ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

ДИПЛОМНА РАБОТА

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

Тема: Разработка на игра с Unreal Engine

Дипломант:
Васил Николаев Христов

Дипломен ръководител:
Александър Ангелов

СОФИЯ

2 0 2 2

ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

Дата на заданието: 14.12.2021 г. Утвърждавам:.....
Дата на предаване: 14.03.2022 г. /проф. д-р инж. Т. Василева/

ЗАДАНИЕ
за дипломна работа

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

на ученика Васил Николаев Христов от 12 А клас
Тема: Разработка на игра с Unreal Engine

Изисквания:
1. Ходова игра за двама души, притежаващи еднакъв набор от единици
2. Възможност за строене на сгради, които да дават бонуси в началото на всеки ход
3. Възможност за превземане на някои сгради от противника
4. Всеки тип единица да има отделна отговаряща на него сграда
5. Различни единици на полето ще имат различни свойства (бонуси, невъзможност да отвръщат на удар или подобни)

Съдържание:
3.1 Теоретична част
3.2 Практическа част
3.3 Приложение

Дипломант: .....
/ Васил Христов /

Ръководител: .....
/ Александър Ангелов /

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

МНЕНИЕ НА НАУЧНИЯ РЪКОВОДИТЕЛ

Дипломантът се е справил много добре. Изпълнил е поставените задачи и може да бъде допуснат до защита. Предлагам за рецензент Виктор Кетипов.

Дипломант: ……

/ Васил Христов /

Дата:

31.03.2022

Ръководител: …….

/ Александър Ангелов /

СЪКРАЩЕНИЯ

- UE4 / Unreal – Unreal Engine 4
- GE – Game engine
- ООП – Обектно ориентирано програмиране
- JVM – Java virtual machine
- GUI – Graphical user interface (Графичен потребителски интерфейс)

УВОД

Cisk – „casual“ стратегическа игра, създадена за приятели, които искат да разпускат заедно, играейки игри. Името е комбинация от имената на 2 други стратегически игри: Civilization, наричана накратко Civ, и Risk. Cisk е опит за обединяване на тези 2 толкова различни игри. В проекта се съчетават прост и лесен gameplay, характерен за Risk, но разширен с някои по-сложни механики, които се срещат в Civilization, като например различни типове единици и терен, както и строене на сгради. В последно време се наблюдава тенденция игрите да стават все по-трудни и сложни, затова и реших да направя нещо просто. Главната идея на играта е следната:

- Двама (в бъдеще може да се направи да поддържа и повече) играчи, които се редуват. Игрите заемат все по-голяма част от ежедневието на подрастващите, а и не само. Затова е необходимо те да имат добро влияние в някои важни сфери от живота, като например социалната. Контактуването с други хора, както и съревноваването с тях в приятелска среда, води до развиване на социалните умения на даден индивид.

- Всички играчи разполагат с еднакъв набор от единици. Това е важно, за да може играта да бъде балансирана, и нито една от двете страни, да не получава надмощие над другата. Ако двете страни не са еднакво силни в самото начало, то голяма част от игрите ще бъдат с ясен край, дори и все още да не са започнали.

- Някои от единиците получават бонуси ако се бият срещу определен тип противник. Това добавя още едно ниво на сложност и изисква малко по-развито стратегическо мислене. Дори и да искам, играта да е проста, прекаленото ѝ улесняване би довело до това, тя да стане скучна и безинтересна.

- Строене на сгради по картата. Те могат да бъдат 2 типа:

1. Производствени, които предоставят играча с ресурси и единици в началото на всеки ход. Всеки играч може сам да прецени какви сгради иска да построи. Например, играчът може да заложи на фабрики, които му предоставят допълнителни ресурси и му позволяват да строи още повече сгради. Това е форма на инвестиция, която може или да му се отплати щедро, или да му изиграе много лоша шега.

2. Укрепления, които предоставят бонуси на единиците, които се намират в тях. Те биха предоставили шанс на губещия играч да обърне положението в своя полза. Тъй като укрепленията са статични, те могат да предоставят доста добра възможност за отбрана.

- Превземане на сгради от противника. Не всички сгради ще могат да бъдат „откраднати“. Част от тях ще бъдат унищожени при превземане, но останалите могат да бъдат присвоени от нападателите. Така трябва да се мисли къде каква сграда да се постави, защото един грешен ход може да те постави в много лоша ситуация.

- Също така, реших да добавя още едно нещо от Risk – елемента на настолната игра. И двамата играчи получават цялата информация. Друг плюс е, че е нужно само едно устройство, за да може да се играе. Това я прави идеална за играене при пътуване или ходене на гости.

Какво представлява играта?

Cisk е стратегическа походова игра за двама души. В началото на всеки ход, играчите получават ресурси (за строене на нови сгради). Производствените сгради предоставят единици, а укрепленията им дават бонуси, като например допълнителна защита. Част от сградите могат да се превземат от противника, а други биват директно унищожени. Някои от типовете единици получават бонуси срещу друг определен тип, като това допринася за стратегическата дълбочина на играта. Играта приключва, когато един от двамата играчи остане без единици и производствени сгради.

ПЪРВА ГЛАВА

1. Среди и технологии за създаване на игри

1.1. Основни принципи и технологии за реализиране на игри

1.1.1. Game Engine

Unreal Engine 4

Фиг 1.1 Unreal Engine 4

Unreal Engine 4[1] или UE4 е един от най-популярните engine-и за разработка на desktop игри. Използва основно C++. Той е също така и може би най-развитият от комерсиалните GE-и, Наближаващият release на UE5 е добра новина, защото проектите, създадени в UE4 ще могат да се port-нат и доразвият с много нови технологии[2]. UE4 има много функционалности[3], като това му придава малко по-голяма сложност. Като цяло е по-добър за разработване на големи проекти, отколкото за малки indie игрички, като Cisk, но опитът с Unreal Engine би бил доста ценен в бъдеще.

Алтернативи на UE4

Unity

фиг 1.2 Unity

Unity другият претендент за #1 GE. Със сигурност е по-лесен за използване от Unreal, понеже използва C# (Много добър език за ООП). Unity е много по-добър за разработване на мобилни или 2D игри, а освен това е и фаворит сред създателите на indie игри. Един недостатък е, че не е open-source (за разлика от UE4). Други недостатъци са, че вградените tool-ове в Unreal са доста по-добри, както и фактът, че графиката в Unity е малко по-лоша.

1.1.2. Език за програмиране

C++

Фиг. 1.3 C++

Освен, че C++ е най-силният език в сферата на ООП, той е един от най-бързите езици[4]. Също така, C++ е един от най-използваните езици за създаване на игри. Много от „частните“ game engine-и използват C++, което го затвърждава като лидера в разработката на големи и високобюджетни игри.

Плюсовете на C++[5] включват:

- Широкоразпространен
- Изпълнява дословно (за разлика от някои езици от високо ниво)

Едно от главните предимства, но и недостатък на C++ е това, че управлението на динамичната памет (heap) става ръчно (за разлика от езици като Java). Това дава възможност програмата да изразходва много по-малко ресурси. За щастие в Unreal Engine има garbage collector, който разчиства памет, която не се използва.

Алтернативи на C++

Java

Фиг. 1.4 Java

Java е един от най-използваните езици за ООП, но не се поддържа от най-популярните GE-и. Една от причините за това е, че изисква много повече ресурси (процесорно време и памет), защото Java кодът не се компилира директно до машинен код, а е необходима виртуална машина (JVM) за да бъде компилиран и интерпретиран. Плюсовете са, че Java е много добре пригоден за ООП. За да го използваш, обаче трябва да се откажеш от използването на някой от по-популярните game engine-и, което би усложнило задачата до такова ниво, че разработването би отнело много повече време.

C#

Фиг. 1.5 C#

C# е много популярен език за разработване на игри. Поддържа се от два от по-големите GE-и: Unity и Godot. Все пак C++ е за предпочитане, понеже C# е пригоден за много работи, и не е толкова оптимизиран. Въпреки предимствата си и широкото си разпространение, главна причина дипломната работа да не написана на C# е, защото идеята ми беше да работя с Unreal Engine 4, а той работи само със C++.

1.1.3. Развойни среди

Microsoft Visual Studio

Фиг. 1.6 Visual Studio

Microsoft Visual Studio е default-ния editor за UE4. Инсталира се заедно с engine-а. Въпреки, че не е един от най-добрите редактори за C++, или поне не от мой личен опит, успява да свърши достатъчно добра работа. Единственият недостатък е, че му трябват допълнителни плъгини за да се ориентира добре в големи проекти или такива с много библиотеки.

1.2 Разучаване на технологиите, използвани в дипломната работа

1.2.1 C++ в Unreal Engine

Unreal Engine 4 е огромен. Включените библиотеки, класове и функции дават много готови функционалности и много възможности за реализиране на едно и също нещо. Много трудно е един човек да се запознае със целия GE, особено ако опитът му е един проект разработен за няколко месеца. В случай, че

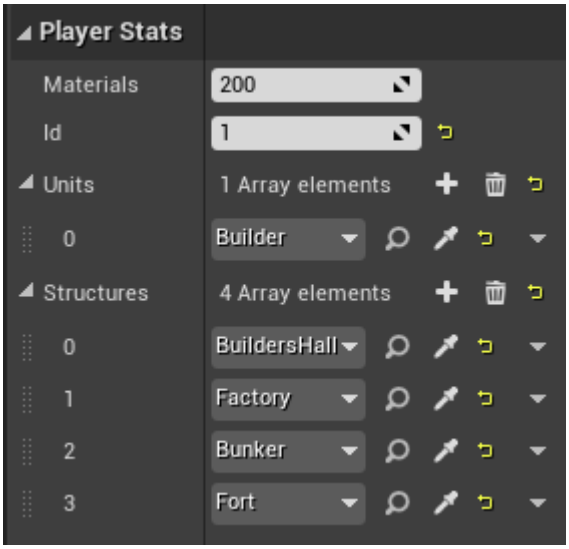
срещна трудности в имплементирането на някоя от функционалностите, проверявам в документацията на UE4[6], а ако и това не помогне, проверявам редица сайтове и форуми(Youtube, Reddit, StackOverflow, UE Forums[7]) за вече съществуващи решения.

Най-често в дипломната работа са използвани следните неща: Класа AActor, както и macro-тата UPROPERTY и UFUNCTION. Буквата A пред името на който и да е клас означава, че класа е или наследява Actor (Пример: ABaseUnitClass). Класа Actor е базов клас за обект, който може да бъде поставен в Level (Ниво/Карта за игра). Аналог на Actor в 2D игрите е Sprite. В UE4 има много функции за работа с Actor-и, а и самия клас има множество имплементирани методи, което прави работата с него доста приятна.

Макрото UPROPERTY вкарва променливата след него в системата за управление на памет на UE. Това позволява на garbage collector-а да работи с нея. Освен това, всички UPROPERTY променливи могат да бъдат променяни в Level Editor-а, което е особено полезно за първите обекти, поставени в едно ниво (В дипломната работа това са всички полета, на които ръчно са им зададени техните съседни, началните сгради и единици на двамата играчи, както и самите играчи). На UPROPERTY макрото могат да му се подадат няколко аргумента – категория, в която да бъдат показани променливите (това се използва в търсачките), дали могат да се достъпват от Blueprint class editor-а (описаното по-горе, Фиг. 1.8), и дали могат да се използват директно в Blueprint графове (Това често се нарича просто Blueprints).

```
UPROPERTY(EditAnywhere, Category = "Building")
    int DefBonus;
```

Фиг. 1.7 Декларация на UPROPERTY

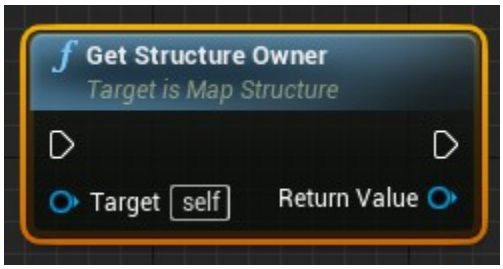


Фиг. 1.8 Редактор на UPROPERTY променливи

UFUNCTION е друго често използвано макро. То се поставя пред функция и позволява тя да бъде използвана в Blueprint графове (ако е подаден правилния за това параметър).

```
UFUNCTION(BlueprintCallable, Category = "Building")
    class ACISKPlayer* GetStructureOwner() {
    return this->StructureOwner;
}
```

Фиг. 1.9 Дефиниция на UFUNCTION



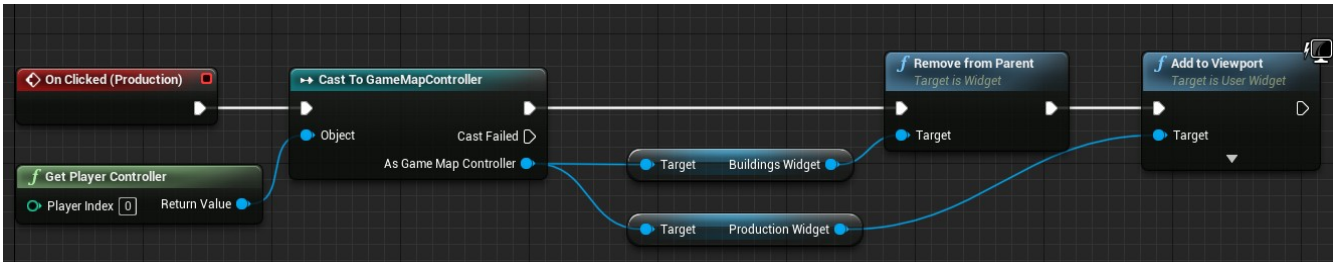
Фиг. 1.10 UFUNCTION в Blueprint граф

APawn е друг клас от UE4, който съм използвал. Той се използва за Actor-и, които приемат input от потребителя или AI. В дипломната работа класа ATile наследява APawn, като това предоставя възможност да се извършват дадени

действия, когато курсора на мишката премине върху дадено поле или когато се цъкне с ляв бутон върху него.

1.2.2 Blueprints в UE4

Blueprint влиза в категорията на така наречените “Visual programming” езици, което означава, че инструкциите не се задават с код, а с логически свързани блокове. Най-различни Blueprints се комбинират в Blueprint графове, като това позволява да се създадат сложни логика за управление на играта, без нито един ред код. Има цели игри, реализирани само чрез Blueprint, но това не е толкова практично, и затова дипломната работа не е изготвена по този начин.



Фиг. 1.11 Blueprint граф

На фиг. 1.11 е показан един от по-простите Blueprint графове, използвани в дипломната работа. Белите черти показват последователността на изпълнение на елементите, а всички останали (в случая са само сини), означават някакви стойности. На този граф, всички стойности са референции (указатели) към обекти, и затова са изобразени в синьо.

ВТОРА ГЛАВА

2. Проектиране на структурата на играта

2.1 Функционални изисквания:

2.1.1 Основни

- Ходова игра за двама души, притежаващи еднакъв набор от единици – Играчите се редуват в изпълнението на своите ходове. И двамата играчи разполагат със еднакви типове единици и сгради (няма различни страни/раси, като в някои други стратегически игри)
- Възможност за строене не сгради, които да дават бонуси в началото на всеки ход - Сградите могат да предоставят ресурси или единици на своя собственик, както и да дават допълнителни щети или защита на единиците разположени в тях.
- Възможност за превземане на някои сгради от противника – някои от сградите на противника могат да бъдат превзети.
- Всеки тип единица да има отделна отговаряща на него сграда – всеки тип единица бива създадена от различна сграда.
- Различни единици на полето ще имат различни свойства (бонуси, невъзможност да отвръщат на удар или подобни)

2.1.2 Други изисквания

- Да се създадат няколко на брой класове за различните типове сгради и единици. В дипломната работа са имплементирани по един основен клас със всички функционалности за сградите и единиците. Създаването на нов тип сграда или единица отнема минути, като единственото, което трябва да се направи за единица е да се променят стойностите (щети, живот и т.н.), а за

сграда да се напишат няколко реда код за да се окаже какво трябва да прави в края на всеки ход или какви бонуси да дава на единиците в нея. Също така трябва и да се даде или създаде 3D модел, който да се използва за изобразяване на дадената единица или сграда.

- Да се имплементира клас за полета (tile), върху които да седят сградите и единиците.
- Да се създадат класове, които да пазят информация за играчите и за играта като цяло.
- Да се добави движение на камерата при потребителски input.
- Да се добави графичен потребителски интерфейс, който да показва на играча повече информация за неговите ресурси и единици.
- Да се добавят менюта за строене на сгради.
- Да се създаде самата логика, която задвижва играта – това включва всички функции за строене на сгради, за редуване на ходовете на играчите, за вземането на данни, които да се изобразят на потребителския интерфейс.
- От играта се изисква да не се чупи/гърми/замръзва по време на нейното изпълнение.

2.2 Избор на технологии и развойна среда

2.2.1 Game engine

За реализирането на играта реших да използвам готов engine, защото това спестява много време и усилия. На готово получавам графичното изобразяване на играта както и много библиотеки, разработени специално за създаването на игри.
За разработването на 3D игра ми беше нужен добре развит и пригоден engine, и Unreal е точно такъв. Въпреки, че е по-сложен от неговите алтернативи, той е много подходящ за изработването на 3D игри. Също така, той притежава и един

много силен инструмент – Blueprints. Това е алтернатива на C++ и заменя писането на код със свързване на различни event-и и логически елементи, променливи и обекти с помощта на блокове. Силата им идва от това, че могат да се използват заедно със C++ и да улеснят част от работата. В дипломната работа са използвани за да се направи връзката между 3D изобразените обекти и сорс кода, който седи отдолу. Използвани са също така за графичен потребителски интерфейс (GUI) – всички менюта и полета със информация за дадена единица или играч. Един от недостатъците на UE4 е, че някои неща са зле документирани, но понеже engine-ът е много популярен има много дискусии и информация по форуми и различни уебсайтове.

2.2.2 C++ или Blueprints?

В дипломната работа се използват и Blueprints и C++. Въпреки, че може всичко да се направи само със C++ или само със Blueprints, работата с комбинация от двете улеснява процеса на разработване.

В Cisk има няколко основни класа, написани на C++, които съдържат методите и атрибутите, които биха били нужни за реализирането на играта. Върху тези C++ класове се надгражда с Blueprints. Всички неща свързани с 3D изобразяването и управлението на единиците и GUI се управляват чрез Blueprints, които работят със C++ класове и използват C++ методи. Пример: Местенето на единица – Визуалното преместване на единицата и отразяването на това действие в GUI се управлява от Blueprints, а променянето на стойностите на променливите и указателите на всички засегнати от действието обекти се случва посредством C++.

2.3 Алгоритъм

При зареждане на играта се отваря нивото MainMenu и се зарежда widget с бутон за започване на игра. Когато бъде натиснат се зарежда нивото GameMap, което първоначално съдържа по една единица и две сгради за всеки играч, всички

плочки върху които ще се развива играта, по един обект който съхранява данните за всеки от играчите, както и един който съхранява информация за ходовете на играчите. Зарежда се и widget с бутон, който стартира хода на първия играч, когато бъде натиснат. Widget-а се скрива и се показва този за ресурсите на играча, както и за приключване на хода.

При натискане на поле (tile) с ляв бутон на мишката се проверява дали на полето има разположена единица или сграда. Ако има се визуализира съответния widget с информация за единица/сграда. Ако няма, те се скриват за потребителя. Допълнително се проверява дали единицата разположена на полето е от клас Builder. Ако е се визуализира и менюто за строене на сгради. При построяване на сграда тя се добавя в колекцията на съответния играч.

Ако се натисне върху поле с десен бутон се взима последната избрана единица и се проверява дали е на играча, който е на ход. Ако е, извършва се проверка дали на полето на което ще местим има единица. Ако има, се проверява дали тя е приятелска или вражеска. В случай, че е вражеска, се изпълнява функцията за атака. Ако вражеската единица бъде унищожена отново се извиква функцията за местене. Втори вариант е единицата да е приятелска. В такъв случай се проверява, дали тя е от същия тип и ако е, двете единици се смесват. Накрая се извършва проверка за да се види дали има сграда на полето, и ако тя е вражеска се превзема или унищожава (зависи от самата сграда и какво и е зададено във функцията).

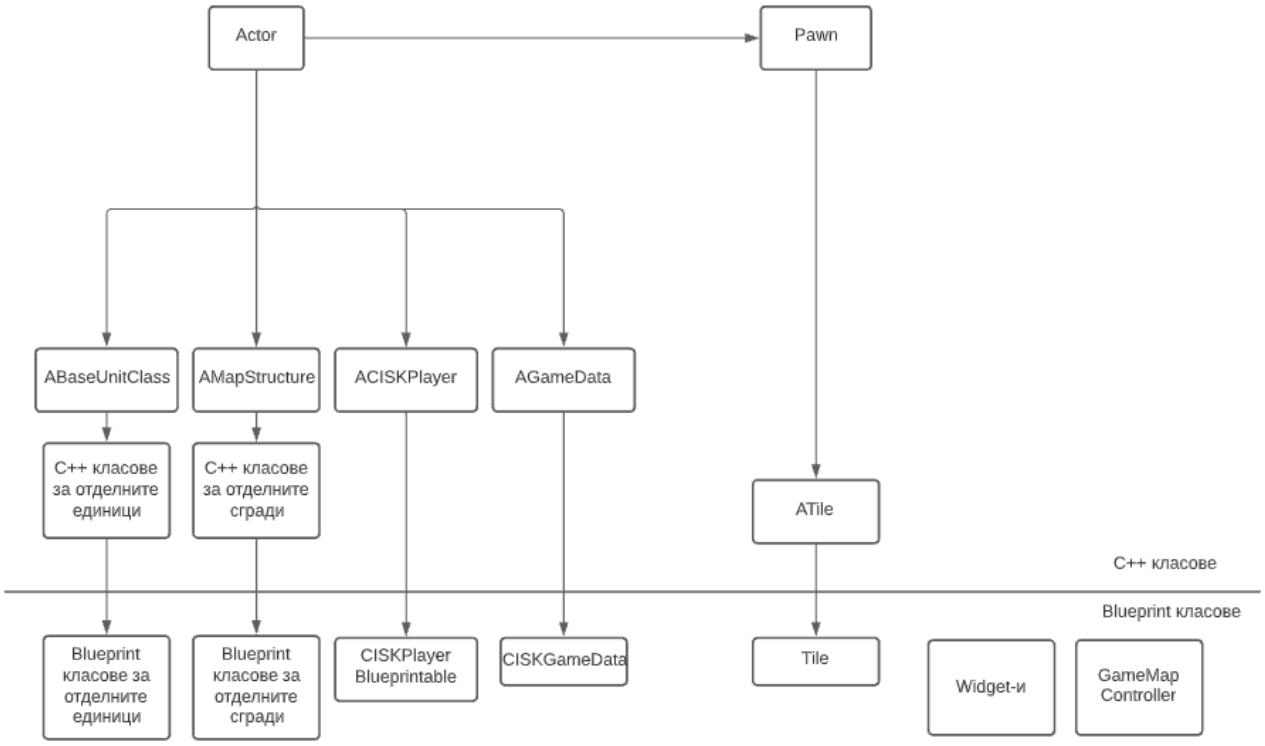
При натискане на бутона за приключване на ход се проверява дали някой от играчите не е загубил всички свои сгради и единици, и ако е, играта приключва. Ако не е, отново се визуализира widget-а за начало на ход, като този път е ред на вторият играч.

Когато и той приключи своя ход се минава през всички сгради и се изпълнява тяхната функция NewTurnRoutine. Ако сградата произвежда единици се

проверява дали нейното поле е заето от друга единица. Минава се и през всички единици за да им се промени атрибута, който показва дали са били местени в текущия ход.

2.4. Архитектура на класовете

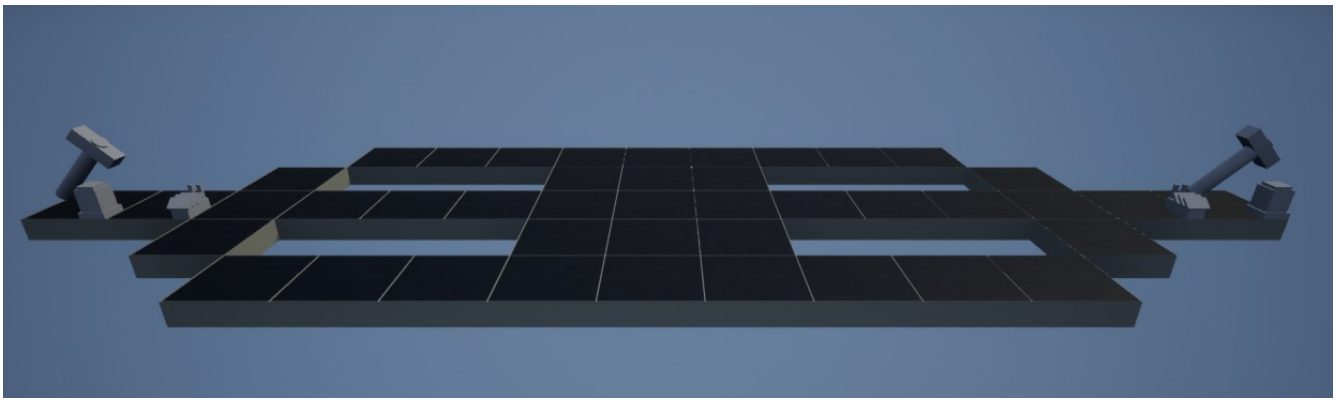
Архитектурата на класовете показва какви класове се използват, йерархията на класовете (inheritance tree), както и дали управлението им се извършва посредством C++ код или Blueprints. Това е опростена схема на йерархията на класовете. A => B значи, че B наследява A.



Фиг. 2.1 – Опростена йерархия на класовете в Cisk

2.5 Как протича една игра?

Играчите започват играта в двата края на картата с фабрика (която дава ресурси в началото на всеки ход), сграда, която създава строители (Builder) и 1 строител (който се използва за изграждане на допълнителни сгради). Двамата играчи се редуват. В своите ходове те местят своите единици, строят нови сгради и се борят за надмощие на картата. Освен за офанзива, играчите трябва да следят и за защитата на своите сгради, защото една незабелязана вражеска единица може да превземе ред сгради и да обърне играта. Когато някой играч получи надмощие, то може да му бъде отнето чрез защитни сгради (укрепления), които дават огромни защитни бонуси, или със стратегическо използване на определен тип единица. Когато един от играчите изгуби по-голямата част от единиците си, той започва да губи и сгради. Когато загуби всички свои единици и сгради, той губи играта.



Фиг. 2.2 Карта на играта

ТРЕТА ГЛАВА

3. Програмна реализация на CISK

3.1. Основни класове

В имплементацията на играта има няколко класа, в които се съдържат всички атрибути и методи. Те дават добра основа за надграждане, което прави изключително лесно създаването на нови типове сгради и единици. Единствено трябва да се направи 3D модел и да се въведат стойности за техните атрибути.

3.1.1. ABaseUnitClass

ABaseUnitClass е основният ми клас за единици. Всички останали единици го наследяват.

Атрибути на класа

```
UPROPERTY(EditAnywhere, Category = "Unit");
class ACISKPlayer* Player;
int ClassId;
int CountersClassId;
bool CanRetaliate;
uint16_t Damage;
uint16_t Defence;
int MaxHealth;
int CurrentHealth;
int Quantity;
FString ClassName;
UPROPERTY(EditAnywhere, Category = "Unit");
class ATile* CurrentLocation;
bool MovedThisTurn;
```

Фиг. 3.1 Атрибути на класа ABaseUnitClass

- Player – типът ACISKPlayer наследява Actor и съдържа основна информация за играчът, който притежава единицата.
- ClassId – Съдържа ID-то на типа на единицата. Всеки тип има различно ID, като това се използва за калкулация на допълнителни щети.
- CountersClassId – Използва се за да покаже срещу кой тип единица има бонуси за щети.
- CanRetaliate – Показва дали единицата може да отвърне на удар, когато бъде атакувана.
- Damage - Базовите щети на единицата. Използва се във формулата за смятане на щетите, които ще бъдат нанесени при атака.
- Defence – Това е защитата на единицата. Колкото повече, толкова по-малко щети поема, когато бъде атакувана.
- MaxHealth – Показва колко е максималния живот / HP / на единицата.
- CurrnetHealth – Текущият живот на единицата. Когато падне под 0 се намалява броя (Quantity) на единицата.
- Quantity – Това е броят на единиците в една клетка / Tile /. Използва се във формулата за изчисляване на щетите. Ако падне под 0 единицата се премахва.
- ClassName – Използва се за да се изпише типа на единицата в GUI
- CurrentLocation – Типът ATile наследява Pawn и съдържа информация за полето, в което единицата се намира, като бонуси от сгради/терен.

Методи на класа

CalculateDamage

Аргументи:

ABaseUnitClass* Attacker, ABaseUnitClass* Defender

Изчислява щетите, които ще бъдат нанесени при атака. Взима предвид атрибутите на двете единици, както и на сградите на съответните полета. Връща цяло число (int). Формулата за изчисляване на щетите се вижда в тялото на функцията.

Връща: int (цяло число), който показва колко щети трябва да бъдат нанесени

```
//Calculates the damage for the AttackEnemy function
int CalculateDamage(ABaseUnitClass* Attacker, ABaseUnitClass* Defender) {

    int DamageToDeal = 1;

    if (Attacker->Damage > Defender->Defence) {
        DamageToDeal += (Attacker->Damage + this->CurrentLocation->GetBonusAtt()) - (Defender->Defence + Defender->CurrentLocation->GetBonusDef());
        if (DamageToDeal <= 0) {
            DamageToDeal = 1;
        }
    }

    if (Defender->ClassId == Attacker->CountersClassId) {
        DamageToDeal = DamageToDeal * 2;
    }

    if (Defender->Quantity / 10 == 0) {
        DamageToDeal = DamageToDeal * Attacker->Quantity;
    }
    else {
        DamageToDeal = DamageToDeal * Attacker->Quantity / (Defender->Quantity / 10);
    }
    return DamageToDeal;
}
```

Фиг. 3.2 Тялото на CalculateDamage метода

AttackEnemy

Аргументи:

ABaseUnitClass* Enemy.

На функцията се подава единица, която да бъде нападната. Проверява се дали тя може да отвръща на удар. Ако броят на някоя от единиците падне под 1, тя бива унищожена. Като резултат връща int, който показва дали някоя от единиците е била унищожена. Ако и двете единици бъдат унищожени, връща -1. Ако само противника бъде унищожен, връща 1. Ако само нападателя бъде унищожен, връща -2, а ако и двете единици оцелеят след сблъсъка, връща 0.

```
int AttackEnemy(ABaseUnitClass* Enemy) {

    int OutputCode = 0;

    int DamageToDeal = CalculateDamage(this, Enemy);
    if (Enemy->CanRetaliate == true) {
        int DamageToTake = CalculateDamage(Enemy, this);
        int UnusedDamageToTake = DamageToTake % this->MaxHealth;
        DamageToTake -= UnusedDamageToTake;
        this->CurrentHealth -= UnusedDamageToTake;
        if (this->CurrentHealth <= 0) {
            this->Quantity -= 1;
            this->CurrentHealth = this->MaxHealth + this->CurrentHealth;
        }
        this->Quantity -= DamageToTake / this->MaxHealth;

    }
```

Фиг. 3.3 Част от AttackEnemy

```
int UnusedDamage = DamageToDeal % Enemy->MaxHealth;
DamageToDeal -= UnusedDamage;
Enemy->CurrentHealth -= UnusedDamage;
if (Enemy->CurrentHealth <= 0) {
    Enemy->Quantity -= 1;
    Enemy->CurrentHealth = Enemy->MaxHealth + Enemy->CurrentHealth;
}
Enemy->Quantity -= DamageToDeal / Enemy->MaxHealth;
```

Фиг. 3.4 Част от AttackEnemy

```
if (Enemy->Quantity <= 0) {
    OutputCode++;
    Enemy->CurrentLocation->Unit = nullptr;
    Enemy->CurrentLocation->HasUnit = false;
    Enemy->GetPlayer()->Units.Remove(Enemy);
    Enemy->Destroy();
}
if (this->Quantity <= 0) {
    this->CurrentLocation->Unit = nullptr;
    this->CurrentLocation->HasUnit = false;
    this->GetPlayer()->Units.Remove(this);
    this->Destroy();
    OutputCode -= 2;
}
return OutputCode;
```

Фиг. 3.5 Част от AttackEnemy

Move(ATile* NewTile)

Извиква се през Blueprints и се използва за да се променят указателите на единицата/клетките, които са засегнати. Прави проверки дали преместването е валидно и променя атрибутите на засегнатите обекти, ако е валидно.

```
UFUNCTION(BlueprintCallable, Category = "Unit")
    int Move(ATile* NewTile) {
    if (this->MovedThisTurn == false) {
        for (int i = 0; i < this->CurrentLocation->NeighboursCount; i++) {
            if (this->CurrentLocation->Neighbours[i] == NewTile) {
                this->MovedThisTurn = true;
                if (NewTile->HasUnit == false) {
                    this->CurrentLocation->HasUnit = false;
                    this->CurrentLocation->Unit = nullptr;
                    this->CurrentLocation = NewTile;
                    NewTile->Unit = this;
                    NewTile->PlayerId = this->Player->GetID();
                    NewTile->HasUnit = true;
                    if (NewTile->Structure != nullptr) {
                        if(NewTile->Structure->GetStructureOwner() != this->GetPlayer())
                        NewTile->Structure->Capture(this->GetPlayer());
                    }
                    return 0;
                }
                else {
                    if (this->GetPlayer() != NewTile->Unit->GetPlayer()) {
```

Фиг. 3.6 Част от алгоритъма за местене на единица

На фиг. 3.6 е изобразена първата част от алгоритъма. Проверява се дали единицата е местена този ход, ако не е, се проверява дали клетката, в която искаме да преместим единицата, е съседна на текущата. След това, се прави проверка, дали клетката е празна. Ако е празна се извършва серия от операции с указатели, за да се сложат правилните стойности на атрибутите на всички засегнати обекти.

В случай, че има единица на даденото поле, се проверява дали е вражеска или приятелска. Ако единицата е вражеска се извиква метода AttackEnemy. Ако тя бъде унищожена, отново се изпълнява серията от операции показана на фиг. 3.6

В случай, че единицата е приятелска, проверяваме дали двете са от един и същ тип, и ако са, двете се смесват, като им се събира тяхното количество (Quantity)

```
                }
                else {
                    if (this->GetPlayer() != NewTile->Unit->GetPlayer()) {
                        int CanContinue = this->AttackEnemy(NewTile->Unit);
                        if (CanContinue == 1) {
                            this->CurrentLocation->HasUnit = false;
                            this->CurrentLocation->Unit = nullptr;
                            this->CurrentLocation = NewTile;
                            NewTile->Unit = this;
                            NewTile->PlayerId = this->Player->GetID();
                            NewTile->HasUnit = true;
                            if (NewTile->Structure != nullptr) {
                                NewTile->Structure->SetStructureOwner(this->GetPlayer());
                            }
                            return 0;
                        }
                    }
                    else if (this->ClassId == NewTile->Unit->ClassId) {
                        NewTile->Unit->Quantity += this->Quantity;
                        this->CurrentLocation->Unit = nullptr;
                        this->CurrentLocation->HasUnit = false;
                        this->GetPlayer()->Units.Remove(this);
                        this->Destroy();
                    }
                }
            }
        }
    }
    return -1;
}
```

Фиг. 3.7 Остатъка от алгоритъма за преместване на единица

3.1.2. Сгради

AMapStructure е клас, който използвам за сградите.

```
UPROPERTY(EditAnywhere, Category = "Building")
class ACISKPlayer* StructureOwner;
bool CanBeCaptured;
UPROPERTY(EditAnywhere, Category = "Building")
int DmgBonus;
UPROPERTY(EditAnywhere, Category = "Building")
int DefBonus;
```

Фиг. 3.8 Атрибути на класа AMapStructure

- ACISKPlayer* StructureOwner – показва кой играч е собственика на сградата.
- CanBeCaptured – показва дали сградата може да се превземе, или ще бъде унищожена при смяна на собственика.
- DmgBonus – Това са допълнителните щети, които дава сградата. Ако не дава никакъв бонус, това просто остава 0
- DefBonus – Същото като DmgBonus, но за защита.
- Пази се и указател към клетката, на която се намира сградата. Използва се за да се укаже на коя клетка да се появяват единиците (ако сградата произвежда)

Методи на класа:

NewTurnRoutine

Изпълнява след всяко извъртане на играчите (когато всички играчи изпълнят по 1 ход). Тази функция е virtual и затова няма тяло. Тя се override-ва във всеки клас, който я наследява. Използва се за да дава материали на играчите или да създава единици.

Capture

Функцията прехвърля собствеността на сградата или я унищожава. Това се случва, когато клетката, на която седи сградата, бъде превзета от друг играч.

```
void AMapStructure::Capture(ACISKPlayer* Attacker) {
    if (this->CanBeCaptured == true) {
        this->GetStructureOwner()->Structures.Remove(this);
        this->SetStructureOwner(Attacker);
        Attacker->Structures.Add(this);
    }
    else {
        this->GetStructureOwner()->Structures.Remove(this);
        this->Tile->Structure = nullptr;
        this->Destroy();
    }
};
```

Фиг. 3.9 Тялото на Capture метода

3.1.3. AGameData

Атрибути на класа:

TurnCount – Пази общо колко хода са изиграли всички играчи. Използва се за да се определи кой играч е на ход, както и колко пъти са се извъртяли всички играчи.

Методи:

GetTurn

GetTurn връща колко пъти са се минали всички играчи. За момента поддържа само двама играчи, но с малка промяна може да се направи за N-брой

```
UFUNCTION(BlueprintCallable, Category = "Turns")
int GetTurn() {
    if (this->TurnCount % 2 == 0) {
        return this->TurnCount / 2;
    }
    else {
        return (this->TurnCount + 1) / 2;
    }
}
```

Фиг. 3.10 GetTurn

GetActivePlayer

Връща кой играч е на ход в момента на извикване. За момента поддържа само двама играчи, но с малка промяна може да се направи за N-брой.

```
UFUNCTION(BlueprintCallable, Category = "Turns")
    int GetActivePlayer() {
    if (this->TurnCount % 2 == 0) {
        return 2;
    }
    else {
        return 1;
    }
}
```

Фиг. 3.11 GetActivePlayer

IncrementTurn

Инкрементира TurnCount когато някой играч приключи хода си.

3.1.4. ATile

Това е клас, който използвам за полетата / tile / в играта. Той наследява Pawn, и през него се осъществява връзката между различни единици и сгради.

```
UPROPERTY(EditAnywhere, Category = "Tile Info")
ATile* Neighbours[8];
UPROPERTY(EditAnywhere, Category = "Tile Info")
class AMapStructure* Structure;
UPROPERTY(EditAnywhere, Category = "Tile Info")
class ABaseUnitClass* Unit;
UPROPERTY(EditAnywhere, Category = "Tile Info")
bool HasUnit;
UPROPERTY(EditAnywhere, Category = "Tile Info")
int NeighboursCount;
```

Фиг. 3.12 Атрибути на класа

- Neighbours – Съдържа указатели към съседните 8 полета. Това се използва, за да може да се следи къде дадена единица може да ходи/атакува.
- Structure – Сочи към сградата, която се намира на съответното поле. През този pointer се взима информация за бонус защита/щети, които предоставя сградата.
- Unit – Съдържа единицата, която се намира на даденото поле. Използва се за атакуване, създаване и увеличаване на броя на единици.
- HasUnit показва дали има единица на даденото поле. С тази променлива се избягва пипането на null pointer-и и случайни адреси в паметта.
- Има и допълнителна променлива int NeighboursCount, която се използва за по-бързо обхождане съседните клетки (проверка при движението на единица)

Методи:

Getter/Setter за Unit/Structure.

Getter-и за допълнителните атака/защита от сградата, която се намира на полето, както и за Id на играча, който „притежава“ клетката.

3.1.5. ACISKPlayer

Този клас съхранява информация за материалите (ресурси) на играч, както и неговите сгради и единици. Ако някой играч изгуби всички свои сгради и единици, той губи играта.

```
UPROPERTY(EditAnywhere, Category = "Player Stats")
    int Materials;

UPROPERTY(EditAnywhere, Category = "Player Stats")
    int id;

UPROPERTY(EditAnywhere, Category = "Player Stats")
    TArray<class ABaseUnitClass*> Units;

UPROPERTY(EditAnywhere, Category = "Player Stats")
    TArray<class AMapStructure*> Structures;
```

Фиг. 3.13 Атрибути на класа

- Materials – Пази броя на материалите на играча. Те се използват за строене на сгради. Всички играчи започват с по 200 материала.
- id – Използва се за да се определи дали даден играч е на ход в момента и дали неговите единици могат да се местят или да се строят негови сгради.
- Units – Това е списък със всички единици на играча. При приключване на ход се проверява дали е празен.
- Structures – Пази всички сгради на играча. И тук се извърша проверка в края на всеки ход. Ако и двата TArray-а са празни, играта приключва.

Методи:

Getter-и за id и Materials

AddMaterials/RemoveMaterials

AddStructure

3.2. Widget-и

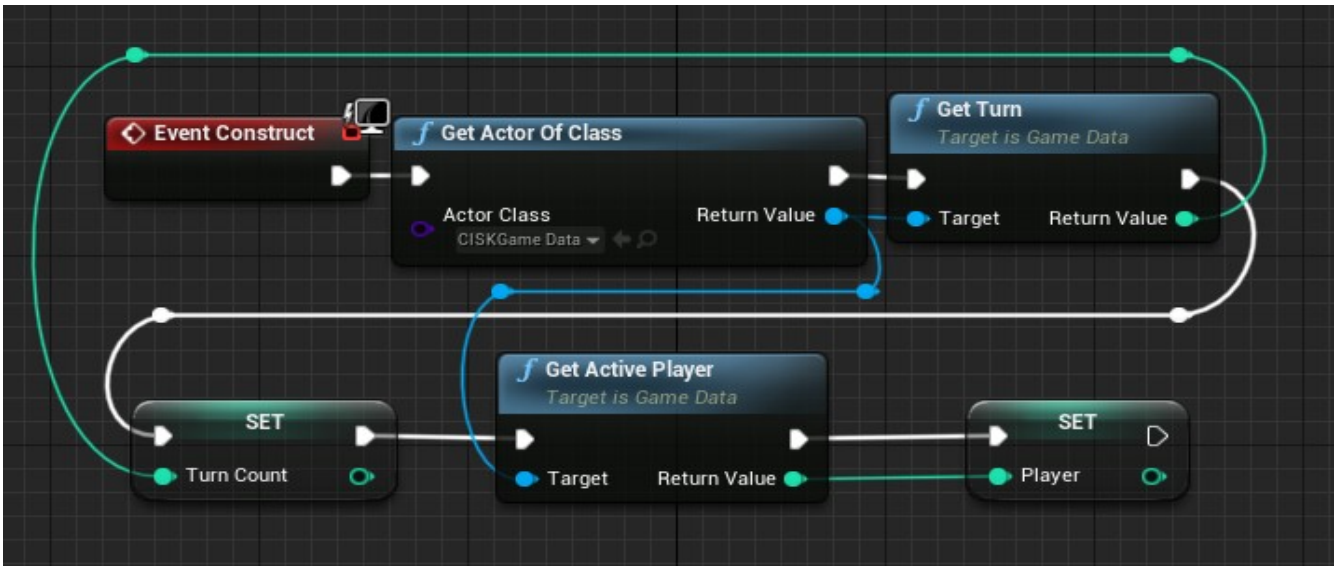
Widget-ите са вид 2D графичен потребителски интерфейс и включват текстове, бутони, изображения и др. В CISK се използват за изобразяване на данните на дадена единица, като например защитата. Използват се също така за менютата в играта. В CISK те се управляват изцяло чрез Blueprint, и използват информация от обекти на класовете описани в 3.1.

3.2.1. MainMenu

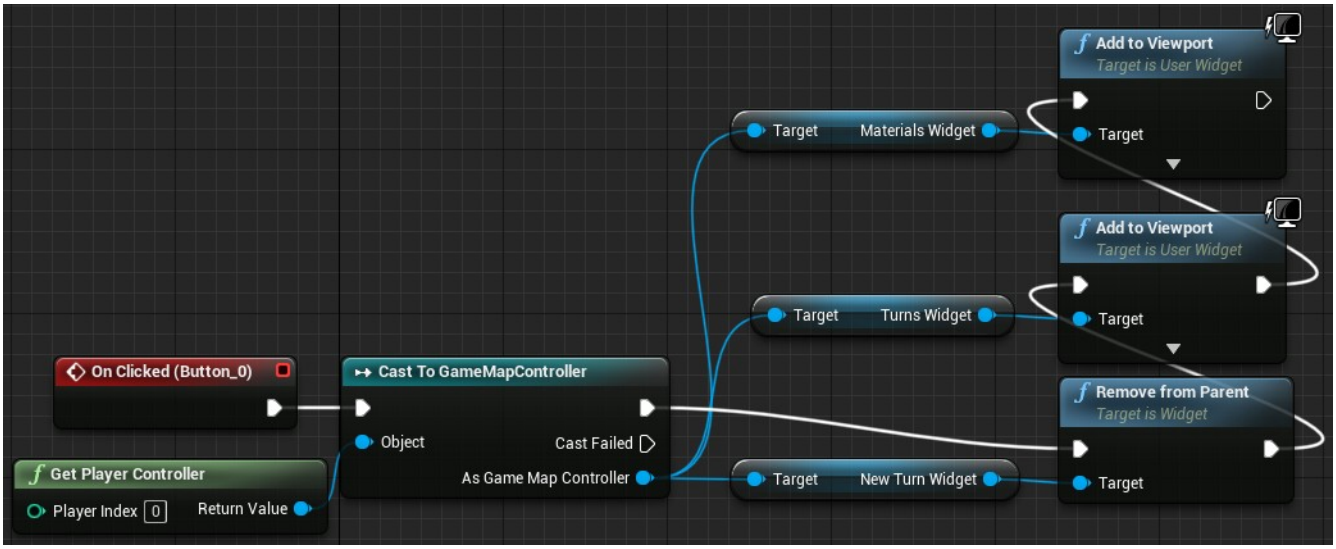
Зарежда се при отварянето на нивото MainMenuLevel. Вкючва изображение, текст и бутон за стартиране на играта. При натискане на бутона се зарежда нивото GameMap. Използвано е чуждо изображение за фон[8].

3.2.2. NewTurnWidget

Визуализира се веднага, след като се зареди GameMapLevel, както и когато някой играч започва хода си. Това е мярка за сигурност, че никой от играчите не е правил нещо от името на друг. Съдържа бутон за започване на хода, както и информация за играча, който е на ход. При натискане на бутона се визуализират останалите widget-и за това ниво (MaterialsWidget и TurnWidget). Информацията за хода и играча се взимат от обект от класа AGameData.



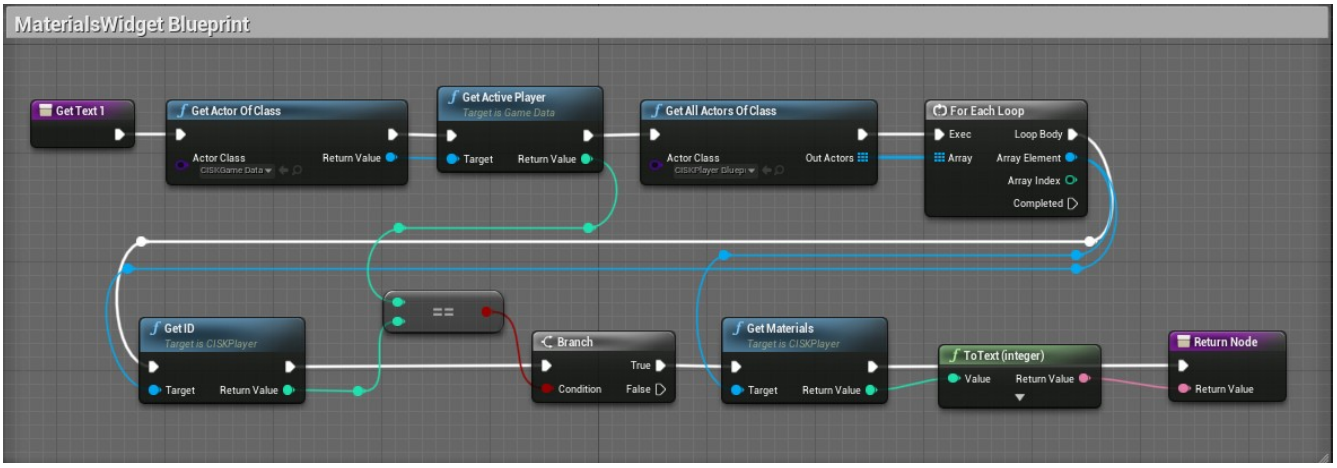
Фиг. 3.14 Изпълнява се след като widget-а се добави към viewport-а



Фиг. 3.15 Изпълнява се при натискането на бутона в NewTurnWidget

3.2.3. MaterialsWidget

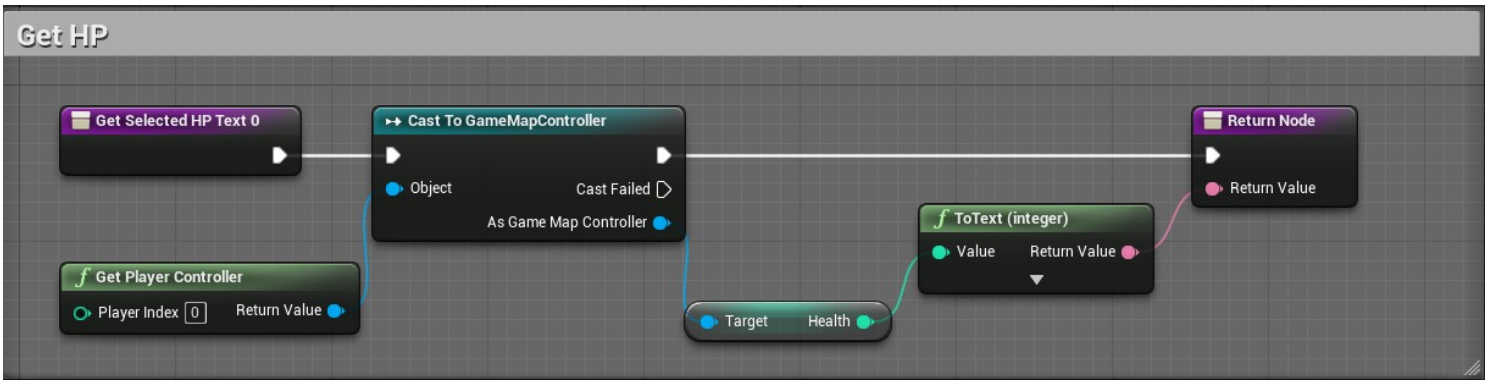
Показва броя материали на активния играч. Информацията се взима чрез Blueprint, като се намира играч с id, равно на числото, което се връща за активен играч.



Фиг. 3.16 Blueprints схема на MaterialsWidget

3.2.4. UnitInfoBarWidget

В този widget влиза цялата информация за избраната в момента единица (единицата, която стои на избраната от играча клетка). Информацията се взима от обект от Blueprint класа GameMapController. Информацията се променя когато играчът цъкне на друга клетка.

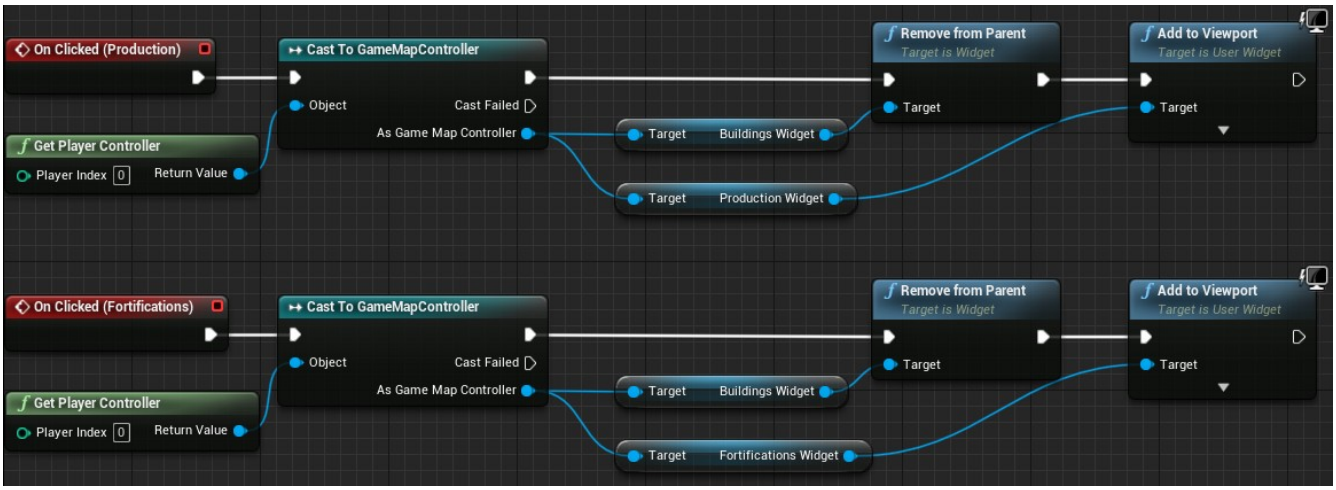


Фиг. 3.17 Blueprints схема за вземане на данни за GUI

Схемата е идентична и за остатъка от променливите, които са ни нужни.

3.2.5. BuildingsWidget

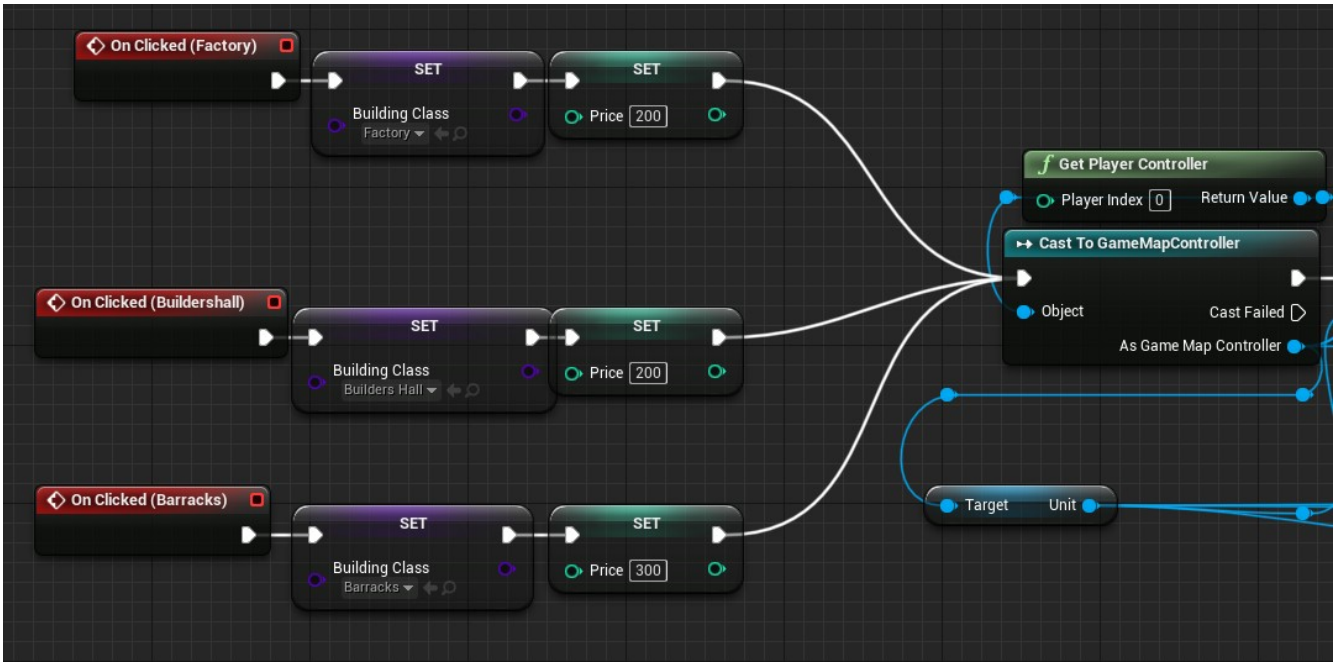
Този widget представлява меню за строене на сгради. То излиза в лявата част на екрана, когато избраната единица е от тип Builder. Има два бутона, всеки от които води към друг widget. Единият от тях е за укрепления (сгради, които нямат имплементиран NewTurnRoutine, а само дават пасивни бонуси за атака/защита. Другият widget показва всички производствени сгради (тези, които произвеждат материали или създават единици).



Фиг. 3.18 BuildingsWidget Blueprint схема

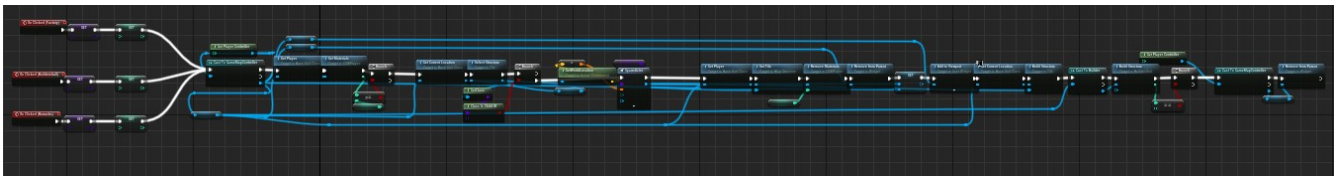
3.2.6 ProductionBuildingsWidget/FortificationsWidget

Двата widget-а са почти еднакви, като единственото различие е, че сградите, които могат да се построят, са разделени на 2 типа – произвеждащи и укрепления. В началото, чрез натискане на бутон, се избира каква сграда да бъде построена, като по този начин се задават цената и класа на сградата (Фиг. 3.19). Проверява се дали играчът има достатъчно материали, за да я построи, и ако има, се прави още една проверка, дали няма сграда на полето, на което искаме да строим. Ако няма се продължава напред



Фиг. 3.19 Избор на сграда

След проверките се извиква SpawnActor, като се дават координатите на полето, на което се намираме (3D моделите са създадени така, че единицата, полето и сградата да са на едни и същи координати и да не се припокриват), заедно със сградата, която е избрана на фиг. 3.19. Взема се референция към собственика на строителя (единицата, която строи сградата) и му се добавя новата сграда в списъка. На новата сграда и се задават полето, на което е, както и кой е нейният собственик. На клетката също и се указва, че вече притежава сграда. Намаля се броя на строителите с 1, и ако той падне под 1, единицата се унищожава.



Фиг. 3.20 Цялата схема

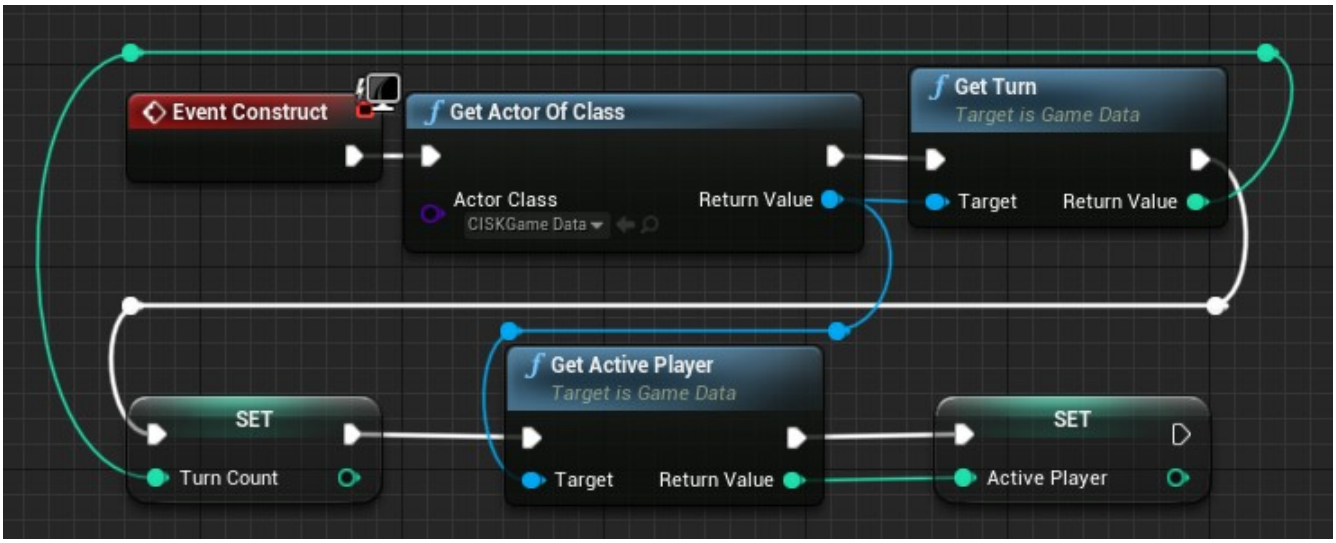
Има и един бутон за връщане назад (към BuildingsWidget). Неговата схема е проста, скрива се единият widget и на негово място се зарежда друг (като на фиг. 3.18)

3.2.7 BuildingsInfoWidget

Това е малък widget, който проверява дали на избраната клетка има сграда и ако има, връща Id-то на нейния собственик, както и името на сградата.

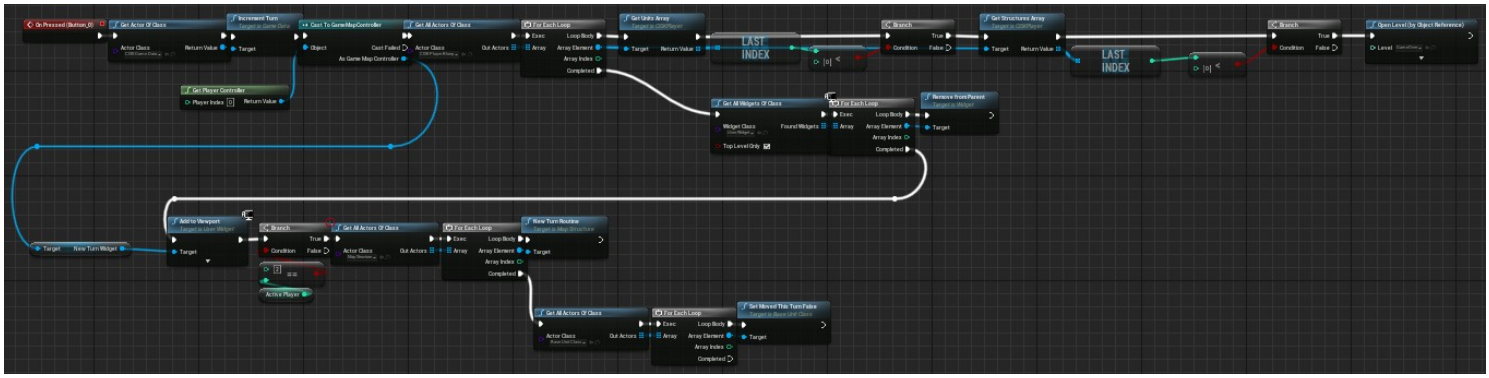
3.2.8 TurnWidget

TurnWidget се състои от 2 части – поле, което показва кой по ред ход е, и друго, в което се съдържа бутон за приключване на хода, както и Id-то на активния играч. Данните се вземат посредством схемата, показана на фиг. 3.21



Фиг. 3.21 Схема за вземане на данни

При натискане на бутона се увеличава броя на ходове (в CISKGameData). Ако някой от играчите е изгубил всички свои сгради и единици се отваря нивото GameOver. Ако и двамата играчи могат да продължат се прави проверка, дали вторият играч е приключил хода си, и ако е бил той, се минава през всички единици и им се казва, че могат да бъдат местени пак. Минава се и през всички сгради и им се извиква метода NewTurnRoutine. Скриват се и всички widget-и и към екрана се добавя NewTurnWidget.



Фиг. 3.22 Цялата схема

3.2.9 GameOverWidget

Визуализира се, когато се зареди нивото GameOver. Самият widget се състои от черен фон с надпис „Game Over”

3.3 Допълнителни класове

Това са всички класове, които наследяват някой от класовете, описани в 3.1. Това включва всички класове за единици, сгради, поле, всички Blueprint класове и др. Някои от тях имат допълнителни променливи и функционалности.

3.3.1 ABuilderUnit / Builder

Наследяват ABaseUnitClass. Това са съответно C++ и Blueprint класове, чрез които е изграден типа единица Builder (строител). Класът притежава нов метод: BuildStructure. Методът управлява част от логиката за строене на сграда, като добавя сградата в списъка на играча, собственик на строителя. Също така, намаля броя на единицата (Quantity) и проверява, дали не е паднал под 1. Ако е, единицата се унищожава. Кодът на функцията може да се види на фиг. 3.23. Функцията се извиква в widget-ите от 3.2.6

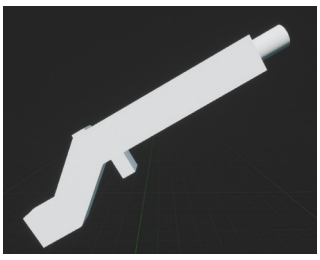
```
UFUNCTION(BlueprintCallable, Category = "Building")
    int BuildStructure(AMapStructure* NewStructure) {
        this->Player->AddStructure(NewStructure);
        this->Quantity -= 1;
        if (this->Quantity <= 0) {
            this->CurrentLocation->Unit = nullptr;
            this->CurrentLocation->HasUnit = false;
            this->GetPlayer()->Units.Remove(this);
            this->Destroy();
            return -1;
        }
        else {
            return 0;
        }
    }
```

Фиг. 3.23 Дефиниция на BuildStructure

В конструктора на BuilderUnit са зададени началните стойности на всичките му атрибути. Builder е Blueprint клас, който наследява BuilderUnit. При него е добавен и 3D модел, който се състои от цилиндър и паралелепипед, образуващи чук.

3.3.2 AInfantryUnit / Infantry

Наследяват AbaseUnitClass. Това е основната единица на всяка армия. InfantryUnit и Infantry отново са съответно C++ и Blueprints клас, като в C++ конструктора са зададени началните стойности на всички атрибути на класа. В Infantry е добавен 3D модел, който представлява огнестрелно оръжие.



Фиг. 3.24 3D модел на Infantry

3.3.3 AArmoredCar / Armored Car

Наследяват ABaseUnitClass. Това е първата единица, която използва бонуси срещу друг определен тип (Infantry). Стойностите са зададени в конструктора. Има и направен 3D модел.

3.3.4 AFactoryStructure / Factory

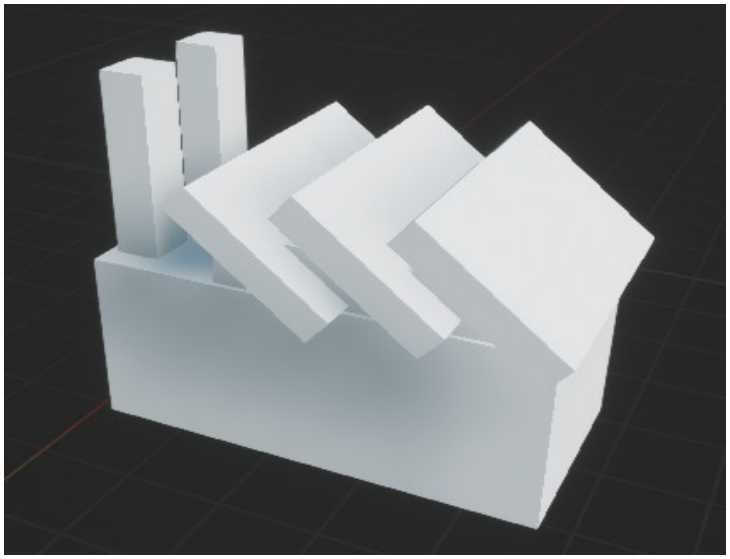
Наследяват AMapStructure. Във FactoryStructure са дефинирани телата на конструктора и на NewTurnRoutine (фиг. 3.25)

```
void AFactoryStructure::NewTurnRoutine(){
    this->StructureOwner->Materials += 50;
}

AFactoryStructure::AFactoryStructure() {
    PrimaryActorTick.bCanEverTick = true;
    this->CanBeCaptured = true;
    this->DmgBonus = 0;
    this->DefBonus = 0;
    this->ClassName = FString(TEXT("Factory"));
}
```

Фиг. 3.25 Дефиниции на конструктор и NewTurnRoutine

За Factory е създаден и прост 3D модел, показан на фиг. 3.26



Фиг. 3.26 3D модел на Factory

3.3.5 ABuildersHallStructure / BuildersHall

Наследяват AMapStructure. Това е сградата, която произвежда Builder-и в началото на всеки ход. Конструктора и NewTurnRoutine са дефинирани в BuildersHallStructure.cpp. Частта със SpawnActor е взета от [9] и е пригодена към моя алгоритъм. За да работи това е вкарана и допълнителна променлива в header файла (фиг 3.28). Стойността ѝ се задава чрез Blueprint class editor-а. Създаден е и 3D модел на класа. Проверява се и дали има единица от същия тип на полето. В случай, че има, NewTurnRoutine увеличава броя ѝ.

```cpp
ABuildersHallStructure::ABuildersHallStructure() {
    PrimaryActorTick.bCanEverTick = true;
    this->CanBeCaptured = true;
    this->DmgBonus = 0;
    this->DefBonus = 0;
    this->ClassName = FString(TEXT("Builder's Hall"));
}


void ABuildersHallStructure::NewTurnRoutine() {

    if (this->Tile->HasUnit == false) {
        const FVector Location = GetActorLocation();
        const FRotator Rotation = GetActorRotation();
        ABaseUnitClass* NewUnit = GetWorld()->SpawnActor<ABaseUnitClass>(UnitToSpawn, Location, Rotation);
        NewUnit->SetPlayer(this->GetStructureOwner());
        this->GetStructureOwner()->Units.Add(NewUnit);
        NewUnit->CurrentLocation = this->Tile;
        this->Tile->Unit = NewUnit;
        this->Tile->HasUnit = true;
    }
    else {
        if (this->Tile->Unit->ClassId == -1) {
            this->Tile->Unit->Quantity++;
        }
    }
}
```

Фиг. 3.27 Дефиниции на конструктор и NewTurnRoutine

```cpp
UPROPERTY(EditAnywhere, Category = "Unit");
TSubclassOf<ABaseUnitClass> UnitToSpawn;
```

Фиг. 3.28 Декларация на допълнителна променлива

3.3.6 ABarracksStructure / Barracks

Почти същото в сравнение с класовете от 3.3.5, но са променени стойностите на ClassName променливата, ClassId в NewTurnRoutine, както и на променливата UnitToSpawn. Вместо Builder-и, сградата произвежда единици от тип Infantry. Създаден е и 3D модел.

3.3.7 AvehicleFactoryStructure / Vehicle Factory

Като при 3.3.5 и 3.3.6, но произвежда Armored Car единици. Има 3D модел.

3.3.8 ABunkerStructure / AFortStructure / Bunker / Fort

Всички наследяват AMapStructure. Bunker и Fort са Blueprint класове, докато техните родители, BunkerStructure и FortStructure, са C++ класове. Не е имплементиран NewTurnRoutine за никой от класовете, понеже те са от тип укрепления и само дават пасивни бонуси. Създадени са и 3D модели.

```
AFortStructure::AFortStructure() {
    PrimaryActorTick.bCanEverTick = true;
    this->CanBeCaptured = false;
    this->DmgBonus = 4;
    this->DefBonus = 7;
    this->ClassName = FString(TEXT("Fort"));
}
```

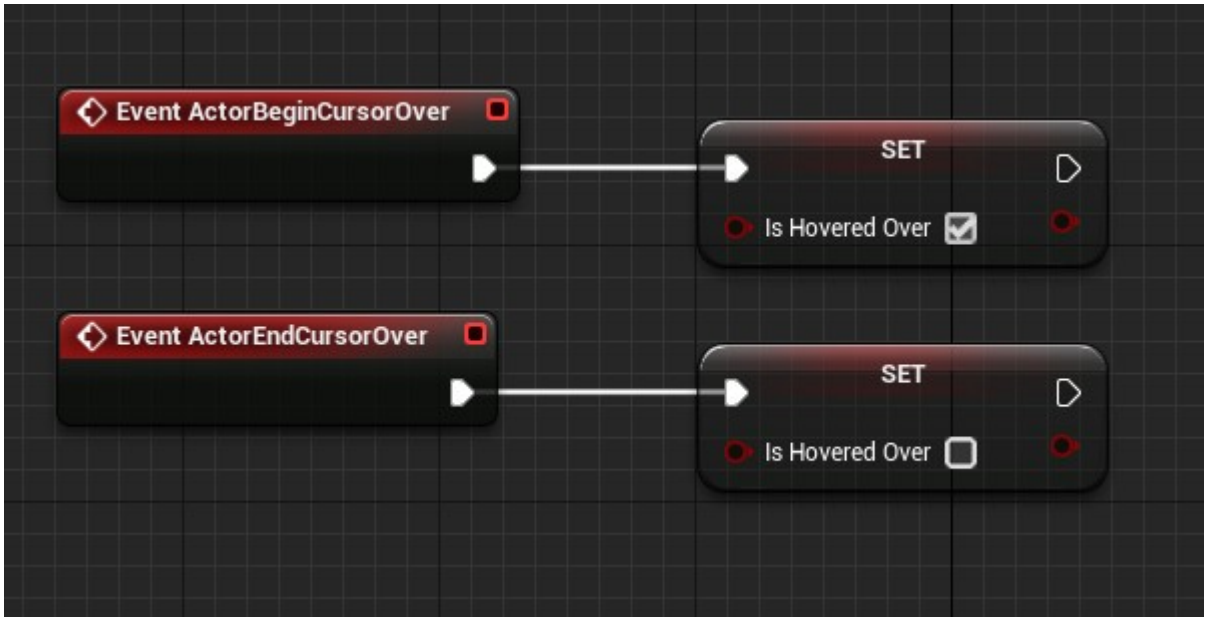
Фиг. 3.29 Конструктор на AFortStructure

```
ABunkerStructure::ABunkerStructure() {
    PrimaryActorTick.bCanEverTick = true;
    this->CanBeCaptured = true;
    this->DmgBonus = 2;
    this->DefBonus = 5;
    this->ClassName = FString(TEXT("Bunker"));
}
```

Фиг. 3.30 Конструктор на ABunkerStructure

3.3.9 Tile

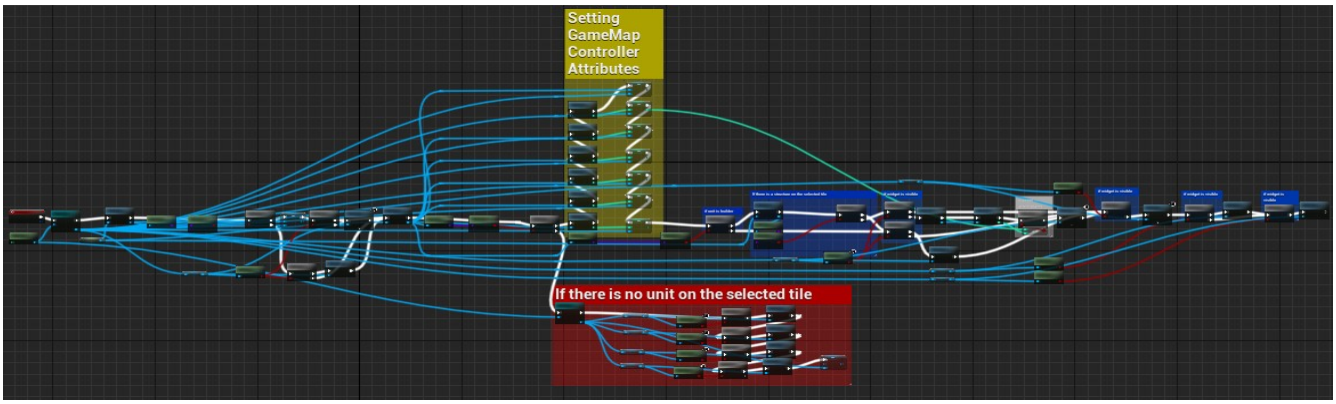
Наследява ATile. Това е един от най-големите Blueprint класове (с най-много промени спрямо C++ класа). Понеже този клас е Pawn, през него се управлява почти всичко (без движението на камерата и бутоните в GUI). Като за начало, класът съдържа булева променлива, която показва дали курсора на мишката е върху дадено поле(IsHoveredOver). Това се използва за местене на единици.



Фиг. 3.31 IsHoveredOver

Най-много действия се извършват, когато играч натисне с ляв бутон върху поле от картата. Първо се проверява дали има върху полето има сграда, за да може да се добави/премахне BuildingsInfoWidget. След това се проверява дали върху полето има единица. В случай, че има, атрибутите ѝ се запазват като променливи на GameMapController, заедно с указател към самата единица. Ако типът на единицата е Builder се проверява дали вече не е отворен BuildingsWidget (За да не се добавя към viewport-а няколко пъти. Затварят се ProductionBuildingsWidget и FortificationsWidget, защото те биха се припокрили с BuidingsWidget.

В случай, обаче, че е нямало единица на даденото поле се затварят всички widget-и, на които би им трябвала единица(UnitInfoBarWidget, BuildingsWidget и др.), като първо се проверява, дали са във viewport-а. Също така, указателят към единица, който се пази в GameMapController се изчиства (не сочи към никоя единица). Цялата схема е изобразена на фиг. 3.32.



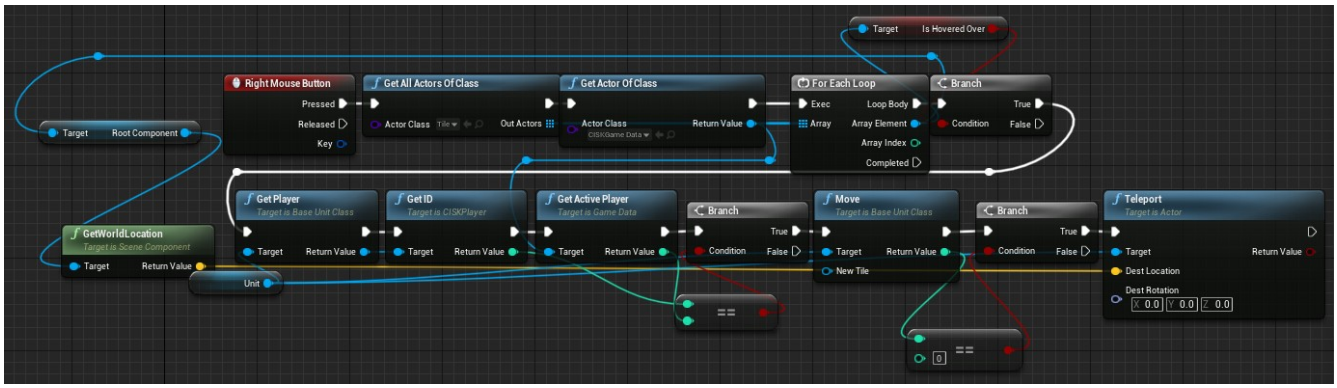
Фиг. 3.32 Логиката, която се изпълнява, когато се цъкне с ляв бутон на Tile

3.3.10 GameMapController

GameMapController наследява PlayerController. PlayerController-а е връзката между потребителя и pawn-а, който той управлява. В GameMapController се съхраняват променливите, които се използват в няколко различни класове, защото най-лесно се достъпва (Чрез GetPlayerController Node в Blueprints). Това са указатели към избраните единица и сграда, както и атрибутите на избраната единица. В него се съхраняват и указатели към всички widget-и, за да могат по-лесно да се добавят/махат от viewport-а. На фиг. 3.33 са показани всички променливи, които се пазят в GameMapController.

Фиг. 3.33 Всички променливи, които се съхраняват в GameMapController

В GameMapController се изпълнява и местенето на единица. Когато се засече натискане с десен бутон се минава през всички полета и се проверява стойността на тяхната IsHoveredOver променлива (за да се разбере върху кое поле е натиснат десният бутон). Ако се открие такава клетка се проверява и дали избраната единица принадлежи на играча, който е на ход. Ако е на активния играч се извиква Move методът на избраната единица. Ако той върне 0 (което значи, че единицата е преместена в „backend“-а), координатите на единицата се заменят с координатите на полето, върху което тя се мести.



Фиг. 3.34 Логика, която се изпълнява при натискане на десен бутон

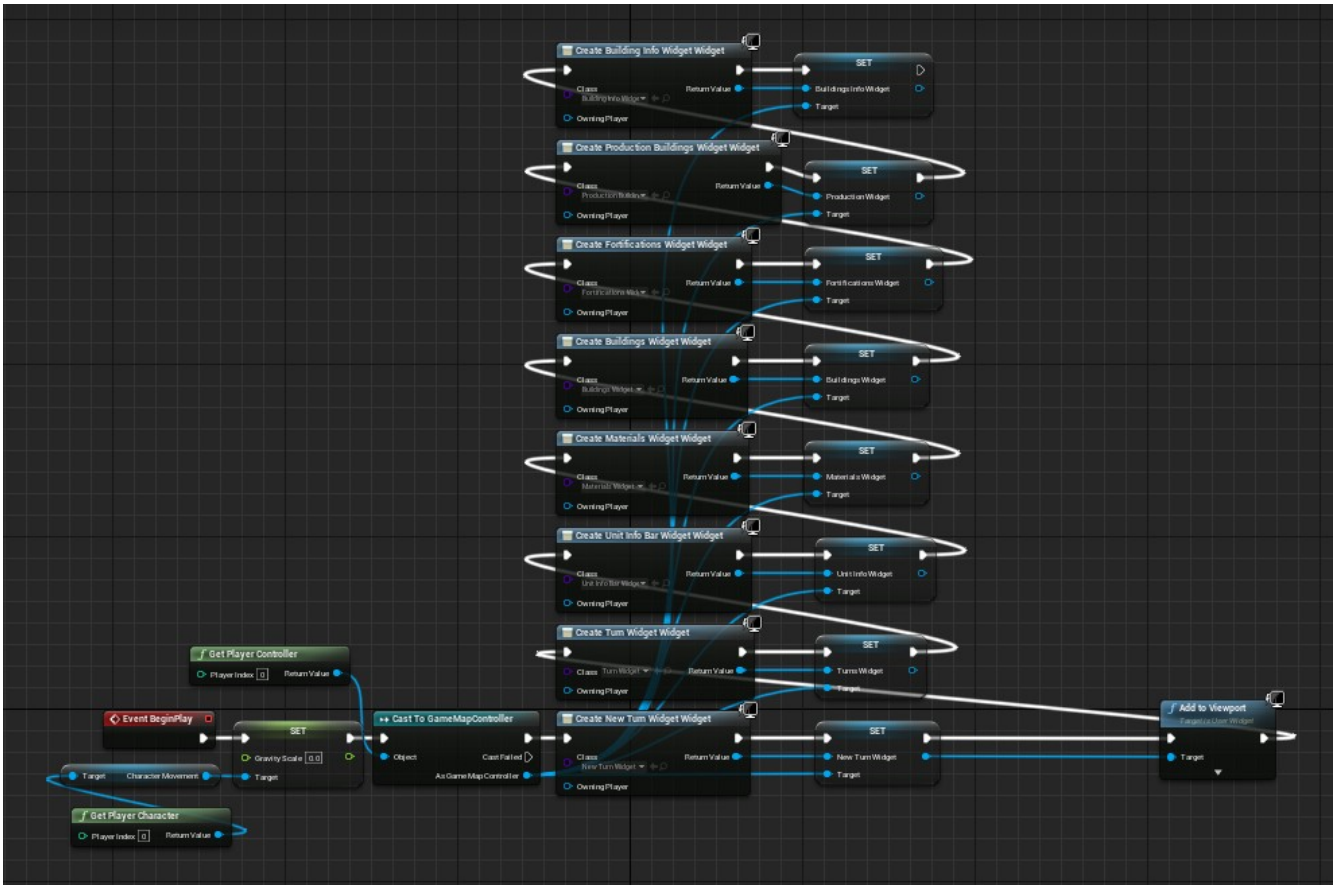
На края се затварят всички widget-и свързани със строене на сгради.

3.3.11 MainCharacter

Използва се за движение на камерата. Логиката за преместване на камерата е взета от [10] и пригодена към моята игра – ускорението е увеличено и е добавено играчът да спира на място веднага щом се пусне копчето за движение

3.4 GameMap Level

При зареждане на нивото GameMap се спира гравитацията на MainCharacter (за да може да си лети във въздуха без да пада). Също така се инициализират и всички Widget-и и се подават указатели към тях на GameMapController.



Фиг. 3.35 GameMap Level Blueprint

ЧЕТВЪРТА ГЛАВА

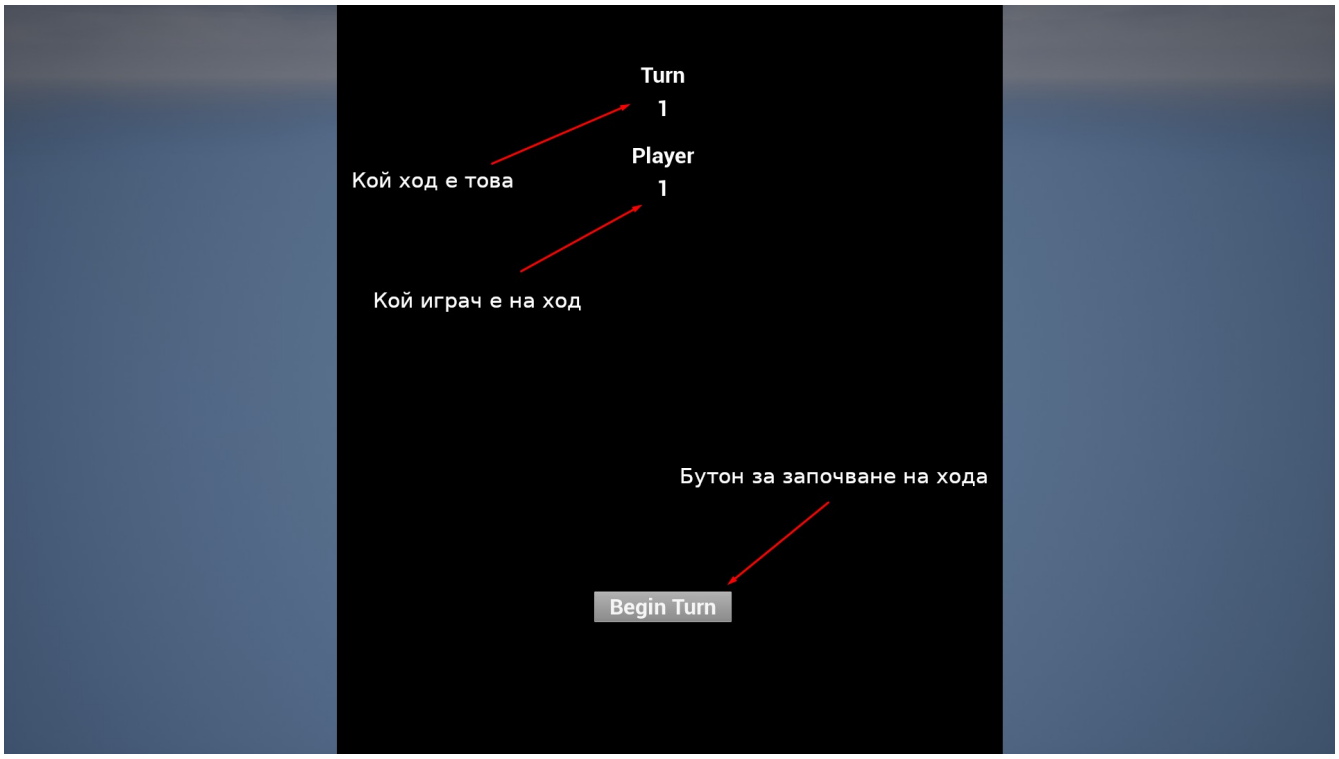
4. РЪКОВОДСТВО ЗА ПОТРЕБИТЕЛЯ

На началният екран има бутон, който стартира играта. Когато и двамата играчи са готови, бутонът може да се натисне (Бутонът е показан със стрелка на фиг. 4.1).

4.1 Потребителски интерфейс и user input

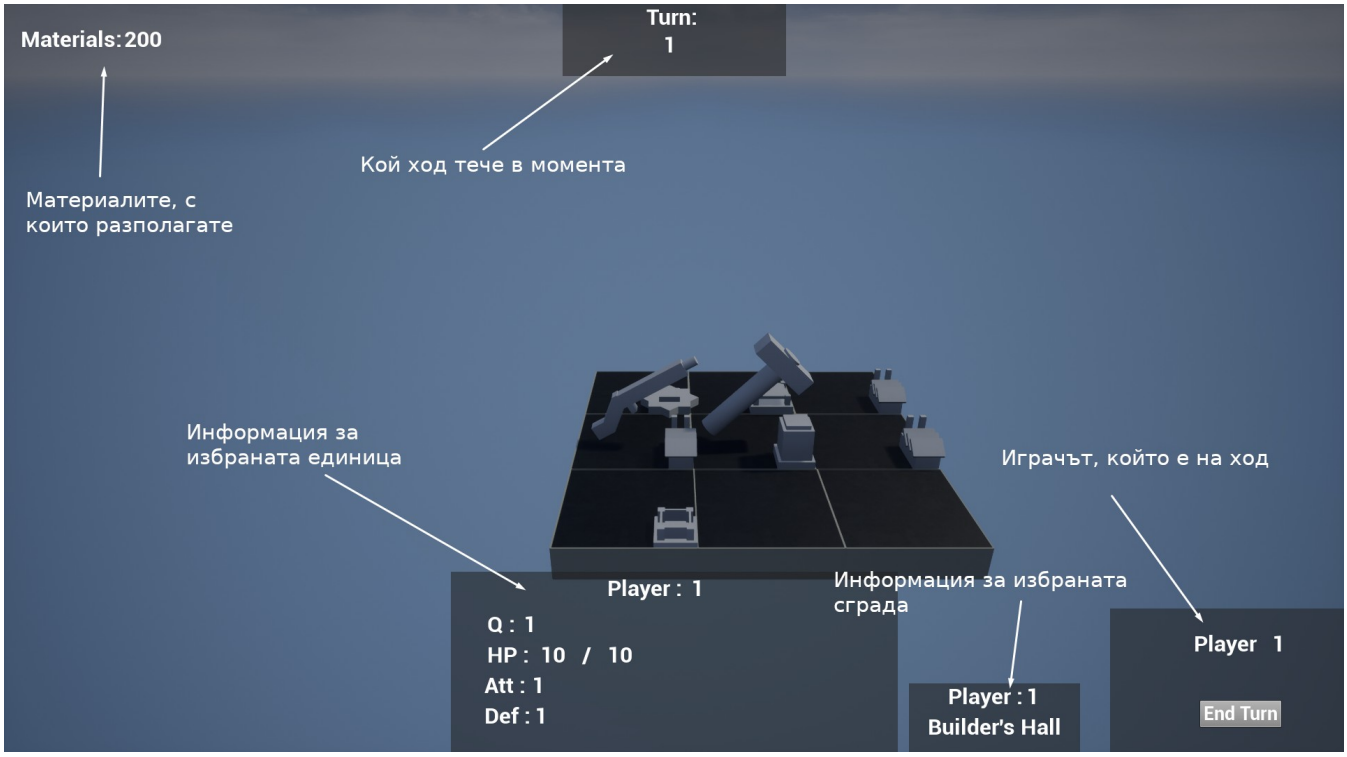
Фиг. 4.1 Начален екран

След като натиснете бутона, ще се озовете на следващия екран. Там получавате информация кой играч е на ход и колко хода е изиграл съответният играч. За да започнете своя ход трябва да натиснете бутона. Това е с цел избягване на измами. Ако екранът с бутона седи (фиг. 4.2), значи другият играч не е правил нищо от ваше име (местене на единици / строене на сгради).



Фиг. 4.2 Екран за започване на ход

След като започнете своя ход, ще видите картата на играта. Може да се местите с натискане на клавишите W,A,S,D. На картата има единици, сгради и полета. За да изберете някоя единица или сграда, цъкнете с ляв бутон на полето, върху която е разположена тя. За да преместите избраната от вас единица, цъкнете с десен бутон върху съседно поле. Ако на полето има вражеска единица, те ще се сбият. Ако на полето има приятелска единица от същия тип, двете армии ще се обединят. Освен единици, сгради и полета, най-вероятно ще видите и още нещо. В горния ляв ъгъл е показан броят на вашите материали. Те се използват за строене на сгради. Можете да добиете повече материали чрез фабрики. В долната част на екрана може да видите 2 полета с информация – едното показва информация за избраната от вас единица, а другото поле показва информация за избраната сграда (фиг 4.3)

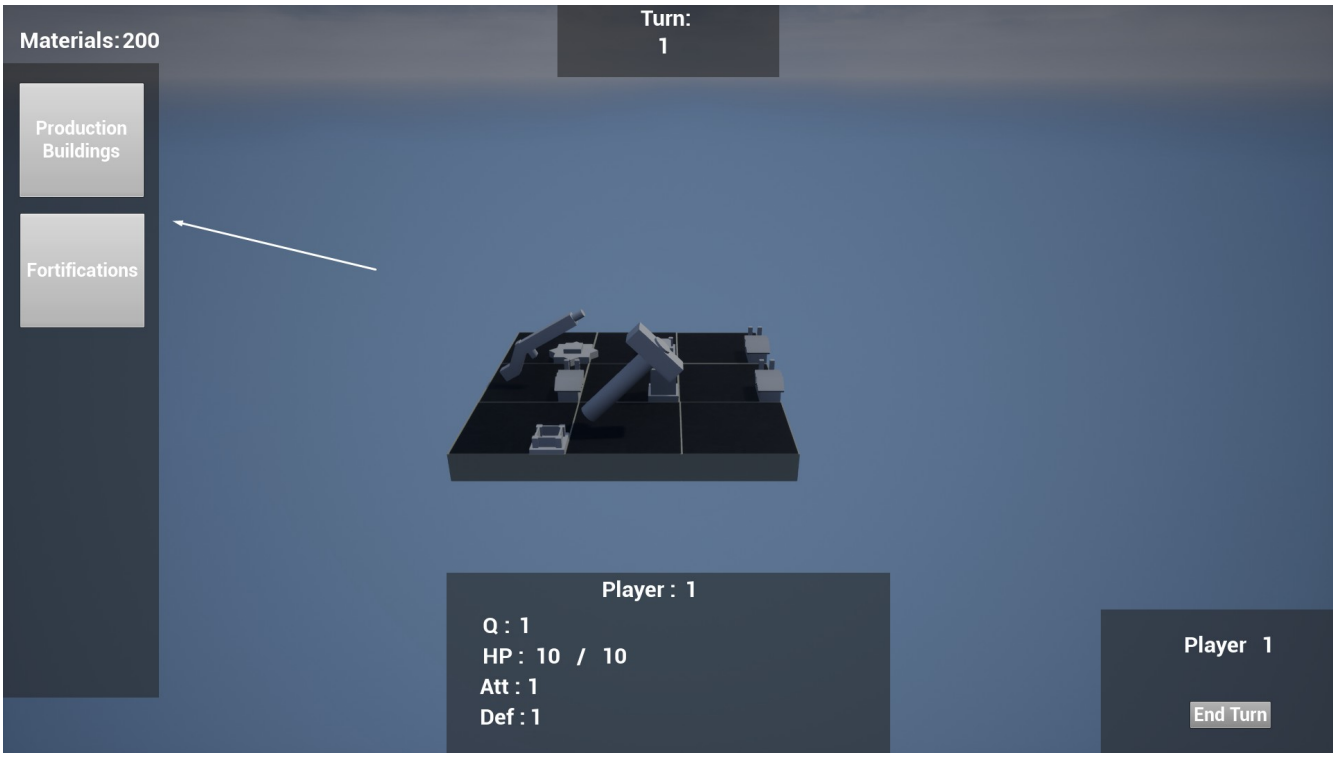


Фиг. 4.3 Потребителски интерфейс

- На фиг. 4.3 са използвани следните съкращения:
- Q – Quantity – това е броят на единиците на даденото поле. Колкото повече са, толкова по-силни стават
- HP – Health (Живот)
- Att – Щети
- Def – Защита

Ако закарате някой от вашите строители на поле, на което няма сграда, ще се отвори още едно меню – то се използва за строене на сгради (фиг 4.4.) В това меню има два избора:

- Production Buildings – това са всички сгради, които дават материали или единици в началото на всеки ход.
- Fortifications – това са всички сгради, които не дават бонуси в началото на всеки ход, но правят единиците, разположени в тях, по-силни.



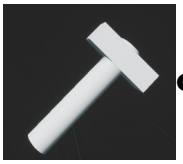
Фиг. 4.4 Меню за строене на сгради

Когато изберете една от двете опции ще ви излязат всички сгради от тази категория, заедно с цената им в материали. Ако имате достатъчно можете да цъкнете върху съответния бутон и да построите сградата. Това ще ви коства и един строител.

Ако искате да приключите хода си, цъкнете бутона с надпис „End Turn” в долната дясна част на екрана ви. Тогава пак ще се отвори екранът от фиг. 4.2, но този път на ход ще е следващият играч. Играта приключва ако един от двамата играчи изгуби всички свои сгради и единици. Ако това се случи, на екрана ще излезе надпис „Game over“ на черен фон.

4.2 Типове единици

За момента в играта има 2 типа единици – Builder (строител) и Infantry (пехота). Двете единици имат следните атрибути:



Фиг. 4.5 Строител:

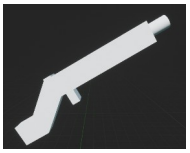
Щети = 1

Защита = 1

Живот = 10

Не може да отвръща на удар.

Създава се от сграда Builder’s Hall



Фиг. 4.6 Пехота:

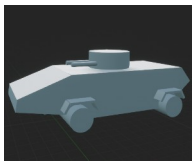
Щети = 4

Защита = 2

Живот = 25

Може да отвръща на удар.

Създава се от сграда Barracks



Фиг. 4.7 Бронирана кола

Щети = 8

Защита = 10

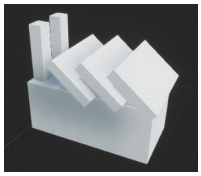
Живот = 50

Може да отвръща на удар.

Създава се от Vehicle Factory

4.3 Типове сгради

Производводствени:

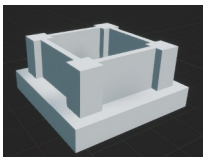


Фиг. 4.8 Factory (Фабрика) – 200 материала.

Дава по 50 материала в началото на всеки ход. Може да се превземе.

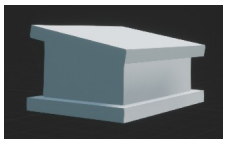
Фиг 4.9 Builder’s Hall – 200 материала.

Произвежда по един строител в началото на всеки ход. Може да се превземе.



Фиг 4.10 Barracks – 300 материала.

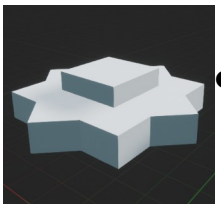
Произвежда по една пехота в началото на всеки ход. Може да се превземе.



Фиг 4.11 Vehicle Factory – 750 материала.

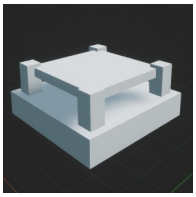
Произвежда по една бронирана кола в началото на всеки ход. Може да се превземе.

Укрепления:



Фиг. 4.10 Форт – 450 материала.

+4 щети
+7 защита
Не може да се превземе.



Фиг. 4.11 Бункер – 300 материала.

+2 щети
+4 защита
Може да се превземе.

4.4 Изтегляне на играта

В Github repository-то на играта има актуални инструкции откъде да се изтегли играта (Може да се наложи да се промени линка/хранилището, от което да се тегли играта). След като изтеглите .zip архива, може да го разархивирате. В разархивираната папка WindowsGame ще намерите файл Cisk.exe. С изпълнението на този файл се отваря играта.

Ако решите, че не ви се играе повече, можете просто да изтриете папката WindowsGame от вашия компютър.

ЗАКЛЮЧЕНИЕ

Това е моята реализация на походова стратегическа игра. В рамките на дипломната работа подобрих уменията си за работа със C++ и указатели, схванах основите на Unreal Engine 4, запознах се с Blueprints и visual programming и научих няколко важни неща за game development като процес. Успях да създам добра основа за бъдещо развитие на играта. Нови типове сгради и единици могат да се добавят в рамките на минути, а променянето на стари типове е също толкова лесно.

Бъдещето развитие на проекта се състои в няколко неща: оптимизиране на C++ кода и Blueprint графовете, добавянето на още сгради и единици, създаване на алгоритъм за автоматично генериране на големи карти (map), подобряване на визията и добавяне на анимации. Също така, би било много интересно имплементирането на нови механики, като например различен терен, който предоставя бонуси, или изследвания (research), които дават бонуси или отключват нови сгради и единици.

ИЗПОЛЗВАНА ЛИТЕРАТУРА И МАТЕРИАЛИ

[1] https://www.unrealengine.com/en-US/

[2] https://www.unrealengine.com/en-US/unreal-engine-5

[3] https://www.unrealengine.com/en-US/features

[4] https://www.geeksforgeeks.org/top-10-fastest-programming-languages/

[5] https://en.wikipedia.org/wiki/C%2B%2B

[6] https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/

[7] https://forums.unrealengine.com

[8] https://www.pexels.com/photo/two-man-holding-rifle-and-pistol-illustration-876343/

[9] https://www.youtube.com/watch?v=7H08G1sPNj8

[10] https://www.youtube.com/watch?v=Lmhhb0ROiqM

СЪДЪРЖАНИЕ

МНЕНИЕ НА НАУЧНИЯ РЪКОВОДИТЕЛ 3

СЪКРАЩЕНИЯ 4

УВОД 5

ПЪРВА ГЛАВА 8

1. Среди и технологии за създаване на игри 8

1.1. Основни принципи и технологии за реализиране на игри 8

1.1.1. Game Engine 8

Алтернативи на UE4 9

1.1.2. Език за програмиране 10

Алтернативи на C++ 11

1.1.3. Развойни среди 13

1.2 Разучаване на технологиите, използвани в дипломната работа 13

1.2.1 C++ в Unreal Engine 13

1.2.2 Blueprints в UE4 16

ВТОРА ГЛАВА 17

2. Проектиране на структурата на играта 17

2.1 Функционални изисквания: 17

2.1.1 Основни 17

2.1.2 Други изисквания 17

2.2 Избор на технологии и развойна среда 18

2.2.1 Game engine 18

2.2.2 C++ или Blueprints? 19

2.3 Алгоритъм 19

2.4. Архитектура на класовете 21

2.5 Как протича една игра? 22

ТРЕТА ГЛАВА 23

3. Програмна реализация на CISK 23

3.1. Основни класове 23

3.1.1. ABaseUnitClass 23

3.1.2. Сгради 29

3.1.3. AGameData 30

3.1.4. ATile 32

3.1.5. ACISKPlayer 33

3.2. Widget-и 34

3.2.1. MainMenu 34

3.2.2. NewTurnWidget 34

3.2.3. MaterialsWidget 35
3.2.4. UnitInfoBarWidget 36
3.2.5. BuildingsWidget 36
3.2.6 ProductionBuildingsWidget/FortificationsWidget 37
3.2.7 BuildingsInfoWidget 39
3.2.8 TurnWidget 39
3.2.9 GameOverWidget 40
3.3 Допълнителни класове 40
3.3.1 ABuilderUnit / Builder 40
3.3.2 AInfantryUnit / Infantry 41
3.3.3 AArmoredCar / Armored Car 42
3.3.4 AFactoryStructure / Factory 42
3.3.5 ABuildersHallStructure / BuildersHall 43
3.3.6 ABarracksStructure / Barracks 44
3.3.7 AvehicleFactoryStructure / Vehicle Factory 44
3.3.8 ABunkerStructure / AFortStructure / Bunker / Fort 45
3.3.9 Tile 45
3.3.10 GameMapController 47

ЧЕТВЪРТА ГЛАВА 50
4. РЪКОВОДСТВО ЗА ПОТРЕБИТЕЛЯ 50
4.1 Потребителски интерфейс и user input 50
4.2 Типове единици 54
4.3 Типове сгради 55
4.4 Изтегляне на играта 56

ЗАКЛЮЧЕНИЕ 57

ИЗПОЛЗВАНА ЛИТЕРАТУРА И МАТЕРИАЛИ 58

СЪДЪРЖАНИЕ 59