInput(AxisValue);
66          GetVehicleMovementComponent()->SetBrakeInput(0);
67      }
68      if (AxisValue < 0) {
69          AxisValue = AxisValue * -1;
70          GetVehicleMovementComponent()->SetBrakeInput(AxisValue);
71          GetVehicleMovementComponent()->SetThrottleInput(0);
72      }
73  }
74  void AMyWheeledVehiclePawn::Steer(float AxisValue)
75  {
76      GetVehicleMovementComponent()->SetSteeringInput(AxisValue);
77  }
```

Фиг. 3.1.2. Имплементация на функциите Accelerate и Steer.

3.2. Камерата

За камерата имаме UCameraComponet, който се прикачва към колата с помощта на USpringArmComponent. Закачваме го за USpringArmComponent, за да предотвратим преминаването на камерата зад обекти, които служат като стени на пистата. За смяната на камерата имаме две променливи – camera_cycle и max_cameras. Camera_cycle служи за определяне на сегашната камера и коя трябва да бъде следващата камера след натискане на клавиша за смяна на камерата. Max_cameras служи за определяне на броят камери, от които може да избира потребителят. Стойността на max_cameras е винаги по-голяма от тази на camera_cycle.

```
if (camera_cycle == 0) {
    OurCameraComponent->SetRelativeLocation(FVector(200.0f, 0.0f, 100.0f));
    OurCameraComponent->SetRelativeRotation(FRotator(0.0f, 0.0f, 0.0f));
}
if (camera_cycle == 1) {
    OurCameraComponent->SetRelativeLocation(FVector(-450.0f, 0.0f, 350.0f));
    OurCameraComponent->SetRelativeRotation(FRotator(-15.0f, 0.0f, 0.0f));
}
camera_cycle = camera_cycle + 1;

if (camera_cycle == max_cameras) {
    camera_cycle = 0;
}
```

Фиг. 3.2.1. Имплементация на функцията ChangeCam.

Самото сменяне на камерата се случва във функцията ChangeCam, която мести камерата на определяна позиция, в зависимост от стойността на camera_cycle.

3.3. Режим на игра, при който играчът преминава трасето за по-кратко от зададеното време

```
protected:

    UPROPERTY(EditDefaultsOnly, Category = "GameStart")
    TSubclassOf<AMyWheeledVehiclePawn> PawnSpawnClass;

    UPROPERTY(EditAnywhere)
    int LapCount = 1;

    UPROPERTY(EditAnywhere)
    float TimePerLap = 200.f;

public:

    void StartPlay();

    void EndMatch() override;

    UFUNCTION()
    int GetLapCount();

    UFUNCTION()
    void SetLapCount(int Laps);

    UFUNCTION()
    float GetTimePerLap();

    UFUNCTION()
    void SetTimePerLap(float Time);
```

Фиг. 3.3.1. Променливи и функции на клас ASinglePlayerGameModeBase.

PawnSpawnClass се използва за да знае от какъв клас ще бъде количката, която режимът на игра създава.

LapCount се използва за броят обиколки, които играчът трябва да направи, за да финишира. Ако стойността не е променена с SetLapCount(int Laps) ще има само една обиколка.

TimePerLap се използва за да се определи колко секунди ще има играчът за изминаване на една обиколка. Крайното време на играча трябва да е по-малко от времето за една обиколка, умножено по броят обиколки, за да спечели.

Функциите StartPlay() и EndMatch се викат в началото и края на състезанието. В StartPlay() се създава финалната точка, колата, контролера, който ще приема входа на играча и ще извиква функциите на колата спрямо този вход. Освен това, функцията подава на колата указател към финалната точка и зарежда асинхронно и най-бързите времена в конзолата.

```
void ASinglePlayerGameModeBase::StartPlay() {

    Super::StartPlay();
    UWorld* World = GetWorld();
    UGameplayStatics::CreatePlayer(World, 0, true);
    APlayerController* Controller1 = UGameplayStatics::GetPlayerController(World, 0);
    AActor* Player1Start = FindPlayerStart(Controller1, "P1");
    AMyWheeledVehiclePawn* Player1 = World->SpawnActor<AMyWheeledVehiclePawn>(PawnSpawnClass, Player1Start->GetActorLocation(), Player1Start->GetActorRotation());


    Controller1->Possess(Player1);

    AFinalCheckpoint* FinishLine = World->SpawnActor<AFinalCheckpoint>(FVector(Player1Start->GetActorLocation().X - 1500, Player1Start->GetActorLocation().Y - 500, Player1Start->GetActorLocation().Z), Player1Start->GetActorRotation());
    FinishLine->BoxComp->SetBoxExtent(FVector(32, 2000, 2000));

    if (Player1->GetController() == nullptr)
    {
        UE_LOG(LogTemp, Error, TEXT("Player 1 not initialized"));
    }
    else
    {
        Player1->SetWrongDirectionCheckpoint(FinishLine);
        Player1->SetMaxLaps(LapCount);
    }

    FAsyncLoadGameFromSlotDelegate LoadedDelegate;
    LoadedDelegate.BindUObject(Player1, &AMyWheeledVehiclePawn::LoadGameDelegateFunction);
    UGameplayStatics::AsyncLoadGameFromSlot("defalut", 0, LoadedDelegate);
}
```

Фиг. 3.3.2. Имплементация на функция ASinglePlayerGameModeBase::StartPlay.

Функцията EndMatch() проверява дали играчът е спечелил или загубил и след това приключва играта.

3.4. Режим на игра, при който играчът може да се състезава срещу друг играч, който играе на същата клавиатура, от същото устройство

В този режим на игра се случва същото, като в първият, само дето се създават два играча вместо един. Екранът се разделя на две части от Unreal Engine, по подразбиране. Единственият проблем е че входът от клавиатурата може да бъде подаден само на един контролер, докато ние използваме два. За да можем да подадем входа от клавиатурата на няколко контролера се използва собствен Viewport. В него се презаписват функциите InputKey(const FInputKeyEventArgs& EventArgs) и InputAxis(FViewport* InputViewport, int32 ControllerId, FKey Key, float Delta, float DeltaTime, int32 NumSamples, bool bGamepad). В новата имплементация на тези функции входът от клавиатурата бива подаден на всички контролери.

```
bool UMyLocalMPGameViewportClient::InputKey(const FInputKeyEventArgs& EventArgs)
{
    UEngine* const Engine = GetOuterUEngine();
    int32 const NumPlayers = Engine ? Engine->GetNumGamePlayers(this) : 0;
    bool bRetVal = false;
    for (int32 i = 0; i < NumPlayers; i++)
    {
        FInputKeyEventArgs arg = FInputKeyEventArgs(EventArgs.Viewport, i, EventArgs.Key, EventArgs.Event, EventArgs.AmountDepressed, EventArgs.bIsTouchEvent);
        bRetVal = Super::InputKey(arg);
        bRetVal;
    }

    return bRetVal;
}
```

Фиг. 3.4.1. Имплементация на функция InputKey.

```cpp
bool UMyLocalMPGameViewportClient::InputAxis(FViewport* InputViewport, int32 ControllerId, FKey Key, float Delta, float DeltaTime, int32 NumSamples, bool bGamepad)
{
	{
		UEngine* const Engine = GetOuterUEngine();
		int32 const NumPlayers = Engine ? Engine->GetNumGamePlayers(this) : 0;
		bool bRetVal = false;
		for (int32 i = 0; i < NumPlayers; i++)
		{
			bRetVal = Super::InputAxis(InputViewport, i, Key, Delta, DeltaTime, NumSamples, bGamepad);
			bRetVal;
		}

		return bRetVal;
	}
}
```

Фиг. 3.4.2. Имплементация на функция InputAxis.

Контролерите различават кой вход за кой играч е по клавишите, които се натискат (те са различни за двата играча). В зависимост от индекса на контролера се определя кой контролер е за кой играч.

```cpp
void ACarPlayerControler::OnPossess(APawn* InPawn)
{

	Super::OnPossess(InPawn);
	AMyWheeledVehiclePawn* Instanced = Cast<AMyWheeledVehiclePawn>(InPawn);
	if (IsValid(Instanced) == false) {
		UE_LOG(LogTemp, Error, TEXT("Wrong Pawn"));
		return;
	}
	if (GetLocalPlayer()->GetControllerId() == 0)
	{
		InputComponent->BindAxis("MySteering", Instanced, &AMyWheeledVehiclePawn::Steer);
		InputComponent->BindAxis("MyThrottle", Instanced, &AMyWheeledVehiclePawn::Accelerate);
		InputComponent->BindAction("ChangeCam", IE_Pressed, Instanced, &AMyWheeledVehiclePawn::ChangeCam);
		InputComponent->BindAction("Reset", IE_Pressed, Instanced, &AMyWheeledVehiclePawn::Reset);
		InputComponent->BindAction("UsePowerup", IE_Pressed, Instanced, &AMyWheeledVehiclePawn::UseSelectedPowerup);
		InputComponent->BindAction("ChangeSlot", IE_Pressed, Instanced, &AMyWheeledVehiclePawn::NextPowerupSlot);
		InputComponent->BindAction("PowerupDrop", IE_Pressed, Instanced, &AMyWheeledVehiclePawn::DropPowerup);


	}
	if (GetLocalPlayer()->GetControllerId() == 1)
	{
		//UE_LOG(LogTemp, Error, TEXT("it works?"));
		InputComponent->BindAxis("P2_Steering", Instanced, &AMyWheeledVehiclePawn::Steer);
		InputComponent->BindAxis("P2_Throttle", Instanced, &AMyWheeledVehiclePawn::Accelerate);
		InputComponent->BindAction("P2_ChangeCam", IE_Pressed, Instanced, &AMyWheeledVehiclePawn::ChangeCam);
		InputComponent->BindAction("P2_Reset", IE_Pressed, Instanced, &AMyWheeledVehiclePawn::Reset);
		InputComponent->BindAction("P2_UsePowerup", IE_Pressed, Instanced, &AMyWheeledVehiclePawn::UseSelectedPowerup);
		InputComponent->BindAction("P2_ChangeSlot", IE_Pressed, Instanced, &AMyWheeledVehiclePawn::NextPowerupSlot);
		InputComponent->BindAction("P2_PowerupDrop", IE_Pressed, Instanced, &AMyWheeledVehiclePawn::DropPowerup);
	}

}
```

Фиг. 3.4.3. Имплементация на контролера.

В контролера се обързва входът с функциите на играча. Този контролер се ползва и в двата режима на игра, защото когато има само един играч има и само един контролер. За този играч ще работят само клавишите за играч 1 от режима на игра за двама играча.

3.5. Предотвратяване на движение назад, връщане на пистата

За да се подсигури че колата на играча се движи в правилната посока са разположени точки, отчитащи прогрес. В колата се пазят две такива точки – последните две преминати. Едната се пази за координатите, на които трябва да се върне колата ако играчът иска да се върне там. Другата е точката, която ако преминеш се връщаш назад. Ако я преминеш играта ще те върне принудително на координатите на първата точка. В играча се пазят и координатите на които трябва да се върне.

```
void AMyWheeledVehiclePawn::Reset()
{
    UE_LOG(LogTemp, Error, TEXT("start y = %f"), ResetLocation.Y);
    UWorld* MyWorld = GetWorld();
    FVector Offset = GetRootComponent()->Bounds.Origin - GetActorLocation();
    ResetLocation = ResetLocation + Offset;
    bool bTeleportSucceeded = MyWorld->FindTeleportSpot(this, ResetLocation, ResetRotation);
    ResetLocation = ResetLocation - Offset;
    if (bTeleportSucceeded)
    {
        this->GetMovementComponent()->Deactivate();
        this->TeleportTo(ResetLocation, ResetRotation);
        this->GetMovementComponent()->Activate();
        UE_LOG(LogTemp, Error, TEXT("end1 y = %f"), ResetLocation.Y);
        Hp = MaxHp;
        return;
    }

    //attempts to find a new spawn
    ResetLocation.Y = ResetLocation.Y + spawn_offset;
    ResetLocation = ResetLocation + Offset;
    bTeleportSucceeded = MyWorld->FindTeleportSpot(this, ResetLocation, ResetRotation);
    ResetLocation = ResetLocation - Offset;
    if (bTeleportSucceeded)
    {
        this->GetMovementComponent()->Deactivate();
        this->TeleportTo(ResetLocation, ResetRotation);
        this->GetMovementComponent()->Activate();
        ResetLocation.Y = ResetLocation.Y - spawn_offset;
        UE_LOG(LogTemp, Error, TEXT("end2 y = %f"), ResetLocation.Y);
        Hp = MaxHp;
        return;
    }
    else
    {
        ResetLocation.Y = ResetLocation.Y - spawn_offset;
    }
```

Фиг. 3.5.1. Имплементация на функция за връщане на играча на определена точка.

Във функцията за връщане на определена точка първо се прави проверка, дали колата ще може да се побере на мястото, на което е точката за връщане. Ако не успее, се проверява дали може да го върне на spawn_offset (константа, намираща се в колата) разтояние от точката. Ако не успее отново, колата си остава на мястото, на

което е била досега. Ако успее, тя бива телепортирана на мястото без засилка в която и да е посока. Когато играчът се движи по пистата, той преминава през тези точки, отчитащи прогреса. Когато преминава през тях, точките запазват указател към себе си в колата. Освен това се променя и точката, която определя дали се движиш в грешна посока.

```
void ACheckpoint::OnOverlapBegin(class UPrimitiveComponent* OverlappedComp, class AActor* OtherActor, class UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
    if ((OtherComp != NULL) && (OtherActor->ActorHasTag(FName("Player"))))
    {
        AMyWheeledVehiclePawn* Player = CastChecked<AMyWheeledVehiclePawn>(OtherActor);
        if (Player->GetWrongDirectionCheckpoint() == this)
        {
            UE_LOG(LogTemp, Error, TEXT("sad"));
            Player->Reset();
            return;
        }
        if (Player->GetResetCheckpoint() != this)
        {
            UE_LOG(LogTemp, Error, TEXT("%s passed a checkpoint"), *(OtherActor->GetActorNameOrLabel()));
            Player->SetResetCheckpoint(this);
        }

    }
}
```

Фиг. 3.5.2. Имплементация на функция OnOverlapBegin.

```
void AMyWheeledVehiclePawn::SetResetCheckpoint(AActor* NewCheckpoint)
{
    if (NewCheckpoint != nullptr)
    {
        if (ResetCheckpoint != nullptr) //if player has passed a checkpoint
        {
            WrongDirectionCheckpoint = ResetCheckpoint;
        }
        ResetCheckpoint = NewCheckpoint;
        ResetLocation = ResetCheckpoint->GetActorLocation();
        ResetRotation = ResetCheckpoint->GetActorRotation();
    }

}
```

Фиг. 3.5.3. Имплементация на функция за задаване на точка, отчитаща прогреса.

В себе си точките, отчитащи прогреса съдържат само UBoxComponent, чрез който може да се зададе как да

взаимодействат с остатъка от света. В случая те пропускат всички и не са под въздействието на гравитация или други сили.

```
ACheckpoint::ACheckpoint()
{
    // Set this actor to call Tick() every frame.  You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;
    BoxComp = CreateDefaultSubobject<UBoxComponent>(TEXT("BoxComp"));
    BoxComp->BodyInstance.SetCollisionProfileName("OverlapAll");
    BoxComp->OnComponentBeginOverlap.AddDynamic(this, &ACheckpoint::OnOverlapBegin);

    RootComponent = BoxComp;
}
```

Фиг. 3.5.4. Конструктор на точка за отчитане на прогрес.

3.6. Финална точка и обиколки

В колата се пазят броя обиколки в състезанието, сегашната обиколка, кога е почнала последната обиколка и времето за най- бързата обиколка. Времето за най-бърза обиколка започва със служебна стойност -1, за да се знае че все още не е завършена нито една обиколка. Кога е почнала последната обиколка се задава, когато започне играта и когато се приключи обиколка. Броят обиколки в състезанието бива зададен от режима на игра след създаването на колата.

Финалната точка наследява обикновената точка, отчитаща прогреса. Разликата при нея е, че се инициализира от режима на игра, за да може да се подаде като указател за точката, гледаща дали се движи колата в правилната посока, на колата. Единствената разлика е в логиката на OnOverlapBegin(class UPrimitiveComponent* OverlappedComp, class AActor* OtherActor, class UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult), където освен че се вика void AMyWheeledVehiclePawn::SetResetCheckpoint(AActor* NewCheckpoint), се вика и void AMyWheeledVehiclePawn::CompleteLap().

```
void AMyWheeledVehiclePawn::CompleteLap()
{
    float seconds = this->GetGameTimeSinceCreation() - this->GetLapStartTime();
    int minutes = 0;
    if (FastestLap == -1)
    {
        FastestLap = seconds;
    }
    if (seconds < FastestLap)
    {
        FastestLap = seconds;
    }
    while (seconds > 60.f)
    {
        minutes = minutes + 1;
        seconds = seconds - 60.f;
    }
     UE_LOG(LogTemp, Warning, TEXT("LapTime: %2d.%2.2f"), minutes, seconds);
     SetLapStartTime(this->GetGameTimeSinceCreation());

    if (CurrentLap == MaxLaps)
    {
        UE_LOG(LogTemp, Error, TEXT("Finished race"));
        if (UFastestTimeSaveGame* LoadedGame = Cast<UFastestTimeSaveGame>(UGameplayStatics::LoadGameFromSlot("defalut", 0)))
        {
            LoadedGame->Times.Add(FastestLap, FString("player 1"));
            LoadedGame->Times.KeySort([](float A, float B) { return A < B; });
        }
        else if(UFastestTimeSaveGame* SaveGameInstance = Cast<UFastestTimeSaveGame>(UGameplayStatics::CreateSaveGameObject(UFastestTimeSaveGame::StaticClass())))
        {
            SaveGameInstance->Times.Add(FastestLap, FString("player 1"));
            SaveGameInstance->Times.KeySort([](float A, float B) { return A < B; });
            if (UGameplayStatics::SaveGameToSlot(SaveGameInstance, "defalut", 0))
            {


            }
        }
        CastChecked<AGameMode>(GetWorld()->GetAuthGameMode())->EndMatch();
    }
    CurrentLap++;
}
```

Фиг. 3.6.1. Имплементация на функция CompleteLap.

В тази функция се взима времето за последната обиколка, като се извади от времето, за което колата е съществувала, времето, което ѝ е отнело да започне тази обиколка. След това проверява дали това е най-бързата обиколка и ако е, записва в променливата за най-бърза обиколка колко време е отнело за изминаване на дадената обиколка. След това се прави проверка дали това е била последната обиколка за състезанието. Ако е била, се запазва най-бързата обиколка, и играта свършва. Ако не е била последната обиколка, тогава променливата, която пази сегашната обиколка, се увеличава с 1.

3.7. Запазване и четене на най-бързо време

```
UCLASS()
class DIPLOMNA_API UFastestTimeSaveGame : public USaveGame
{
    GENERATED_BODY()


public:
    UFastestTimeSaveGame();

    UPROPERTY(EditDefaultsOnly)
    TMap<float, FString> Times;
};
```

Фиг. 3.7.1. Формат на SaveGame.

Класът UFastestTimeSaveGame пази в себе си само карта, съдържаща най-бързото време в секунди и името на играчът извършил го. Най-бързите извършени времена се зареждат асинхронно и излизат в конзолата при началото на играта. Това действие се извършва от режима на игра.

```
    FAsyncLoadGameFromSlotDelegate LoadedDelegate;
    LoadedDelegate.BindUObject(Player1, &AMyWheeledVehiclePawn::LoadGameDelegateFunction);
    UGameplayStatics::AsyncLoadGameFromSlot("defalut", 0, LoadedDelegate);
}
```

Фиг. 3.7.2. Асинхронно зареждане на най-бързи времена.

Запазването на най-бързото време се случва, когато играчът приключи състезанието. Запазва се само времето от най-бързата обиколка, след което се сортират, така че най-бързото време да е с индекс 0.

```
if (UFastestTimeSaveGame* LoadedGame = Cast<UFastestTimeSaveGame>(UGameplayStatics::LoadGameFromSlot("defalut", 0)))
{
    LoadedGame->Times.Add(FastestLap, this->GetActorNameOrLabel());
    LoadedGame->Times.KeySort([](float A, float B) { return A < B; });
}
else if(UFastestTimeSaveGame* SaveGameInstance = Cast<UFastestTimeSaveGame>(UGameplayStatics::CreateSaveGameObject(UFastestTimeSaveGame::StaticClass())))
{
    SaveGameInstance->Times.Add(FastestLap, GetActorNameOrLabel());
    SaveGameInstance->Times.KeySort([](float A, float B) { return A < B; });
    if (UGameplayStatics::SaveGameToSlot(SaveGameInstance, "defalut", 0))
    {

    }
}
```

Фиг. 3.7.3. Запазване на най-бърза обиколка.

3.8. Подобрения – базов клас

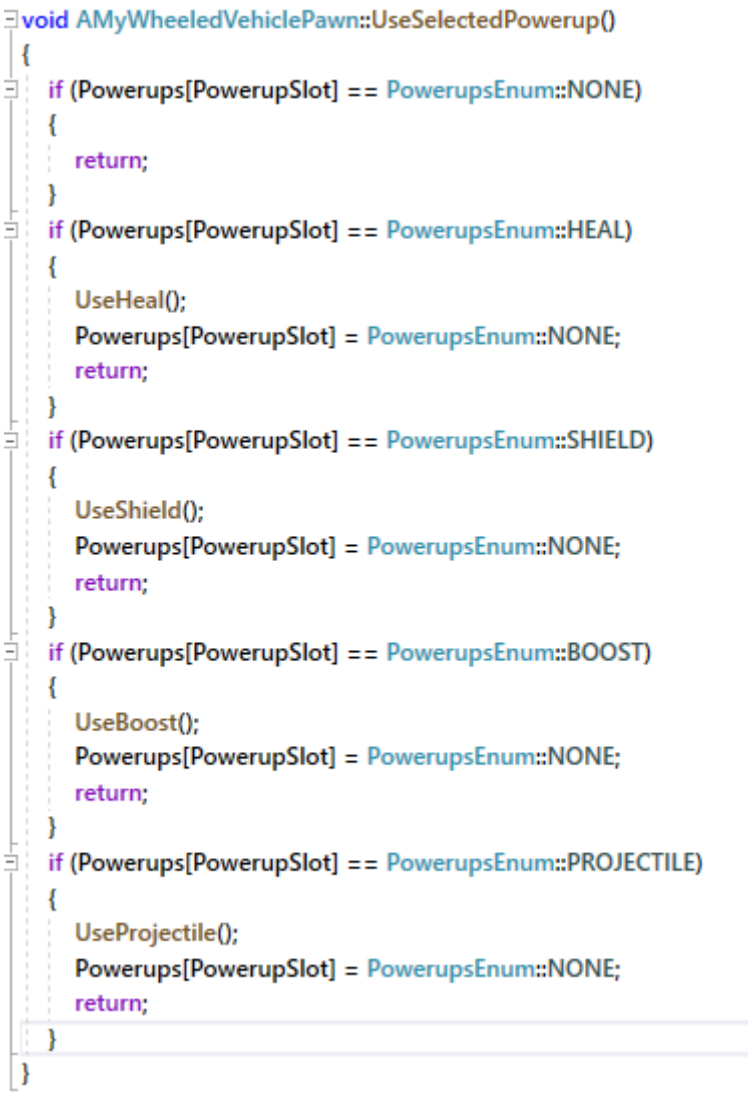
Подобренията представляват Enum, благодарение на който разпознаваме какво подобрение имаме.

```
#include "CoreMinimal.h"
#include "PowerupsEnum.generated.h"

UENUM(BlueprintType)
enum class PowerupsEnum : uint8
{
    NONE = 0 UMETA(DisplayName = "None"),
    BOOST = 1 UMETA(DisplayName = "Boost"),
    PROJECTILE = 2 UMETA(DisplayName = "Projectile"),
    SHIELD = 3 UMETA(DisplayName = "Shiled"),
    HEAL = 4 UMETA(DisplayName = "Heal")
};
```

Фиг. 3.8.1. PowerupsEnum – стойности.

В колата се пази списък от PowerupsEnum с размер 3, защото имаме място за 3 подобрения. Освен това, в колата се пази размерът на масива и кое подобрение ще бъде използвано, ако използваме едно от подобренията. Функцията, която достъпва играчът, е void AMyWheeledVehiclePawn::UseSelectedPowerup(). Тя проверява дали на избрания индекс имаме подобрение, и ако имаме, то извиква функция, в зависимост от подобрението, след което го променя на стойност NONE.



```
void AMyWheeledVehiclePawn::UseSelectedPowerup()
{
    if (Powerups[PowerupSlot] == PowerupsEnum::NONE)
    {
        return;
    }
    if (Powerups[PowerupSlot] == PowerupsEnum::HEAL)
    {
        UseHeal();
        Powerups[PowerupSlot] = PowerupsEnum::NONE;
        return;
    }
    if (Powerups[PowerupSlot] == PowerupsEnum::SHIELD)
    {
        UseShield();
        Powerups[PowerupSlot] = PowerupsEnum::NONE;
        return;
    }
    if (Powerups[PowerupSlot] == PowerupsEnum::BOOST)
    {
        UseBoost();
        Powerups[PowerupSlot] = PowerupsEnum::NONE;
        return;
    }
    if (Powerups[PowerupSlot] == PowerupsEnum::PROJECTILE)
    {
        UseProjectile();
        Powerups[PowerupSlot] = PowerupsEnum::NONE;
        return;
    }
}
```

Фиг. 3.8.2. Имплементация на UseSelectedPowerup.

3.9. Подобрение – изстрел

Колата има точки живот. Когато тя бива улучена от подобрението изстрел, точките живот на колата спадат с 1. Когато точките живот на колата стигнат 0 тя бива принудително върната на последната измината точка за отчитане на прогреса, след което точките живот се връщат на максимума. За да бъде създаден този изстрел, колата пази в себе си класа на изстрела. Изстрелът в себе си съдържа:

- UprojectileMovementComponent, чрез който му задаваме накъде и колко бързо да се движи;
- UstaticMeshComponent, за да можем да видим куршума;
- UBoxComponent, за да можем да разпознаем, че се пресича с друг обект.

Когато изстрелът се пресече с друг обект той се разрушава. От своя страна колата също проверява дали не се пресича с изстрела, и ако се пресече с него, се намаляват точките живот на колата.

```
void AMyWheeledVehiclePawn::UseProjectile()
{
    FVector SpawnLocation = GetActorLocation();
    FRotator SpawnRotation = GetActorRotation();
    SpawnLocation = SpawnRotation.RotateVector(FVector(400, 0, 50)) + SpawnLocation;//spawns the projectile in front of the car, no matter the rotation
    GetWorld()->SpawnActor(this->ProjectileClass, &SpawnLocation, &SpawnRotation);
    return;
}
```

Фиг. 3.9.1. Имплементация на подобрението „изстрел“.

3.10. Подобрение – ремонт

Колата в себе си пази максималният си брой точки живот. Когато бива използвано това подобрение, точките живот на колата стават равни на максималният брой точки живот на колата.

3.11. Подобрение – засилка

Колата бива засилена в посоката, в която тя е насочена. Импулсът действа върху цялото тяло и променя скоростта му с тези параметри.

```
void AMyWheeledVehiclePawn::UseBoost()
{
    this->GetMesh()->AddImpulse(this->GetActorRotation().RotateVector(FVector(1000, 0, 0)), "None", true);
}
```

Фиг. 3.11.1. Имплементация на подобрение – засилка.

3.12. Подобрение – щит

Колата пази в себе си int Shield, който е 0, когато играчът няма щит и повече от 0, когато играчът има щит.

```
void AMyWheeledVehiclePawn::UseShield()
{
    this->SetShield(this->GetShield() + 1);
    FTimerHandle Timer;
    FTimerDelegate Delegate;
    Delegate.BindUObject(this, &AMyWheeledVehiclePawn::SetShield, this->GetShield()-1);
    GetWorld()->GetTimerManager().SetTimer(Timer,Delegate, 5.f, false);
    UE_LOG(LogTemp, Warning, TEXT("Shield started"));
}
```

Фиг. 3.12.1. Имплементация на подобрение – щит.

Функцията UseShield() увеличава променливата Shield с 1 и след 5 секунди я намалява с 1. Променливата Shield е int, а не bool, защото играчът може да използва подобрението щит, докато същото подобрение се използва. Така щитът ще продължи за 5 секунди след активацията на следващ щит, а не за 5 секунди след активацията на първия/предходния щит. Общата дълготрайност на щитовете не се събира.

3.13. Подобрения – spawner

Този обект позволява да бъдат взимани подобрения от пистата веднъж на определен период от време. В себе си съдържа:

- UStaticMeshComponent – за да се вижда къде е подобрението, какво е и дали може да бъде взето;
- PowerupsEnum – за да знае какво подобрение трябва да даде на играчите;
- bCanBeCollected – за да знае дали подобрението можеда бъде взето;
- UBoxComponent – за да може да отчита дали някой от играчите е преминал през него и е взел подобрението.

```
void APowerupSpawner::OnOverlapBegin(class UPrimitiveComponent* OverlappedComp, class AActor* OtherActor, class UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{

    if (bCanBeCollected) {
        if ((OtherComp != NULL) && OtherActor->ActorHasTag(FName("Player")))
        {
            AMyWheeledVehiclePawn* Player = CastChecked<AMyWheeledVehiclePawn>(OtherActor);
            for (int i = 0; i < Player->GetMaxPowerupSlots(); i++)
            {
                if (Player->GetPowerup(i) == PowerupsEnum::NONE)
                {
                    Player->SetPowerup(Powerup, i);
                    bCanBeCollected = false;
                    FTimerHandle Handle;
                    GetWorld()->GetTimerManager().SetTimer(Handle, this, &APowerupSpawner::Respawn, 2.f, false);
                    this->Mesh->SetVisibility(false, false);
                    return;
                }
            }
        }
    }
}

void APowerupSpawner::Respawn()
{
    bCanBeCollected = true;
    this->Mesh->SetVisibility(true, false);
}
```

Фиг. 3.13.1. Имплементация на логиката на spawner за подобрения.

Когато някоя от колите премине през spawner-а, той проверява дали може да бъде взет, след това проверява дали колата има свободно

място за новото подобрение. Ако и двете изисквания са изпълнени, то тогава колата получава това подобрение и spawner-ът се изключва за две секунди с помощта на Timer. През това време spawner-ът не може да бъде видян.

3.14. Подобрения – изхвърляне и еднократно взимане

Освен че играчът може да използва дадено подобрение, той може и да го изхвърли. Това действие създава нов обект, който пази в себе си едно подобрение. Класът съдържа:

- UBoxComponent, за да се определи дали някой от играчите минава през този обект;
- UStaticMeshComponent, за да може да се види обектът от играчите;
- PowerupsEnum, за да знае какво подобрение трябва да даде на играча, ако той премине през него.

В класа на колата се проверява дали преминава през подобрение за еднократно взимане. Ако колата има свободно място, в което да вземе това подобрение, то колата ще го вземе, след което ще премахне обекта за еднократно взимане на подобрение.

```
if (OtherActor->IsA(ASinglePowerupDrop::StaticClass()))
{
    UE_LOG(LogTemp, Error, TEXT("by a powerupdrop"));
    for (int i = 0; i < MaxPowerupSlots; i++)
    {
        if (Powerups[i] == PowerupsEnum::NONE)
        {
            Powerups[i] = CastChecked<ASinglePowerupDrop>(OtherActor)->Powerup;
            OtherActor->Destroy();
            return;
        }
    }
}
```

Фиг. 3.14.1. Логика за преминаване през подобрение за еднократно взимане.

Колата също пази в себе си списък от подкласове на подобрението за еднократно взимане. Чрез тях колата знае какъв клас трябва да създаде, когато иска да изхвърли едно от подобренията си.

Четвърта глава

Ръководство на потребителя

4.1. Инсталация и препоръчителни изисквания

Отворете папката “Executables”. След това отворете папката “SP” за състезание срещу часовника или “MP” за състезание срещу друг човек на същата клавиатура. След това отворете “.exe” файла.

Препоръчителни изисквания към системата:

- RAM: 16GB
- CPU: Intel Core i7 2.6GHz
- OS: Windows 10
- GPU: Nvidia GeForce RTX 2060
- Storage: 2GB

4.2. Контроли

Играч 1/ Единствен играч:

W – движение напред

A – движение наляво

S – движение назад

D – движение надясно

C – смяна на камерата

E – използване на избраното подобрение

F – избиране на следващо място за подобрение

R – връщане на колата до последната точка, отчитаща прогреса

Q – изхвърляне на избраното подобрение

Играч 2:

Стрелка нагоре – движение напред

Стрелка наляво – движение наляво

Стрелка надолу – движение назад

Стрелка надясно – движение надясно

M – смяна на камерата

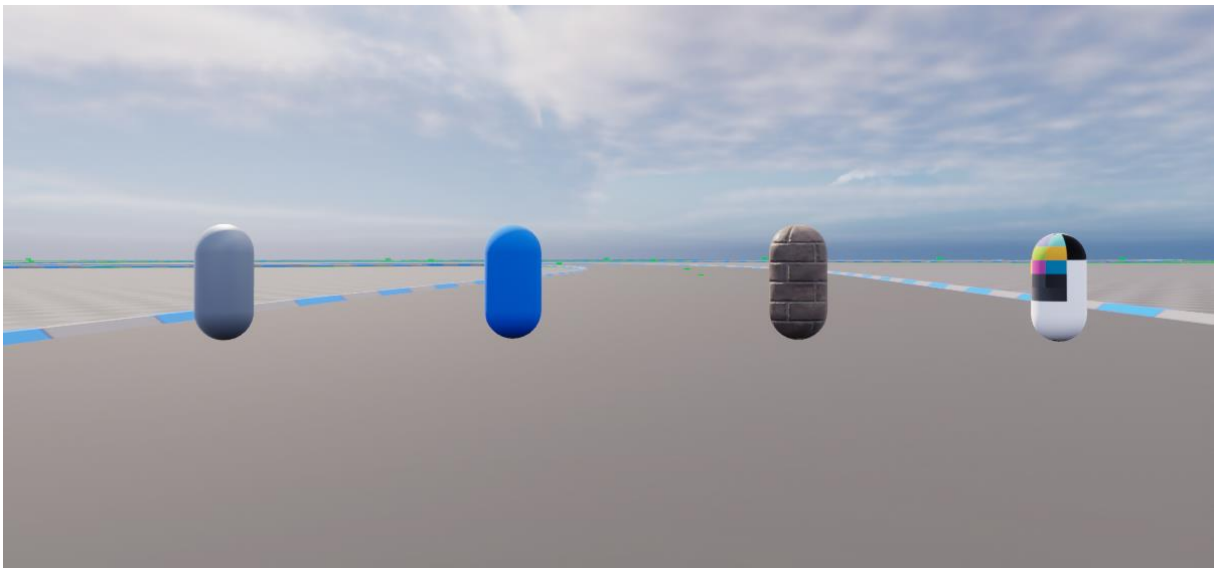
O – използване на избраното подобрение

L – избиране на следващо място за подобрение

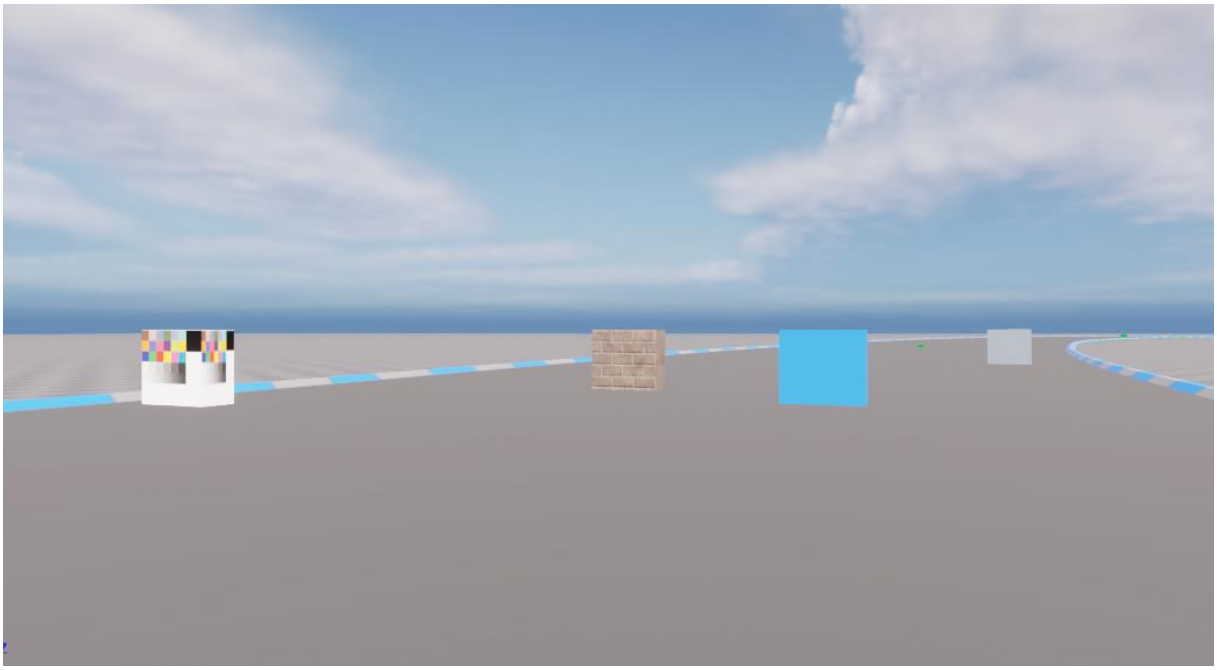
P – връщане на колата до последната точка, отчитаща прогреса

K – изхвърляне на избраното подобрение

4.3. Описание на подобренията



Фиг. 4.3.1. Spawner-и на подобрения.



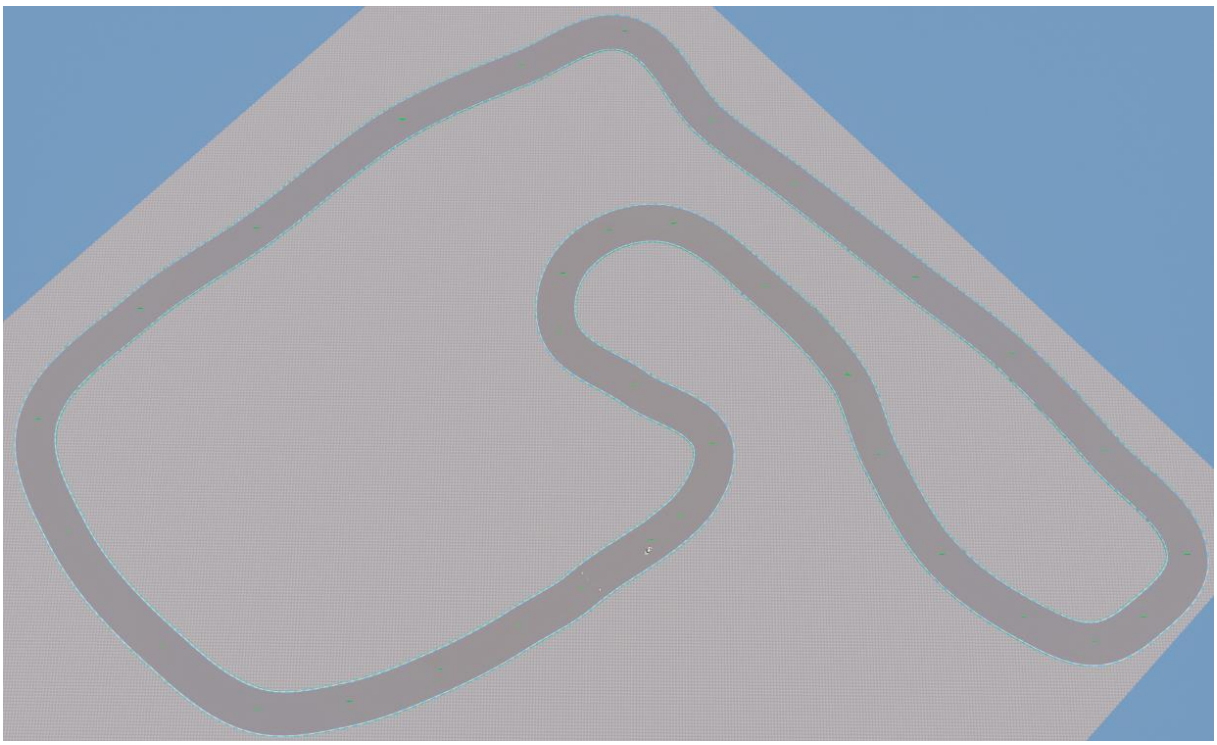
Фиг. 4.3.2. Подобрения за еднократно взимане.

Капсулите са spawner-и на подобрения. Кубовете са подобрения за еднократно взимане. В зависимост от цвета на подобрението играчът ще получи различно подобрение.

- Подобренията в синьо са ремонтите.
- Подобренията в сиво са засилките.
- Подобренията с вид на тухли са изтрелите.
- Подобренията с шареният вид са щитовете.

4.4. Описание на игровия процес

Целта на играта е, играчът да направи една обиколка на пистата за възможно най-кратко време.



Фиг. 4.4.1. Пистата, на която се извършват състезанията.

Заключение

Всички изисквания, споменати във Втора глава, точка 1, са изпълнени.

Бъдещето развитие в краткотраен план съдържа:

- Промяна на системата за запзване на най-бързи времена;
- Втора писта;
- Меню, от което може да се избира броят на обиколките в състезанието и пистата за провеждане на състезанието;
- Промяна на външния вид на пистата;
- Избор от още една кола.

Бъдещето развитие в дългосрочен план се състои от:

- Състезания с повече от двама играчи;
- Състезания срещу опоненти, контролирани от компютъра;
- Състезания срещу хора, играещи на други устройства.

Източници

Ben Tristem. Unity vs. Unreal: Which Game Engine is Best For You? https://blog.udemy.com/unity-vs-unreal-which-game-engine-is-best-for-you/

Daniel Buckley. Unity vs. Unreal – Choosing a Game Engine. GameDev Academy. 19.12.2022 https://gamedevacademy.org/unity-vs-unreal/

Fandom. Mario Kart Racing Wiki https://mariokart.fandom.com/wiki/Mario_Kart_8_Deluxe#Grand_Prix:

freeCodeCamp, What is Game Development? 28.12.2019 https://www.freecodecamp.org/news/what-is-game-development/

Unity официален уебсайт. https://unity.com/

Unreal Engine официален уебсайт https://docs.unrealengine.com

Wikipedia. Bizarre Creations https://en.wikipedia.org/wiki/Bizarre_Creations

Wikipedia. Blur. https://en.wikipedia.org/wiki/Blur_(video_game)

Wikipedia. History of video games https://en.wikipedia.org/wiki/History_of_video_games

Wikipedia. Mario Kart. https://en.wikipedia.org/wiki/Mario_Kart

Wikipedia. Mario Kart 8. https://en.wikipedia.org/wiki/Mario_Kart_8

Wikipedia. Unity. https://en.wikipedia.org/wiki/Unity_(game_engine)

Wikipedia. Unreal Engine. https://en.wikipedia.org/wiki/Unreal_Engine

Wikipedia. Video game development. https://en.m.wikipedia.org/wiki/Video_game_development#:~:text=Commercial%20game%20development%20began%20in,a%20full%20and%20complete%20game

Съдържание