Форма № Н-9.02

Рівненський державний гуманітарний університет
Факультет математики та інформатики
Кафедра інформаційних технологій та моделювання

«До захисту допущено»
Завідувач кафедри
_____ Мороз І.П
Протокол № _____
від _____

Дипломний проєкт (робота)

ступеня бакалавр

зі спеціальності 122 «Комп'ютерні науки»

на тему: «Комп'ютерна система навчання гри в шахи на етюдах»

Виконав: студент 4 курсу, групи КН-41

Волощук Владислав Олександрович_____ _____
(прізвище, ім'я, по-батькові) (підпис)

Керівник: доцент, к.т.н. Сінчук А.М._____ _____
(посада, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент: _____ _____
(посада, науковий ступінь, прізвище та ініціали) (підпис)

м. Рівне, 2023 рік

Зміст

ВСТУП

Ранні версії шахових рушіїв були нездатні впоратися з глибиною гри. Їм не вистачило обчислювальної потужності для пошуку потенційних виграшних ходів. Лише в 1950 році все почало змінюватися. Алан Тюрінг, відомий математик, розробив комп’ютерну програму для гри в шахи. Відтоді програмісти в усьому світі працювали над підвищенням ефективності шахового двигуна.

Сьогодні комп’ютерні шахи дуже популярні. Справжні шахові фігури та шахівниця не в кожного є. А комп'ютерні шахи можна встановити майже на будь-який пристрій чи платформу, що дає змогу грати в шахи будь-де й будь-коли.

Також в комп'ютерних шахах зараз масово використовуються нейронні мережі та штучний інтелект для аналізу стану гри та здійснення ходів.

Тому **метою даної кваліфікаційної роботи** є опанування можливості розробки комп’ютерної візуалізації гри «Шахи» на етюдах, зокрема, задля розвитку пам'яті, концентрації, інтелекту, ментальної гнучкості, а також для цікавого проведення часу.

Для досягнення мети кваліфікаційної роботи були поставлені такі основні завдання:

- дослідити історію виникнення гри в шахи;
- проаналізувати та описати процес створення програмного забезпечення, у якому буде розроблено гру в «Шахи»;
- дослідити та описати різні класи функцій, які будуть використані в роботі;
- описати процес створення 3D моделей фігур, шахівниці, оточення та головного меню для гри;
- застосувати розроблене програмне забезпечення під час гри «людина проти комп’ютера».

Предмет дослідження – візуалізація комп’ютерної системи навчання гри в шахи типу «людина проти комп’ютера».

Об'єкт дослідження – гра «Шахи».

Апробація результатів дослідження:

- звітна науково-практична конференція професорсько-викладацького складу Рівненського державного гуманітарного університету, яка відбулася 18-19 травня 2023 року;
- XV Всеукраїнська науково-практична конференція "Інформаційні технології у професійній діяльності" Рівненського державного гуманітарного університету, яка відбулася 1 листопада 2022 року (м. Рівне);
- XII Міжнародна науково-практична конференція «Математика. Інформаційні технології. Освіта» Волинського національного університету ім. Л. Українки, яка відбулася 5-6 червня 2023 року (м. Луцьк).

Структура роботи. Бакалаврська робота складається з вступу, двох розділів, висновків, списку використаної літератури та додатки.

В процесі розробки та написанні письмової роботи були використані такі ресурси: технічна тематична література, технічні комп’ютерні журнали, наукові статті, тематичні відео, технічні сайти, блоги та форуми.

РОЗДІЛ 1. Дослідження та аналіз створення програмного забезпечення для комп’ютерної гри «Шахи»

1.1. Історія створення гри «Шахи»

Походження шахів залишається предметом суперечок. Немає достовірних доказів того, що шахи існували у формі, близькій до сучасної гри до 6 століття н.е. Ігрові фігури, знайдені в Китаї, Індії, Центральній Азії, Пакистані та інших країнах, які були визначені як старші за ті, тепер вважаються такими, що походять з більш ранніх дистанційно пов'язаних настільних ігор, часто із залученням гральних кісток, а іноді з використанням ігрових дощок зі 100 або більше квадратів [9, с.25].

Однією з таких ранніх ігор була військова гра під назвою чатуранга, санскритська назва бойового формування, згаданого в індійському епосі «Махабхарата» [9, с.28]. Чатуранга процвітала на північному заході Індії до 7 століття і вважається найбільш раннім попередником сучасних шахів, оскільки вона мала дві ключові особливості, виявлені у всіх пізніших варіантах шахів — різні фігури мали різну силу (на відміну від шашок і го), і перемога була заснована на одній фігурі, короля сучасних шахів.

Як еволюціонувала чатуранга, неясно. Деякі історики кажуть, що чатуранга, в яку, можливо, грали кістками на 64-квадратній дошці, поступово перетворилася на шатрандж (або чатранг), гру для двох гравців, популярну в північній Індії, Пакистані, Афганістані та південних частинах Центральної Азії після 600 року н.е. [9, с.29-30]. Шатрандж нагадував чатурангу, але з додаванням нової фігури - фірзана (радник). Гру в шатрандж можна було виграти шляхом усунення усіх наявних фігур супротивника (відкривши короля), або забезпечивши захоплення короля. Початкові позиції пішаків і лицарів не змінилися, але існували значні регіональні та часові відмінності для інших фігур.

Гра поширилася на схід, північ і захід, та дуже сильно змінювалася, шляхом додавання нових правил, фігур, дощок та методів гри. На Сході гра, яку несли буддиські паломники, торговці Шовкового шляху та інші, перетворилася на гру з розписними дисками, які часто розміщувалися на перетині ліній дошки, а не всередині квадратів. Близько 750 року н.е. шахи досягли Китаю, а до 11 століття вони прийшли до Японії та Кореї [9, с.32]. Китайські шахи, найпопулярніша версія східної гри, має 9 стовпців і 10 рядів, а також межу — річку між 5-м і 6-м рядами — яка обмежує доступ до ворожого табору і робить гру повільнішою, ніж її західний родич.

Чатуранга (або шатрандж) потрапила до Європи через Персію, Візантійську імперію і, можливо, найважливішу з усіх, Аравійську імперію, яка стрімво розширювалась. Найдавніша записана гра, була знайдена в рукописі 10-го століття, та була зіграна між багдадським істориком, який вважається улюбленцем трьох послідовних халіфів, і учнем [9, с.33].

Мусульмани принесли шахи в Північну Африку, Сицилію та Іспанію приблизно в 10 столітті. Східні слов'яни поширили його на Київську Русь приблизно в той же час. Вікінги перенесли гру аж до Ісландії та Англії і, як вважають, відповідальні за найвідомішу колекцію шахових фігур - 78 фігур з моржових та слонових кістках, які були знайдені на острові Льюїс на Західних островах у 1831 році і датуються 11 або 12 століттям.

Шахи і гра в кості періодично заборонялися королями і релігійними лідерами. Наприклад, король Людовик IX заборонив гру у Франції в 1254 році [9, с.37]. Однак популярності гри допоміг її соціальний статус: шаховий сет часто асоціювався з багатством, знаннями і владою. Шахи були фаворитом королів Генріха I, Генріха II, Іоанна та Річарда I Англійських, Філіпа II та Альфонсо X (Мудрого) Іспанії та Івана IV (Грозного) Росії. Шахи були відомі як королівська гра ще в 15 столітті.

Правила та їх зміни. Сучасні правила і зовнішній вигляд творів розвивалися повільно, з широко поширеними регіональними варіаціями. До 1300 року, наприклад, Пішак придбав можливість переміщати два квадрати на

своєму першому повороті, а не тільки по одному, як це було в шатранджі. Але це правило не завоювало загального визнання у Європі.

Шахи досягли найбільшого прогресу після двох важливих змін правил, які стали популярними після 1475 року. До цього часу Радник обмежувався переміщенням одного квадрата по діагоналі за раз. І, оскільки Пішак, який досяг восьмого ряду, міг стати лише Радником, просування пішака було відносно незначним фактором в ході гри. Але згідно з новими правилами, Радник зазнав зміни статі і отримав значно більшу рухливість, і став наймогутнішою фігурою з усіх - сучасною Королевою. Це, а також збільшення цінності просування пішаків додали шахам новий динамічний елемент. Крім того, фігура чатуранги під назвою Слон, яка була обмежена діагональним стрибком у два квадрати в шатранджі, стала Єпископом, збільшивши її дальність пересування більш ніж вдічі.

Поки цих змін не відбулося, мат був відносно рідкісним явищем, і частіше гра вирішувалася шляхом оголення короля. З новими повноваженнями королеви та єпископа окопна війна середньовічних шахів була замінена грою, в якій мат міг бути поставлений всього за два ходи.

Останні дві серйозні зміни в правилах - рокіровка і пасивне захоплення - зайняли більше часу, щоб завоювати визнання. Обидва правила були відомі в 15 столітті, але мали обмежене використання до 18 століття. Незначні зміни в інших правилах тривали до кінця 19 століття; наприклад, у багатьох частинах Європи ще в середині 19 століття було неприйнятно просувати Пішака до Королеви, якщо гравець все ще мав оригінальну Королеву.

Дизайн фігур. Зовнішній вигляд фігур чергувався між простим і орнаментованим ще з часів чатуранги. Просте оформлення фігур до 600 року н.е. поступово призвело до появи наборів фігурок, що зображують тварин, воїнів та різних вельмож. Але мусульманські набори 9-12 століть часто були нерепрезентативними і виготовлялися з простої глини або різьбленого каменю, дотримуючись ісламської заборони зображень живих істот. Вважається, що повернення до більш простих, символічних фігур шатранджа стимулювало

популярність гри, полегшуючи створення наборів і перенаправляючи увагу гравців зі складних фігур на саму гру.

Стилізовані набори, часто прикрашені дорогоцінними і напівкоштовними каменями, повернулися в моду, коли гра ширилася по Європі. Ігрові дошки, які мали однотонні клітини в мусульманському світі, отримали чорні та білі, або червоні та білі,клитини, що чергуються до 1000 року н.е. і часто виготовлялися з цінних порід дерева або мармуру.

Король став найбільшою фігурою і отримував корону, а іноді і складний трон булавою. Близьке ототожнення лицаря з Конем походить з чатуранги. Пішак, як найнижчий за владою та соціальним становищем, традиційно був найменшим і найменш представницьким з усіх фігур. Королева збільшилася в розмірах після 1475 року, коли її повноваження розширилися, і змінилася з радника-чоловіка на жінку-дружину короля. Єпископ був відомий під різними іменами - наприклад, «дурень» французькою мовою і «слон» українською мовою - і не був загальновизнаний аж до 19 століття. Зображення Тури також значно варіювалося. Іноді вона представлялась як вітрильник. В інших місцях це був воїн на колісниці або замкова башта.

Стандарт для сучасних наборів був встановлений близько 1835 року з простою конструкцією та формами англійцем Натаніелем Куком. Після того, як він був запатентований у 1849 році, дизайн був схвалений Говардом Стонтоном, тоді найкращим гравцем у світі; через широке просування Стонтона, згодом він став відомий як дизайн Стонтона. Тільки набори, засновані на дизайні Стонтона, сьогодні допускаються до міжнародних змагань [9, с.42-51].

В історії шахів було дуже багато неправди, суперечностей та підозрілих історичних моментів. Але історикам все ж вдалося визначити історію шахів з доволі правдивим розвитком подій. Звісно ж суперечності нікуди не ділися, але їх вдалося успішно прибрати із загальної низки подій. Як вже було зазначено раніше, загально прийнятої версії щодо появи шахів нема, але усі існуючі версії досить близькі одна до одної. Відрізняються лише деякі назви

та дати. Історики досі не здаються та вивчають дедалі старіші рукописи та згадки, задля визначення більш точної історії винайдення шахів.

1.2. Історія розробки програмного забезпечення для гри «Шахи»

Перші графічні шахові ігри з’явилися лише коли комп’ютери змогли відображати яку-небудь графіку. Звісно ж при появі комп’ютерів вчені першочергово намагалися створити алгоритм, за допомогою якого комп’ютер міг би грати проти людини. Графічна візуалізація йшла на другий план. Але все ж таки потужності комп’ютерам тих років дуже не вистачало. Гру зі звичайними людьми реалізувати змогли, але грати проти гросмейстерів комп’ютери точно не змогли б. І ось у 1976 році компанія Micro-Ware розробила на комп’ютері KIM-1 гру «Microchess» (рис. 1). Це була перша спроба графічно візуалізувати шахи [4, с.56].

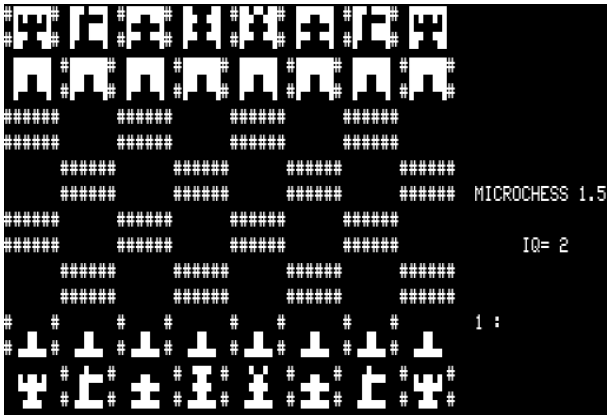

Рис. 1.1. Відео гра «Microchess»

Після дуже вдалих продажів «Microchess», гру портували на багато інших комп’ютерів від різних фірм. Інші компанії теж почали випускати свої аналоги шахів. Наприклад «Video Chess» для комп’ютерів Atari 2600 (рис. 1.2), «Colossus Chess» для Comodoro (рис. 1.3), або ж «Sargon Chess» для Apple II (рис. 1.4).

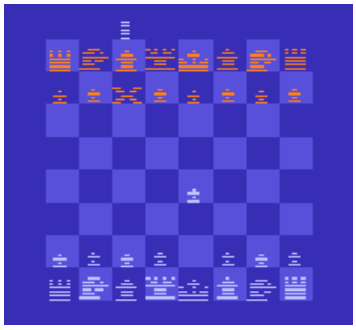

Рис. 1.2. Video Chess

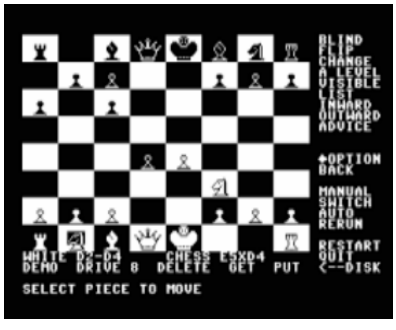

Рис. 1.3. Colossus Chess

Рис. 1.4. Sargon Chess

Також графічні версії шахів розроблялися для багатьох інших комп’ютерів. Але їх розглядати ми не будемо. Набагато більше ігор було розроблено для домашніх ігрових консолей. Шахи в тому числі. Кількість консольних шахових ігор була такою великою, що перерахувати всі не вдасться. Але можна виділити для кожної консолі на рис. 1.5 – 1.16 найпопулярніші.

Рис. 1.5. NES – Battle Chess

Рис. 1.6. Sega Master system – Master Chess

Рис. 1.7. Game Boy - ChessMaster

Рис. 1.8. Sega Mega Drive - Chess

Рис. 1.9. SNES - ChessMaster

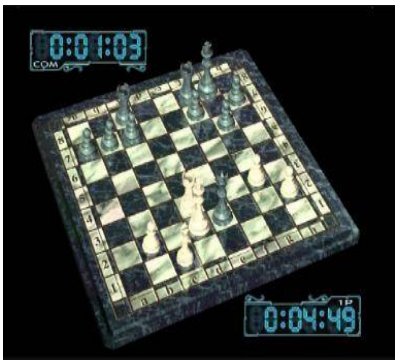

Рис. 1.10. PlayStation - Chess

Рис. 1.11. Game Boy Color - ChessMaster

Рис. 1.12. Nintendo 64 – Virtual Chess

Рис. 1.13. Sega Dreamcast - DreamChessй

Рис 1.14. PlayStation 2 – Chess Challenger

Рис 1.15. Game Boy Advanced - ChessMaster

Рис 1.16. PlayStation Portable – Taisen Chess

Кожна з цих консолей була популярна у свій час й мала величезну кількість підтримуваних ігор та програм. Зараз, на жаль, більшість старих ігрових консолей залишилися в минулому, але деякі живуть й до сьогодні. Нечисленні ентузіасти навіть примудряються, з неймовірними труднощами, створювати ігри для старих та мертвих консолей. Звісно ж ці ігри створюються не так як сучасні. Використовується багато застарілих, або ж взагалі мертвих, мов програмування, методів та алгоритмів. Але, все ж, створювати ігри для таких платформ все ще можливо.

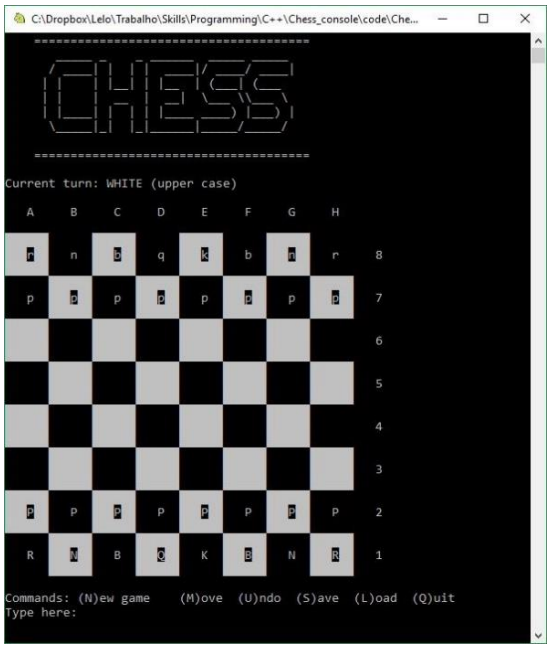

Рис. 1.17. Шахи в консолі Windows написані на C++

Також майже всі сучасні (і не тільки) мови програмування, що підтримують консоль Windows, дають змогу створювати та запускати ігри та програми на ній. Наприклад, можна знайти безліч шахових ігор для консолі, або взагалі написати їх самому (рис. 1.17). Цей спосіб створення ігор можна вважати застарілим, але консольні програми все ще створюються (зазвичай розробники створюють саме консольні програми, коли не хочуть додавати графічний інтерфейс, або коли в цьому немає потреби).

Також деякі ентузіасти дуже полюбляють створювати ігри в деяких застосунках від Microsoft. Особливо сильно вони люблять Excel. За всі роки існування Excel у ньому створили та портували безліч повноцінних ігор. Наприклад Minecraft [10] та Doom (який відомий тим, що його можна портувати всюди куди завгодно) [15]. Шахи в Excel теж є [12]. В PowerPoint, наприклад, теж можна створювати ігри, наприклад простенькі платформери [11], або ритмічні ігри (типу Osu) [3]. Щоправда, вони будуть дуже обмеженими.

Рис. 1.18. Повноцінні 3D шахи розроблені для браузера

Як один зі способів створення гри «Шахи», також можна виділити гру в браузері, створену за допомогою стандартних HTML, CSS та JavaScript. За допомогою HTML та CSS розробляється дизайн, а за допомогою JavaScript – сама логіка гри. Це можуть бути як і дуже простенькі шахи, так і повноцінні (рис. 1.18).

Ще можна пригадати середовище та інтерпретовану динамічну візуальну мову програмування Scratch. Так, в першу чергу Scratch орієнтований на дітей та початкове знайомство з основними концепціями та ідеями програмування. Але в руках дорослих та досвідчених людей Scratch перетворюється на дуже потужний рушій для створення ігор [2]. Шахи там теж є, і досить багато [1]. Але Scratch вже, можна сказати, вже в минулому. Бо єдині, хто створює на ньому ігри – рідкісні давні фанати.

Щодо ігрових рушіїв, то їх дуже багато. На Вікіпедії є величезний список [8]. Там зібрані популярні, або ж відомі рушії. Але список далеко не повний. Туди не включені дуже старі та маловідомі рушії, а також ті, які були розроблені звичайними людьми просто для розваги, бо, як вже могло стати

зрозуміло, таких рушіїв набагато більше ніж офіційних. Більшість з них знаходиться на Github і мають відкритий вихідний код, але ними мало хто користується.

1.3. Сучасне програмне забезпечення

В даній роботі, для створення комп'ютерної візуалізації гри «Шахи» було використано таке програмне забезпечення:

- Unity – для створення кінцевої гри;
- Microsoft Visual Studio – для написання коду;
- Blender – для створення модельок шахівниці та фігур;
- Adobe Photoshop – для створення текстури для моделей.

Трохи детальніше про кожне програмне забезпечення.

1.3.1. Можливості роботи Unity

Unity – це ігровий рушій професійної якості, який використовується для створення відеоігор, орієнтованих на різноманітні платформи. Це не лише інструмент для професійного розвитку, який щодня використовують тисячі досвідчених розробників ігор, але й один з найдоступніших сучасних інструментів для розробників-початківців.

Створені за допомогою Unity ігри можуть бути у дво- або тривимірній графіці, та працювати на настільних комп'ютерах, мобільних пристроях, гральних консолях та на пристроях віртуальної чи доповненої реальності. Усього рушій може запускатися на 25 платформах.

Логіка ігор та програм пишеться за допомогою мови програмування C#. Раніше також була можливість використовувати Boo та JavaScript, але розробники відмовились від їх підтримки [13].

Візуально робочий процес відрізняється від більшості інших середовищ розробки ігор. У той час як інші інструменти для розробки ігор часто являють

собою складну суміш із різних частин, які не працюють належним чином, або, можливо, бібліотеку, яка вимагає від вас налаштувати власне інтегроване середовище розробки (IDE), ланцюжок збірок тощо, робочий процес розробки в Unity закріплено за допомогою складного візуального редактора (рис. 1.19).

Редактор використовується для компонування сцен у вашій грі та

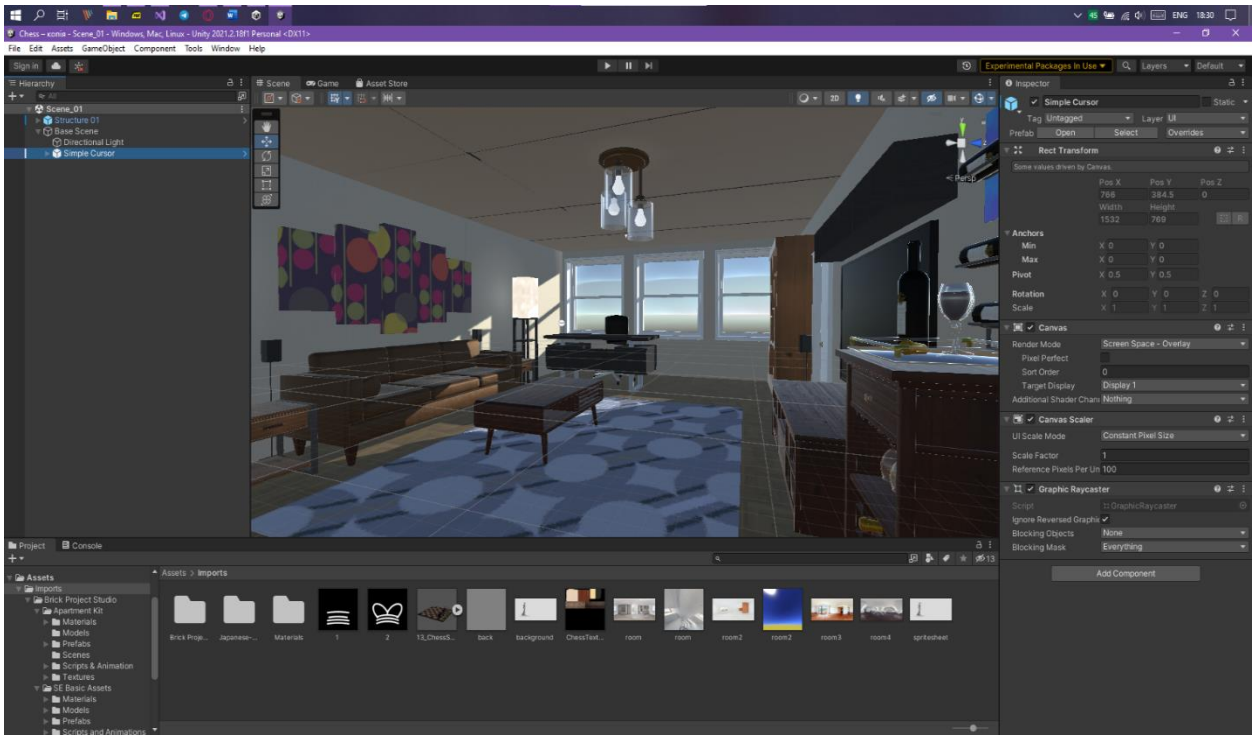

Рис. 1.19. Інтерфейс Unity

об’єднання графічних ресурсів і коду в інтерактивні об’єкти. Принадність цього редактора полягає в тому, що він дає змогу швидко й ефективно створювати ігри професійної якості, надаючи розробникам неймовірну продуктивність, водночас використовуючи широкий список новітніх технологій у відеоіграх.

Редактор особливо корисний для виконання швидкої ітерації, відточування гри через цикли прототипування та тестування. Ви можете налаштовувати об’єкти в редакторі та переміщати речі навіть під час роботи гри. Крім того, Unity дозволяє налаштовувати сам редактор, написавши сценарії, які додають нові функції та меню до інтерфейсу.

Окрім значних переваг редактора у продуктивності, іншою головною перевагою набору інструментів Unity є високий рівень кросплатформної підтримки. Unity є багатоплатформним не тільки з точки зору цілей розгортання (програму можна розгорнути на ПК, мобільних пристроях, консолях або в інтернеті), але й багатоплатформним з точки зору інструментів для розробки (ви можете розробити гру для Microsoft Windows або Apple macOS). Така платформонезалежність значною мірою пояснюється тим, що Unity розроблявся як програмне забезпечення лише для Mac, а пізніше був перенесений на Windows.

Небагато ігрових рушіїв підтримують стільки цілей розгортання, як Unity, і жоден не робить розгортання на кількох платформах таким простим, тому що майже всі ігрові рушії дуже туго прив'язані до однієї платформи.

Тому для розробки гри «Шахи» Unity підходив якнайкраще.

1.3.2. Принципи використання Microsoft Visual Studio

Microsoft Visual Studio – інтегроване середовище розробки (IDE). Це програма, яка дозволяє розробляти, писати, редагувати, компілювати та виконувати код. Використовується для розробки комп'ютерних програм, як консольні програми, так і графічні, а також вебсайти, вебзастосунки та мобільні застосунки.

Може створювати як машинний код, так і керований код для всіх платформ, що підтримуються Microsoft Windows, Windows Mobile, Windows Phone, Windows CE, .NET Framework, .NET Compact Framework та Microsoft Silverlight.

Visual Studio містить редактор коду (рис. 1.20), що підтримує IntelliSense (компонент завершення коду). Вбудований налагоджувач працює як налагоджувач на рівні вихідного коду, так і на рівні машинного. Інші вбудовані інструменти включають конструктор для створення програм із графічним інтерфейсом, веб-конструктор, конструктор класів і конструктор

схем бази даних. Він дозволяє використання плагінів, які розширюють функціональні можливості майже на кожному рівні, включно з додаванням підтримки розподілених систем керування версіями файлів (наприклад, Subversion і Git) і додавання нових наборів інструментів, таких як редактори та візуальні дизайнери для доменних мов або наборів інструментів для інших аспектів життєвого циклу розробки програмного забезпечення.

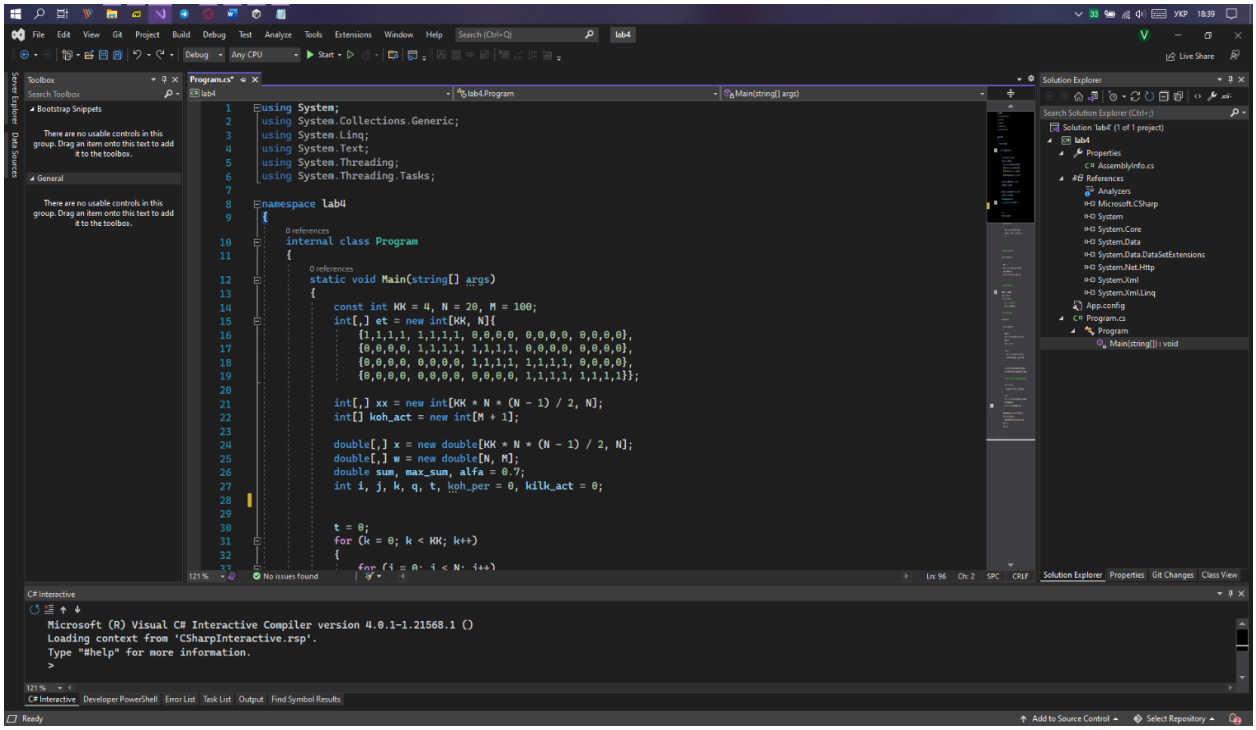

Рис. 1.20. Інтерфейс Microsoft Visual Studio

Visual Studio підтримує 36 різних мов програмування та дозволяє редактору коду та налагоджувачу підтримувати майже будь-яку мову програмування, якщо існує спеціальна служба для цієї мови. Вбудовані мови включають C, C++, C++/CLI, Visual Basic .NET, C#, F#, JavaScript, TypeScript, XML, XSLT, HTML, CSS. Інші мови , які як Python, Ruby, Node.js та M доступні лише через плагіни. Java і J# підтримувались в минулому [14].

Visual Studio є одним з найзручніших інтегрованих середовищ розробки, тому саме його було використтано у роботі. Код писався на C#.

1.3.3. Переваги роботи в Blender

Blender — це пакет для створення 3D-контенту, який пропонує широкий спектр основних інструментів, включаючи моделювання, візуалізацію, анімацію, редагування відео, VFX, композицію, текстурування, такелаж і багато типів симуляції. Він є кросплатформним із графічним інтерфейсом OpenGL, єдиним для всіх основних платформ. [6]

Для запуску багатьох програм потрібен час, особливо якщо запущено іншу програму. Але з Blender це не проблема. Щоб запустити та розпочати свій проєкт, знадобляться лише секунди.

Можливість налаштувати інтерфейс Blender (рис. 1.21) відповідно до ваших уподобань є однією з особливостей, яка приваблює багатьох його користувачів. Його можна налаштувати за допомогою сценаріїв Python.

Рис. 1.21. Інтерфейс Blender

У Blender є багато вбудованих доповнень, які ви можете досліджувати, щоб значно полегшити свій проєкт. Доповнення потрібно купувати. Але також є безкоштовні доповнення, створені розробниками в спільноті, які можна завантажити.

Blender пропонує неупереджений, фізично заснований механізм трасування шляху, який імітує роботу камери в реальному світі. Він оптимізований для GPU анімації та візуалізації.

До того ж Blender є вільним програмним забезпеченням, має відкритий вихідний код та розповсюджується під ліцензією GNU GLP.

За допомогою Blender було створено 3D моделі дошки та шахових фігур.

1.3.4. Використання Adobe Photoshop

Adobe Photoshop (рис. 1.22) — це програмне забезпечення для редагування зображень і ретушування фотографій для використання на комп’ютерах Windows або MacOS.

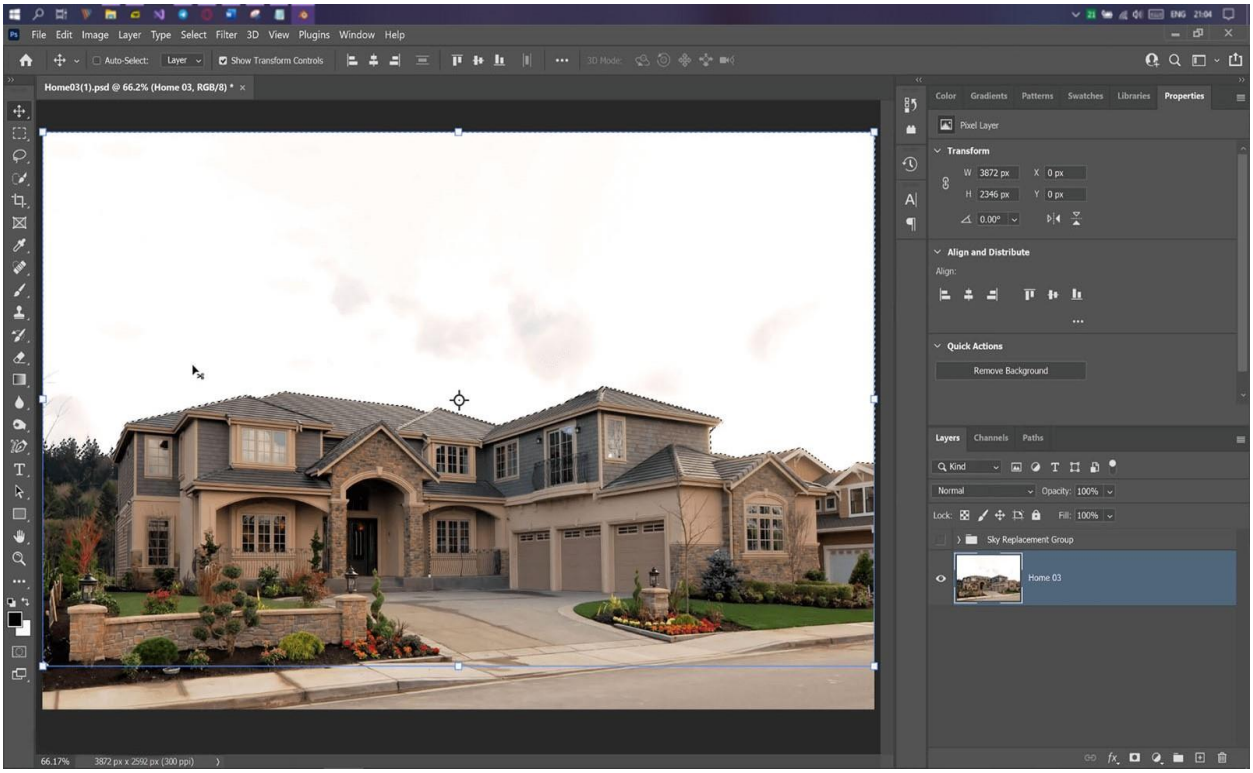

Рис. 1.22 Інтерфейс Photoshop

Photoshop пропонує користувачам можливість створювати, вдосконалювати або іншим чином редагувати зображення, ілюстрації та ілюстрації. Зміна фону, імітація картини з реального життя або створення альтернативного погляду на Всесвіт – все це можливо за допомогою Adobe Photoshop.

Це найпоширеніший програмний інструмент для редагування фотографій, обробки зображень і ретушування численних форматів файлів зображень і відео. Інструменти Photoshop дозволяють редагувати як окремі зображення, так і великі партії фотографій. Існує кілька версій Photoshop, зокрема Photoshop CC, Photoshop Elements, Photoshop Lightroom і Photoshop Express, версія Photoshop для iOS з обмеженими можливостями.

Функціональність Photoshop можна розширити за допомогою додаткових програм, які називаються плагінами (або плагінами) Photoshop. Існують різні типи плагінів, як-от фільтр, експорт, імпорт, вибір, корекція кольору та автоматизація. Найпопулярнішими плагінами є фільтри (також відомі як плагіни 8bf), доступні в меню «Фільтр» у Photoshop. Плагіни фільтрів можуть змінювати поточне зображення або створювати вміст.

За допомогою Photoshop були створені текстури для наших моделей.

РОЗДІЛ 2.Розробка програмного забезпечення гри «Шахи»

2.1. Моделювання фігур та шахівниці

Правила гри в шахи широко відомі, тому в цій роботі вони не будуть наведені. Але освіжити пам’ять можна на Chess.com – найбільшому та на найбільш відвідуваному шаховому сайті в інтернеті.

Рис. 2.1. Шахові фігури

Так от, для гри в шахи потрібні: шахівниця – спеціальна дошка поділена на 64 світлі та темні клітини (поля); 16 світлих (білих) та 16 темних (чорних) фігур (рис. 2.1);

Моделювалося усе в Blender.

Спочатку було створено пішака. Для цього було імпортовано профільне зображення пішака в Unity (рис. 2.2) та додаємо циліндр (рис. 2.3).

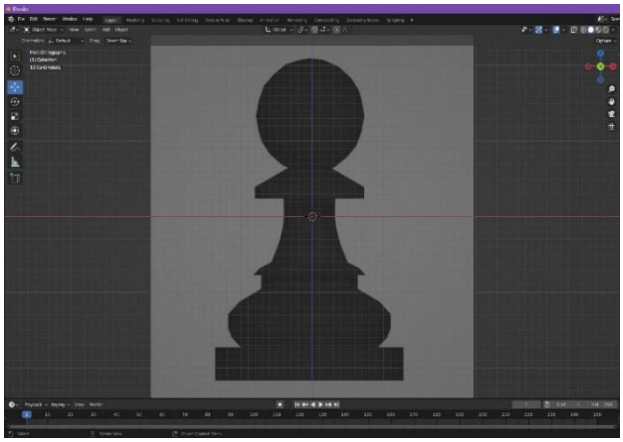

Рис. 2.2. Імпорт зображення

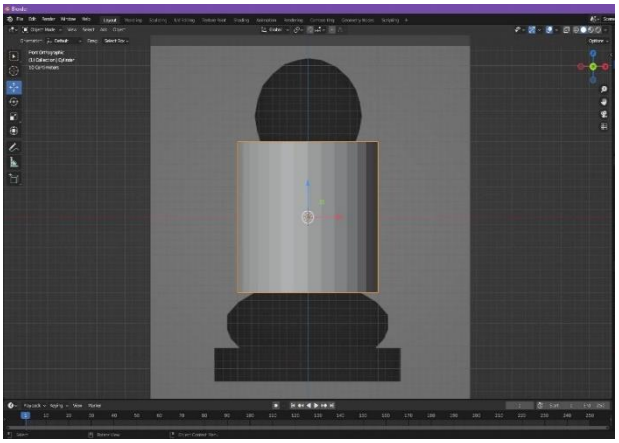

Рис. 2.3. Додавання циліндру

Саму фігуру було створено із циліндра за допомогою екструдування (рис. 2.4 та 2.5) – інструменту, який дозволяє змінювати геометрію об’єктів в режимі редагування за рахунок створень копій ребер і граней шляхом свого роду видавлювання – переміщення та зміну розмірів.

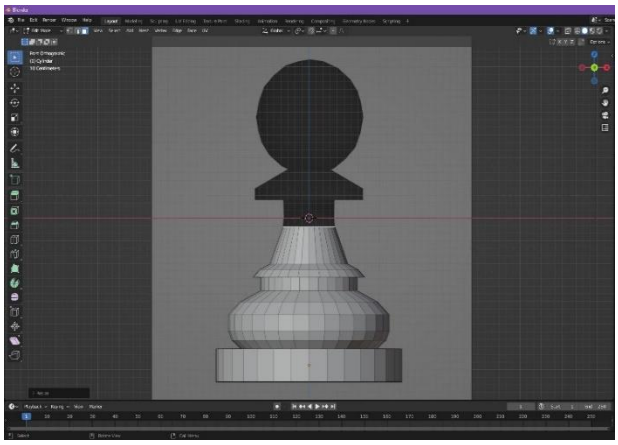

Рис. 2.4. Перший етап екструдування

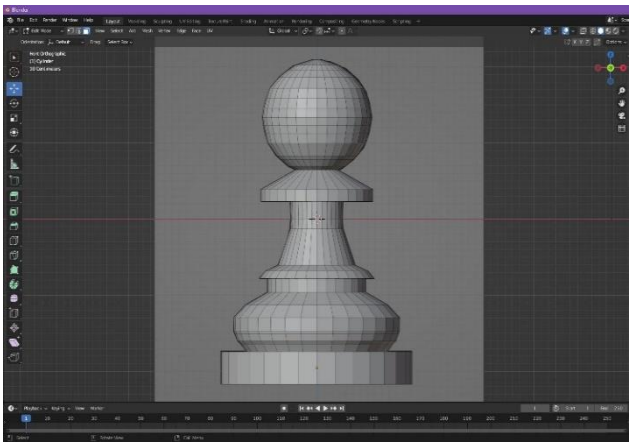

Рис. 2.5. Другий етап екструдування

Результат екструдування показано на рис. 2.6.

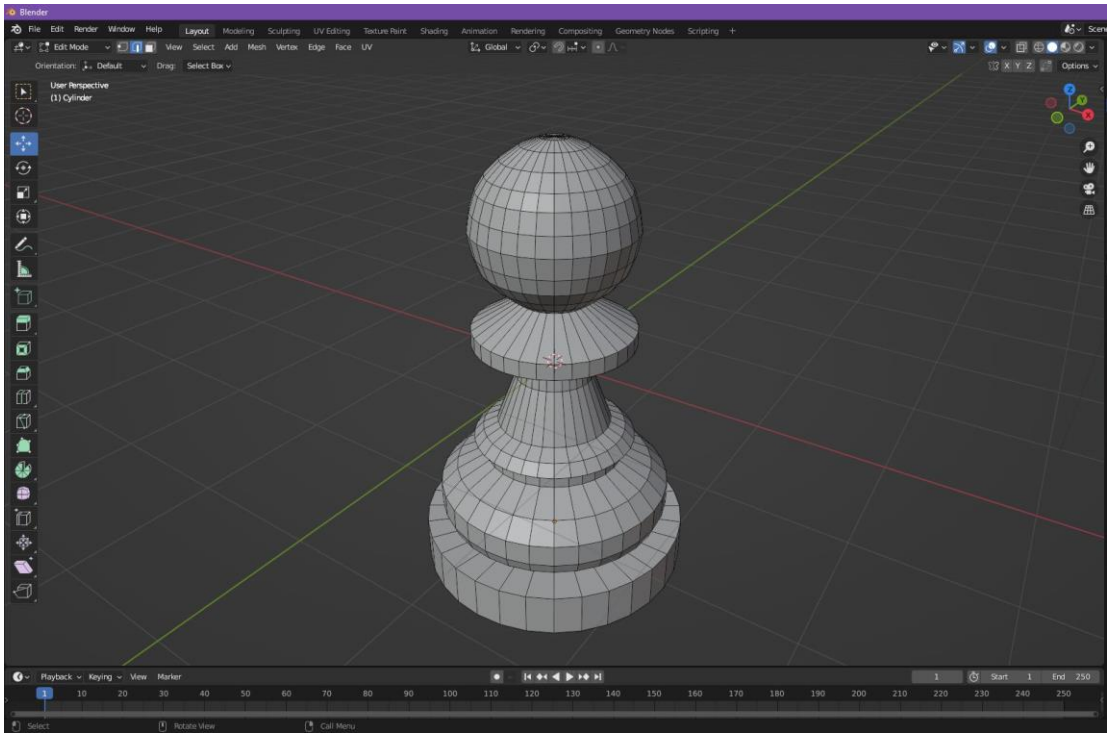

Рис. 2.6. Результат екструдування

Цим же способом були створено всі інші фігури.

Далі було додано текстуру до нашого пішака. Спочатку було створено для нього новий матеріал. Створення матеріалу для нашої фігури автоматично додає купу налаштувань, як от наприклад, зміну базового кольору (рис. 2.7) на зображення (рис. 2.8), яке буде слугувати нам текстурою.

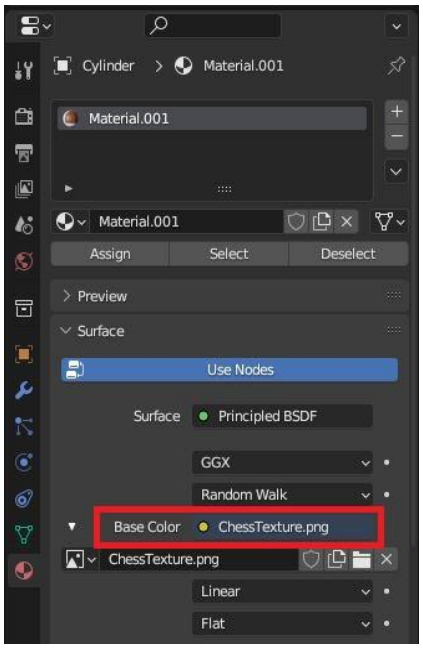

Рис. 2.7. Додавання текстури

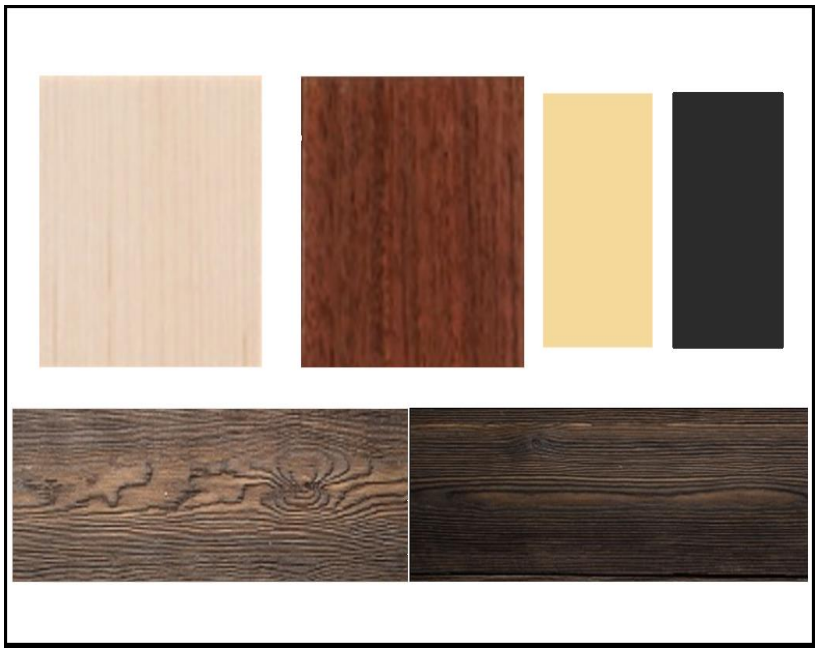

Рис. 2.8. Текстура

Текстура була додана, але вона відображалася некоректно. За допомогою UV Mapping (процес переносу 3D сітки моделі на 2D поверхню для подальшого розмальовування або ж редагування текстури) можна пересувати нашу текстуру по фігурі. Для зручнішого відображення нашої фігури у вікні редактору можна скористатись режимом Project from View (Проектування з Огляду), який вмикається в меню UV Mapping. саме меню викликається кнопкою U в режимі редагування. Зрівняти відображення з цим режимом та без можна на рис. 2.9 та 2.10.

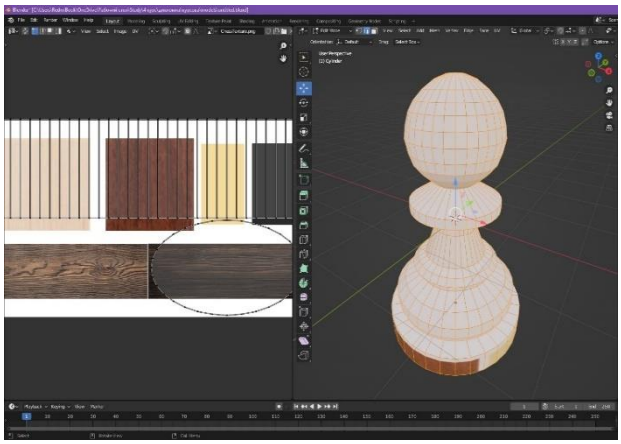

Рис. 2.9. Без Project from View

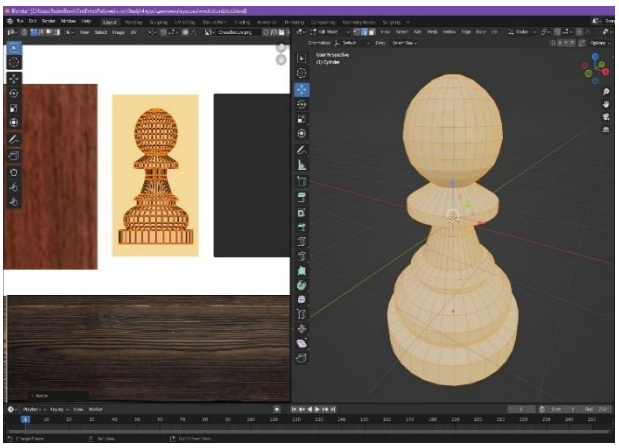

Рис. 2.10. З Project from View

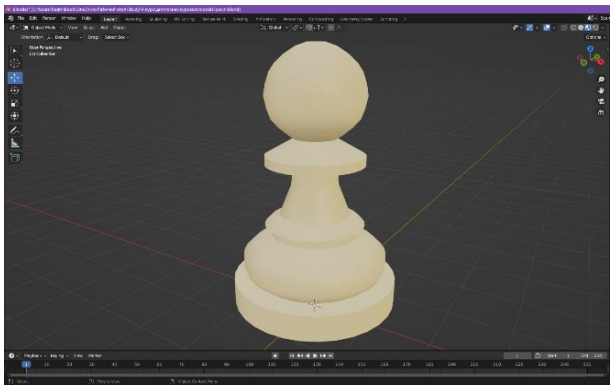

Рис. 2.11. Результат текстурування

Тепер у фігури є текстура. Але модель все ще трохи кутаста. Для згладжування можна використати кілька способів. Наприклад, Shade Smooth, або ж використати модифікатор Subdivision Surface. Тут було використано Shade Smooth, тобто Згладжене тонування. Результати на рис. 2.11.

Далі модель була збережена з розширенням .fbx, яке дозволило імпортувати її в Unity.

Схожим чином створилися усі інші фігури. Для створення фігур іншого кольору потрібно всього лиш перемістити мапу моделі на інший колір текстури (рис. 2.10).

Далі була створена дошка. Вона є спрощеною, тобто без написів на бокових сторонах.

Спочатку було створено куб, який було підлаштованопід розміри фігур (рис. 2.12), а потім продубльовано (рис. 2.13).

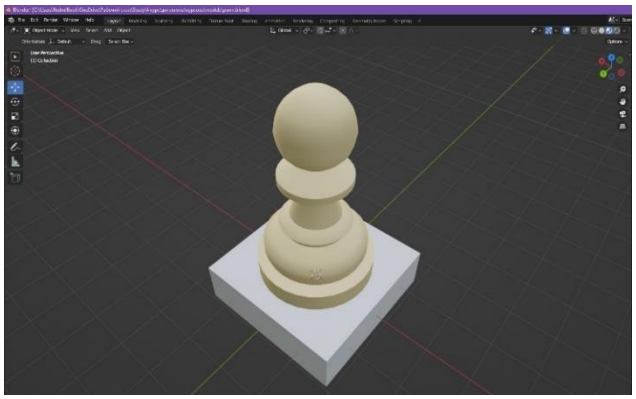

Рис. 2.12. Створення клітини

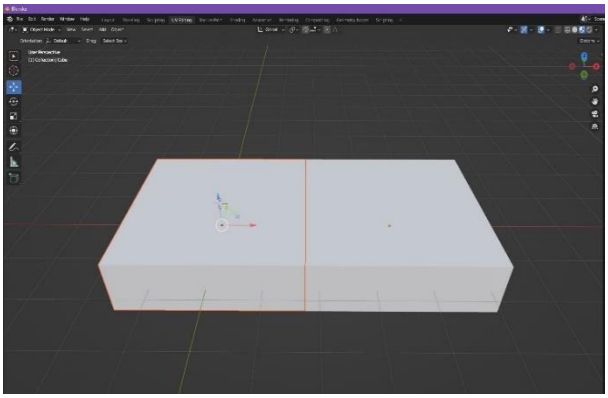

Рис. 2.13. Дублювання клітини

Далі тим же способом було додано текстуру (було підключено матуріал для обох кубів та імпортоване зображення-текстура). Далі було виконане переміщення мапи моделей кубів на певний колір дерева (рис. 2.14).

Наступним кроком було дублювання цих двох кубів до поки не вийшла дошка 8 на 8 клітин (рис 2.15 – 2.17).

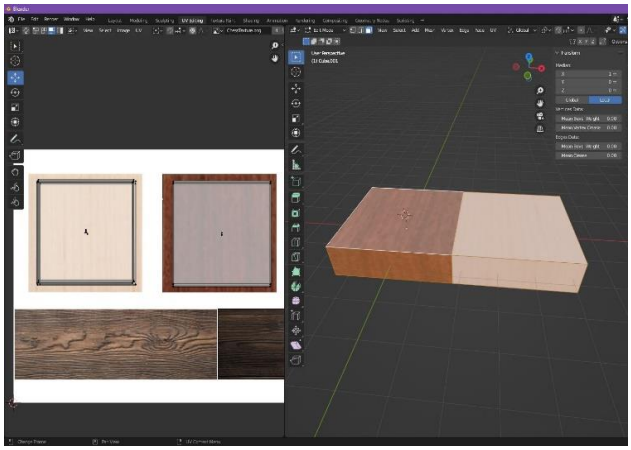

Рис. 2.14. Додавання текстури

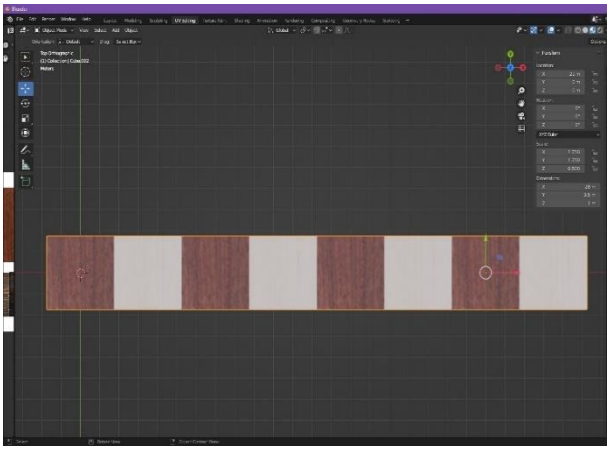

Рис. 2.15. Дублювання

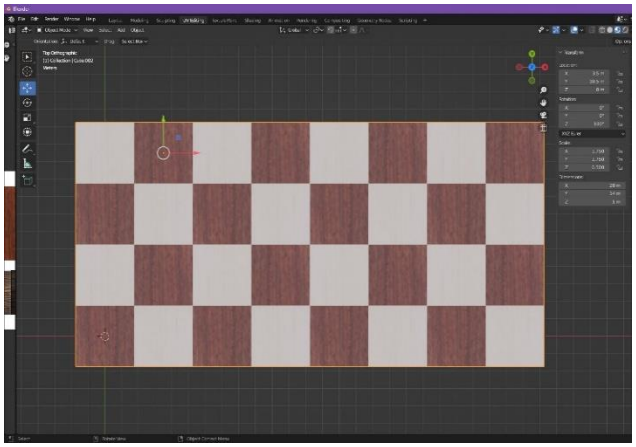

Рис. 2.16. Дублювання

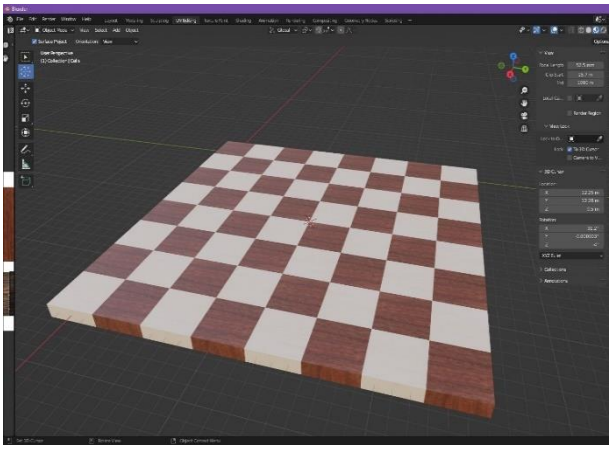

Рис. 2.17. Дублювання

Далі був доданий обід для нашої дошки. Для цього було створено куб, змінено його розміри та обрізано краї, як показано на рис. 2.18

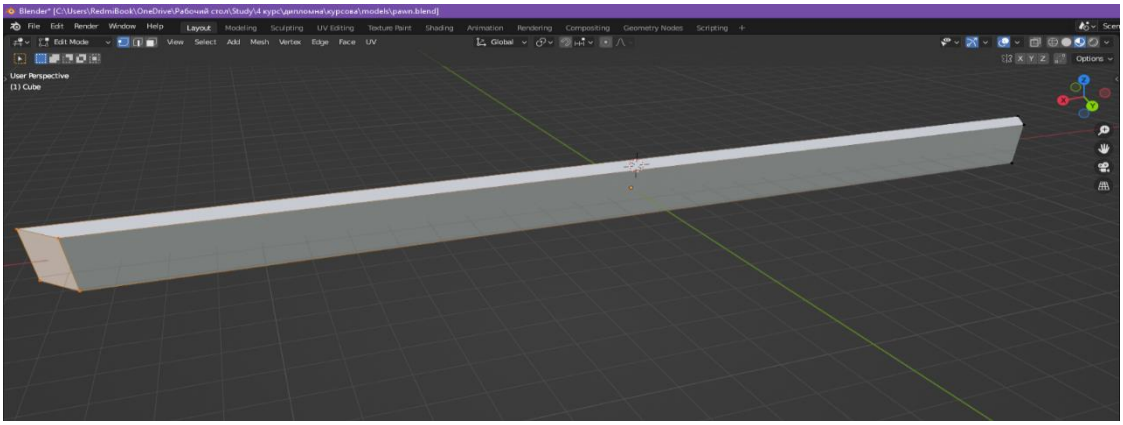

Рис. 2.18. Створення обідка дошки

Потім продубльовано обідок на кожну сторону (рис. 2.19) та додано текстури (рис. 2.20).

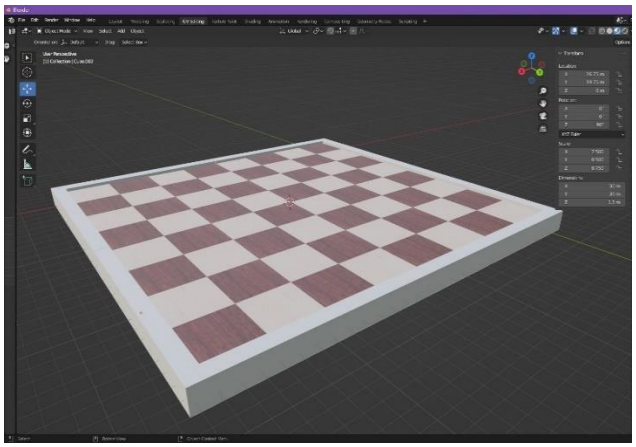

Рис. 2.19. Дублювання обідка

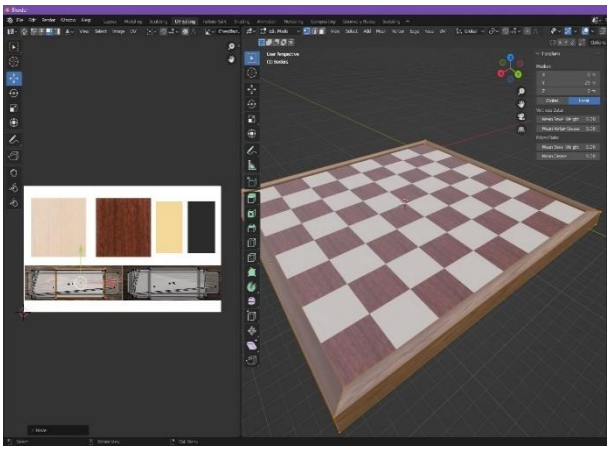

Рис. 2.20. Текстурування обідка

Все. Наша шахова дошка готова.Тому її було збережено як модель з розширенням .fbx.

Також було створено квадрати індикатори (рис. 2.21): жовтий – індикатор курсора, синій – відображення дозволених ходів фігури, червоний – підсвітка фігур, які можна побити.

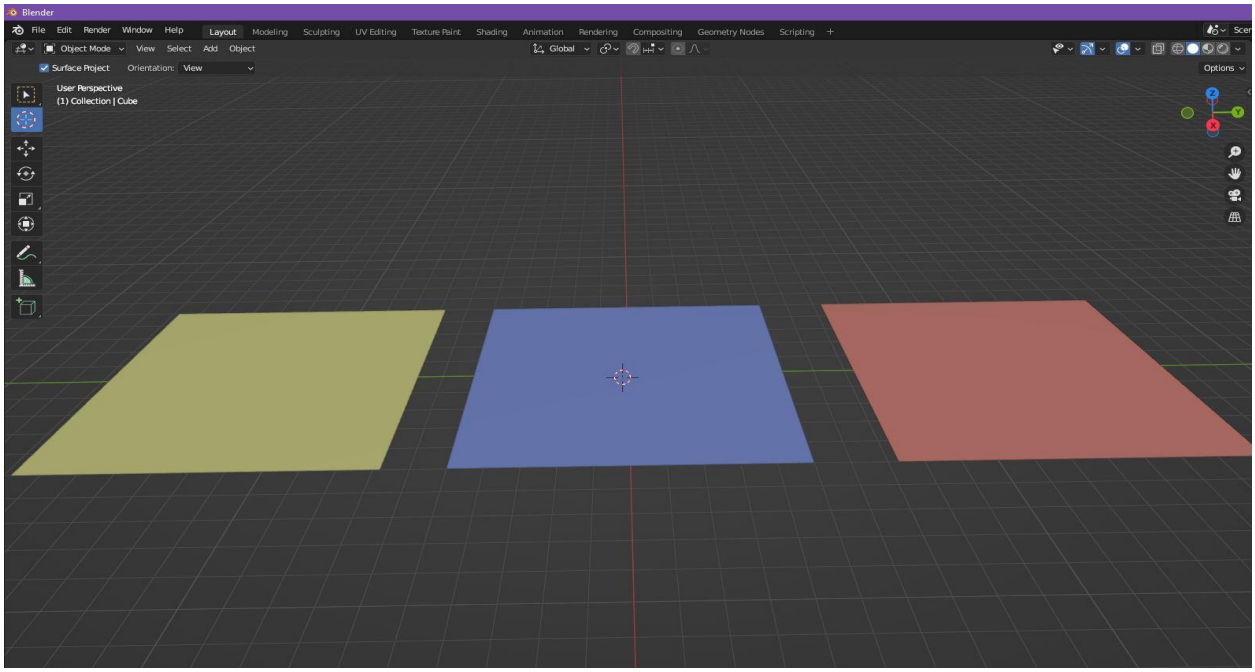

Рис. 2.21. Квадрати індикатори

Остаточний вигляд усіх фігур та шахової дошки зображений на рис. 2.22.

Рис. 2.22. Кінцевий варіант усіх фігур та дошки

Далі для нашої шахівниці в гру було додане оточення.

2.2 Створення декораційного оточення

Задля того, щоб не грати в шахи посеред білої пустоти, було додано до наших шахів оточення, а саме кімнату зі столом, на якому, власне, й буде розміщена дошка з фігурами.

Було кілька варіантів створення кімнати. Найпростіший – це використати панорамне зображення певної кімнати (рис. 2.23), та розтягнути його на внутрішній стороні сфери. А саму сферу збільшити та помістити в центр модельку стола разом з нашою дошкою та фігурами.

Ідея дуже проста, але для нас не підходить. Тому було використано варіант з тривимірною моделлю кімнати (рис. 2.24).

Рис. 2.23. Панорамне зображення кімнати

Рис. 2.24. Тривимірна модель кімнати

В даній роботі було використано готову тривимірну кімнату з Asset Store в Unity.

Asset Store – це онлайн магазин, в якому можна придбати різні ассети для Unity, такі як 3D моделі, звуки/музика, набори UI, шейдери, набори спрайтів, а також різноманітні інструменти, які роблять процес розробки більш комфортним та швидким. Звісно ж в Asset Store є багато безкоштовних

моделей, які можна використовувати у своїх проєктах. Тому цю кімнату (рис. 2.24) було завантажено та імпортовано саме звідти.

2.3 Створення головного меню

Наше меню складається з кнопок «Нова гра», «Продовжити гру», «Налаштування» та «Вихід» (рис. 2.25).

Кнопка «Нова гра» завантажує нову сцену з кімнатою та шахами. Кнопка «Продовжити гру» завантажує сцену з попередньою грою. Кнопка «Налаштування» активує екран для налаштування гри (рис. 2.26).

Рис. 2.25. Головне меню

Рис. 2.26. Налаштування

Налаштування можна розширяти та налагоджувати. Кнопка «Вихід» вимикає програму. Також під час гри, при натисненні кнопки «Esc» відбудеться вихід в головне меню.

2.4 Написання вихідного коду гри

2.4.1 Основна структура коду

Увесь код гри писався в інтегрованому середовищі розробки Visual Studio на мові програмування C#.

Для зручності та «чистоти» весь код повинен бути правильно структурований. Кожен великий клас повинен бути в окремому файлі. Кожна важлива функція теж мусить бути відокремлена. І повинен бути основний файл, який керує всім іншим кодом.

Загалом було створено 18 файлів з кодом гри. Приблизне загальне число рядків: 1600. Це число може змінюватися в залежності від змін самого коду. Ось список та опис усіх файлів з класами у порядку важливості:

- GameManager – найголовніший і найбільший клас. Тут створюється і виставляється дошка та усі фігури. Тут описані різного роду інструкції для створення, додавання, вилучання фігур, додаткові медоди для пересування та захоплення фігур, а також ще кілька допоміжних методів для зміни гравця, пошуку всіх можливих ходів фігури тощо.
- MainMenu – тут описані інструкції завантаження основних сцен із грою, обробники основних кнопок меню «Нова гра», «Продовжити гру», «Налаштування», «Вихід», обробники всіх налаштувань та збереження всіх налаштувань.
- Minimax – тут описаний та реалізований агоритм «Мінімакс» для шахів, який і дозволяє «грати з комп’ютером».
- TileSelector – тут описані функції вибору фігури та показ можливих ходів шляхом активації синіх квадратів-індикаторів, які вказують на усі можливі ходи фігури.
- MoveSelector – тут описані основні методи для пересування раніше вибраної фігури, захоплення фігури противника та відміна ходу.
- Board – тут описані *основні* методи для створення та ініціалізації фігури, її знищення, пересування, вибору та його скасування.
- Geometry – тут описані функції для визначення точки за координатами, координат за точкою та, власне, створення сітки на якій все і відбуватиметься.
- Player – тут визначається гравець.
- CameraMove – тут описаний метод для повороту камери.

- Esc – тут описаний метод виходу з гри в меню та збереження прогресу гри.
- Piece – це основний батьківський базовий клас для всіх фігур. Тут визначені ходи по вертикалі, горизонталі та діагоналі, а також ствренний список з усіма фігурами. Його успадковують усі класи з фігурами, а саме: King, Queen, Rook, Knight, Bishop та Pawn. В усіх цих класах описуються ходи тої чи іншої фігури.

2.4.2 Функції класів MainMenu, Esc та CameraMove

Спочатку доречно було б розповісти про другорядні класи і вже потім оглядати головні. Тому в цьому та наступному розділах будуть описані усі функції та методи другорядних класів, без яких не було б головних.

Клас **MainMenu** дозволяє взаємодіяти, власне, з меню гри та дає змогу перемикатися між сценами гри.

Цей клас складається з таких методів:

- NewGame() – це метод-обробник кнопки «Нова гра». Дозволяє починати гру та перемикати сцену меню на сцену гри, а також записувати стан гри у текстовий файл (рис. 2.27).

```
public void NewGame()
{
    using (StreamWriter sw = new StreamWriter(@"C:\Game.txt"))
    {
        sw.WriteLine(0);
    }
    loadMyScene.Add(SceneManager.LoadSceneAsync(1));
    loadMyScene.Add(SceneManager.LoadSceneAsync(2, LoadSceneMode.Additive));
}
```

Рис. 2.27. Метод NewGame()

Тут файл Game.txt розташоватий саме на диску C умовно і в дійсності директорія, в яку зберігаються усі файли інша. У файл записується 0, бо він визначає стан гри (0 – нова гра, 1 – продовжити гру). Далі йде завантаження сцен гри.

- ContinueGame() – метод-обробник кнопки «Продовжити гру». Повністю ідентичний з NewGame(), але у файл Game.txt записується 1, а не 0 (рис. 2.28).

```
public void ContinueGame()
{
    using (StreamWriter sw = new StreamWriter(@"C:\Game.txt"))
    {
        sw.WriteLine(1);

    }
    loadMyScene.Add(SceneManager.LoadSceneAsync(1));
    loadMyScene.Add(SceneManager.LoadSceneAsync(2, LoadSceneMode.Additive));
}
```

Рис. 2.28. Метод ContinueGame()

Власне, продовжується гра у класі GameManager, який при запуску аналізує файл Game.txt.

- GameDifficulty() – метод-обробник випадного списку Difficulty Dropdown, що знаходиться в налаштуваннях. Записує значення value у файл GameDifficulty.txt (рис. 2.29).

```
public void GameDifficulty(int val)
{
    using (StreamWriter sw = new StreamWriter(@"C:\GameDifficulty.txt"))
    {
        sw.WriteLine(val);
        sw.Close();
    }
}
```

Рис. 2.29. Метод GameDifficulty()

- GameMode() – метод-обробник випадного списку GameModeDropdown, що знаходиться в налаштуваннях. Записує значення value у файл GameMode.txt (рис. 2.30).

```
public void GameMode(int val)
{
    using (StreamWriter sw = new StreamWriter(@"C:\GameMode.txt"))
    {
        sw.WriteLine(val);
        sw.Close();
    }
}
```

Рис. 2.30. Метод GameMode()

- ChangeTexture() – метод-обробник випадного списку TextureDropdown, що знаходиться в налаштуваннях. Записує значення value у файл GameTexture.txt (рис. 2.31).

```
public void ChangeTexture(int val)
{
    using (StreamWriter sw = new StreamWriter(@"C:\GameTexture.txt"))
    {
        sw.WriteLine(val);
        sw.Close();
    }
}
```

Рис. 2.31. Метод ChangeTexture()

- Exit() – медод-обробник кнопки «Вихід». Завершує процес програми та здійснює з неї вихід (рис. 2.32).

```
public void Exit()
{
    Debug.Log("Game over");
    Application.Quit();
}
```

Рис. 2.32. Метод Exit()

- SaveOptions() – метод, який відновлює стани налаштувань, виставляє вже раніше обрані та збережені значення в опції налаштувань (рис. 2.33).

```
public void SetOptions()
{
    TMP_Dropdown GameModeDropdown = GameObject.Find("GameModeDropdown").
                                                GetComponent<TMP_Dropdown>();
    using (StreamReader sr = new StreamReader(@"C:\GameModeCheck.txt"))
    {
        GameModeDropdown.value = Convert.ToInt32(sr.ReadLine());
        sr.Close();
    }
    TMP_Dropdown DifficultyDropdown = GameObject.Find("DifficultyDropdown").
                                                  GetComponent<TMP_Dropdown>();
    using (StreamReader sr = new StreamReader(@"C:\GameComplexCheck.txt"))
    {
        DifficultyDropdown.value = Convert.ToInt32(sr.ReadLine());;
        sr.Close();
    }
}
```

Рис. 2.33. Метод SaveOptions()

Існує кілька варіантів збереження налаштувань та іншої інформації у файлах. Вже відомий спосіб вписування значень налаштувань в окремі файли. Ще наприклад збереження налаштувань можна здійснити за допомогою PlayerPrefs [7, 165 ст]. Однак було прийнято рішення зберігати всю інформацію у вигляді окремих текстових файлів.

Клас **Esc** виконує важливу роль. Він дає змогу зберігати стан усієї гри у файл GameBoard.txt та здійснювати вихід у головне меню (рис. 2.34).

```
public class Ecs : MonoBehaviour {
    void Update() {
        if (Input.GetKeyDown(KeyCode.Escape)) {
            using (StreamWriter sw = new StreamWriter(@"C:\GameBoard.txt")) {
                sw.Write("");
                for (int i = 0; i < 8; i++) {
                    for (int j = 0; j < 8; j++) {
                        sw.Write(GameManager.instance.boardArr[i, j] + " ");
                    }
                    sw.WriteLine();
                }
                sw.WriteLine(GameManager.instance.WhiteScore.text);
                sw.WriteLine(GameManager.instance.BlackScore.text);
            }
            SceneManager.LoadScene(0);
        }
    }
}
```

Рис. 2.34. Клас Esc

Як можна помітити, тут зберігається вся дошка зі всіма фігурами, а також очки гравців у файлі GameBoard.txt.

Клас **CameraMove** дозволяє, власне, рухати камерою, бо інакше камера була б зафіксована в одному положенні, а це не дуже зручно (рис. 2.35).

```
public class CameraMove : MonoBehaviour
{
    [SerializeField] private Camera cam, Transform target;
    private Vector3 previousPosition;
    void Update() {
        if (Input.GetMouseButtonDown(1)) {
            previousPosition = cam.ScreenToViewportPoint(Input.mousePosition);
        }else if (Input.GetMouseButton(1)) {
            Vector3 direction = previousPosition - cam.ScreenToViewportPoint(Input.mousePosition);
            cam.transform.position = new Vector3();
            cam.transform.Rotate(new Vector3(1, 0, 0), direction.y * 360);
            cam.transform.Rotate(new Vector3(0, 1, 0), -direction.x * 360, Space.World);
            cam.transform.Translate(new Vector3(0, 0, -10));
            previousPosition = cam.ScreenToViewportPoint(Input.mousePosition);
        }
    }
}
```

Рис. 2.35. Клас CameMove

Також був варіант здійснювати поворот камери за допомогою структури Quaternion. Однак з нею були певні незручності, тому зрештою був використаний вище вказаний спосіб. Звісно ж є ще кілька способів повороту камери у тривимірному просторі, але вище вказаний спосіб є найзручнішим та найстабільнішим способом.

2.4.3 Функції класів Player, Geometry та Board

У класі **Player** описані характеристики гравця, а також створені списки активних та захоплених фігур (рис. 2.36).

```
public class Player
{
    public List<GameObject> pieces, capturedPieces;
    public string name;
    public int forward;
    public Player(string name, bool positiveZMovement) {
        this.name = name;
        pieces = new List<GameObject>();
        capturedPieces = new List<GameObject>();
        if (positiveZMovement == true) this.forward = 1;
        else this.forward = -1;
    }
}
```

Рис. 2.36. Клас Player

У класі **Geometry** описані функції для визначення точки за координатами, координат за точкою та, власне, створення сітки на якій все і відбуватиметься. Він є швидше допоміжним класом, ніж обов'язковим (рис 2.37).

```
public class Geometry
{
    static public Vector3 PointFromGrid(Vector2Int gridPoint) {
        float x = -3.5f + 1.0f * gridPoint.x;
        float z = -3.5f + 1.0f * gridPoint.y;
        return new Vector3(x, 0, z);
    }
    static public Vector2Int GridPoint(int col, int row) => new Vector2Int(col, row);
    static public Vector2Int GridFromPoint(Vector3 point) {
        int col = Mathf.FloorToInt(4.0f + point.x);
        int row = Mathf.FloorToInt(4.0f + point.z);
        return new Vector2Int(col, row);
    }
}
```

Рис. 2.37. Клас Geometry

У класі **Board**, як було зазначено трохи вище, описані *основні* методи для створення та ініціалізації фігури, її знищення, пересування, вибору та його скасування (рис 2.38).

```
public class Board : MonoBehaviour
{
    public Material defaultMaterial, selectedMaterial;

    public GameObject AddPiece(GameObject piece, int col, int row) {
        Vector2Int gridPoint = Geometry.GridPoint(col, row);
        GameObject newPiece = Instantiate(piece, Geometry.PointFromGrid(gridPoint), Quaternion.identity, gameObject.transform);
        return newPiece;
    }
    public void RemovePiece(GameObject piece) {
        Destroy(piece);
        if (GameManager.instance.CurrPlayer() == "white") {
            int whiteScore = int.Parse(GameManager.instance.WhiteScore.text.Remove(0, 7));
            whiteScore += 1;
            GameManager.instance.WhiteScore.text = "White: " + whiteScore;
        } else {
            int blackScore = int.Parse(GameManager.instance.BlackScore.text.Remove(0, 7));
            blackScore += 1;
            GameManager.instance.BlackScore.text = "Black: " + blackScore;
        }
    }
    public void MovePiece(GameObject piece, Vector2Int gridPoint) {
        piece.transform.position = Geometry.PointFromGrid(gridPoint);
    }
    public void SelectPiece(GameObject piece) {
        MeshRenderer renderers = piece.GetComponentInChildren<MeshRenderer>();
        renderers.material = selectedMaterial;
    }
    public void DeselectPiece(GameObject piece) {
        MeshRenderer renderers = piece.GetComponentInChildren<MeshRenderer>();
        renderers.material = defaultMaterial;
    }
}
```

Рис. 2.38. Клас Board

Саме за допомогою цього класу модельки фігур виставляються насцену гри – на тривимірну шахівницю.

2.4.4 Функції класів Piece, King, Queen, Rook, Knight, Bishop та Pawn

Усі ці класи описують усі можливі ходи кожної з фігур. Кожен метод MoveLocations() приймає координати своєї фігури і на основі цього створює список з координат, на які може піти фігура. Також у класі Piece, що є базовим для всіх інших класів фігур, створено список Enum з назвами всіх фігур і визначено напрямки ходьби, а саме по горизонталі, по вертикалі та по діагоналям.

Клас **Piece** (рис. 2.38):

```
public enum PieceType {King, Queen, Bishop, Knight, Rook, Pawn};
public abstract class Piece : MonoBehaviour
{
    public PieceType type;
    protected Vector2Int[] RookDirections = {new Vector2Int(0,1), new Vector2Int(1, 0), new Vector2Int(0, -1), new Vector2Int(-1, 0)};
    protected Vector2Int[] BishopDirections = {new Vector2Int(1,1), new Vector2Int(1, -1), new Vector2Int(-1, -1), new Vector2Int(-1, 1)};
    public abstract List<Vector2Int> MoveLocations(Vector2Int gridPoint);
}
```

Рис. 2.38. Клас Piece

Клас **Knight** (рис. 2.39):

```
public class Knight : Piece {
    public override List<Vector2Int> MoveLocations(Vector2Int gridPoint) {
        List<Vector2Int> locations = new List<Vector2Int>();
        locations.Add(new Vector2Int(gridPoint.x - 1, gridPoint.y + 2));
        locations.Add(new Vector2Int(gridPoint.x + 1, gridPoint.y + 2));
        locations.Add(new Vector2Int(gridPoint.x + 2, gridPoint.y + 1));
        locations.Add(new Vector2Int(gridPoint.x - 2, gridPoint.y + 1));
        locations.Add(new Vector2Int(gridPoint.x + 2, gridPoint.y - 1));
        locations.Add(new Vector2Int(gridPoint.x - 2, gridPoint.y - 1));
        locations.Add(new Vector2Int(gridPoint.x + 1, gridPoint.y - 2));
        locations.Add(new Vector2Int(gridPoint.x - 1, gridPoint.y - 2));
        return locations;
    }
}
```

Рис. 2.39. Клас Knight

Клас **Pawn** (рис. 2.40):

```
public class Pawn : Piece {
    public override List<Vector2Int> MoveLocations(Vector2Int gridPoint) {
        List<Vector2Int> locations = new List<Vector2Int>();
        int forwardDirection = GameManager.instance.currentPlayer.forward;
        Vector2Int forwardOne = new Vector2Int(gridPoint.x, gridPoint.y + forwardDirection);
        if (GameManager.instance.PieceAtGrid(forwardOne) == false) locations.Add(forwardOne);
        Vector2Int forwardTwo = new Vector2Int(gridPoint.x, gridPoint.y + 2 * forwardDirection);
        if (GameManager.instance.HasPawnMoved(gameObject) == false &&
            GameManager.instance.PieceAtGrid(forwardTwo) == false) locations.Add(forwardTwo);
        Vector2Int forwardRight = new Vector2Int(gridPoint.x + 1, gridPoint.y + forwardDirection);
        if (GameManager.instance.PieceAtGrid(forwardRight)) locations.Add(forwardRight);
        Vector2Int forwardLeft = new Vector2Int(gridPoint.x - 1, gridPoint.y + forwardDirection);
        if (GameManager.instance.PieceAtGrid(forwardLeft)) locations.Add(forwardLeft);
        return locations;
    }
}
```

Рис. 2.40. Клас Pawn

У класах **King** та **Queen** просто перечисляються в циклі усі ходи в усіх напрямках, тому вони, можна сказати, абсолютно однакові. Різниця хіба що в тому, що в класі King перечисляються ходи лише на одну клітинку вперед в усіх напрямках, тоді як у класі Queen ходи визначаються в межах усієї дошки в усіх напрямках.

Класи **Rook** та **Bishop** теж ідентичні. Теж через цикл перечисляються усі можливі ходи та визначаються в межах дошки. Різниця в напрямках ходів. Rook використовує RookDirection для визначення напрямків, тобто ходи відбуватимуться по горизонталі й вертикалі. Bishop натомість використовує BishopDirection для визначення напрямків ходів, які відбуватимуться по діагоналях.

Клас **Knight** визначає ходи для Коня на основі його позиції на дошці. Цикл використовувати нема сенсу, бо усіх ходів існує всього 8, тож вони були просто перечислені.

Клас **Pawn** визначає усі можливі ходи певного Пішака. Тобто, найперший хід може бути як на одну клітинку вперед, так і на дві. Далі пішак рухатиметься строго на одну клітинку вперед, тільки якщо не буде захоплючати інші фігури, тобі ходитиме на одну клітинку по діагоналі.

Тут визначаються загальні ходи фігур. Наприклад, якщо це буде Королева, то вона зможе ходити на 8 клітинок у всіх напрямках. Але в такому разі існує ймовірність виходу за межі дошки. Цю проблему вирішує метод MovesForPiece(), що знаходиться в класі GameManager. Це метод був представлений та описаний у підрозділі 2.4.5.

2.4.5 Функції класу GameManager

Як було зазначено кількома сторінками раніше, це найбільший та найголовніший клас, який об’єднує роботу всіх інших класів.

Опочатку створюються екземпляри усіх необхідних на даному етапі складових гри (рис. 2.41).

```
public static GameManager instance;
public Board board;
public GameObject whiteKing, whiteQueen, whiteBishop, whiteKnight, whiteRook, whitePawn;
public GameObject blackKing, blackQueen, blackBishop, blackKnight, blackRook, blackPawn;
public GameObject[,] pieces;
private List<GameObject> movedPawns;
private static Player white, black;
public Player currentPlayer, otherPlayer;
public Text WhiteScore, BlackScore, Win, CCPlayer;
```

Рис. 2.41. Екземпляри складових гри

Це всі поля, які потрібні для роботи класу. Далі йде низка методів, а саме:

- Start() – тут ініціалізуються всі поля класу, окрім фігур, їх ініціалізувати нема потреби.

```
void Start() {
    pieces = new GameObject[8, 8];
    movedPawns = new List<GameObject>();
    white = new Player("White", true);
    black = new Player("Black", false);
    GameObject CPlayer = GameObject.Find("CurrentPlayer");
    currentPlayer = white;
    otherPlayer = black;
    CCPlayer.text = "Хід білих!!!"; ;
    WhiteScore = "Білі:    0";
    BlackScore = "Чорні: 0";
    InitialSetup();
}
```

Рис. 2.41. Екземпляри складових гри

- InitialSetup() – дуже важливий метод, можна навіть сказати найважливіший на даному етапі, адже він в залежності від вибраної дії гравця (гравець починає нову гру чи продовжує вже наявну) виставляє фігури на дошку.

Спочатку перевіряється файл Game.txt (рис. 2.42).

```
using (StreamReader sr = new StreamReader(@"C:\Game.txt"))
{
    game = Convert.ToInt32(sr.ReadToEnd());
}
```

Рис. 2.42. Перевірка файлу Game.txt

І вже на основі збереженого значення визначається спосіб виставлення фігур (рис. 2.43, 2.44).

```
if (game == 0) {
    AddPiece(whiteRook, whiteKnight, whiteBishop, whiteQueen, whiteKing, whiteBishop, whiteKnight, whiteRook);
    for (int i = 0; i < 8; i++) AddPiece(whitePawn);
    AddPiece(blackRook, blackKnight, blackBishop, blackQueen, blackKing, blackBishop, blackKnight, blackRook);
    for (int i = 0; i < 8; i++) AddPiece(blackPawn);
    boardArr = new int[8, 8] {
    {50,  10, 0, 0, 0, 0, -10, -50}, {35,  10, 0, 0, 0, 0, -10, -35}, {30,  10, 0, 0, 0, 0, -10, -30},
    {90,  10, 0, 0, 0, 0, -10, -90}, {900, 10, 0, 0, 0, 0, -10, -900}, {30,  10, 0, 0, 0, 0, -10, -30},
    {35,  10, 0, 0, 0, 0, -10, -35}, {50,  10, 0, 0, 0, 0, -10, -50}};
}
```

Рис. 2.43. Виставлення фігур при значенні 0 (Нова гра)

```
else if (game == 1) {
    using (StreamReader sr = new StreamReader(@"C:\GameBoard.txt")) {
        for (int i = 0; i < 8; i++) {
            string[] arr = sr.ReadLine().Split(' ', '\n');
            for (int j = 0; j < 8; j++) {
                boardArr[i, j] = Convert.ToInt32(arr[j]);
                switch (boardArr[i, j]) {
                    case 10: AddPiece(whitePawn, white, i, j); break;
                    case 50: AddPiece(whiteRook, white, i, j); break;
                    case 35: AddPiece(whiteKnight, white, i, j); break;
                    case 30: AddPiece(whiteBishop, white, i, j); break;
                    case 90: AddPiece(whiteQueen, white, i, j); break;
                    case 900: AddPiece(whiteKing, white, i, j); break;
                    case -10: AddPiece(blackPawn, black, i, j); break;
                    case -50: AddPiece(blackRook, black, i, j); break;
                    case -35: AddPiece(blackKnight, black, i, j); break;
                    case -30: AddPiece(blackBishop, black, i, j); break;
                    case -90: AddPiece(blackQueen, black, i, j); break;
                    case -900: AddPiece(blackKing, black, i, j); break;
                }
            }
        }
        tempPieces = pieces;
        GameManager.instance.WhiteScore.text = sr.ReadLine();
        GameManager.instance.BlackScore.text = sr.ReadLine();
    }
}
```

Рис. 2.44. Виставлення фігур при значенні 1 (Продовжити гру)

При стані нової гри (значення 0) фігури просто виставляються на дошку в початкових позиціях. Також заповнюється двовимірний масив BoardArr. Він заповнюється числами, які репрезентують очки за захоплення цих самих фігур. Від'ємні числа використовується тільки, щоб відрізняти чорні фігури від білих. Таким чином: 10 – пішак, 30 – слон, 35 – кінь, 50 – тура, 90 – королева, 900 – король.

При стані продовження вже наявної гри (значення 1) зчитується файл GameBoard.txt та вже на основі отриманих даних виставляються фігури на дошку у позиціях, які були раніше збережені, а також знову ініціалізуються текстові об'єкти WhiteScore та BlackScore для відновлення очок.

- ChangeTexture() – метод, задача якого є змінювати текстуру фігур. При його виклику у методі Start() відбувається зчитування файлу GameTexture.txt і

вже на основі збереженого значення опції, текстури фігур змінюються на вибрані (рис. 2.45).

```
public void ChangeTexture()
{
    using (StreamReader sr = new StreamReader(@"C:\GameTexture.txt")) => int texture = Convert.ToInt32(sr.ReadLine());
    Material standartMaterial = AssetDatabase.
    LoadAssetAtPath<Material>(" ChessTexture.mat");
    Material redMaterial = AssetDatabase.
    LoadAssetAtPath<Material>(" /ChessTexture2.mat");

    GameObject whiteQueen = GameObject.Find("QueenWhite(Clone)");
    .......
    if (texture == 0) {
        board.AddPiece(whiteRook, 0, 0).transform.Find("RookWhite(Clone)").
        GetComponentInChildren<Renderer>().material = standartMaterial;
        ........


        for (int i = 0; i < 8; i++) {
           board.AddPiece(whitePawn, i, 1).transform.Find("PawnWhite").
           GetComponentInChildren<Renderer>().material = standartMaterial;
        }
        ........

    }
    if (texture == 1) {
        board.AddPiece(whiteRook, 0, 0).transform.Find("RookWhite").
        GetComponentInChildren<Renderer>().material = redMaterial;
        ........


        for (int i = 0; i < 8; i++) {
            board.AddPiece(whitePawn, i, 1).transform.Find("PawnWhite").
            GetComponentInChildren<Renderer>().material = redMaterial;
        }
        ........
    }
}
```

Рис. 2.45. Зміна текстура

Наші фігури складаються з, так би мовити, двох шарів. З обгортки та основної фігури. І тому щоб дістатися до основної фігури було використано метод GetComponentInChildren<>() задля отримання доступу до дочірнього об’єкту.

- AllPossibleMoves() – метод, який створює список усіх можливих та коректних ходів для усіх наявних на дошці фігур (рис. 2.46).

```
public List<Minimax.Move> AllPossibleMoves() {
    List<Minimax.Move> allPieceMoves = new List<Minimax.Move> { };
    List<Vector2Int> moves;
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            if (GameManager.instance.boardArr[j, i] < 0) {
                moves = MovesForPiece(pieces[j, i]);
                if (moves.Count > 0) {
                    foreach (var item in moves) {
                        Minimax.Move move = new Minimax.Move(j, i, item.x, item.y);
                        allPieceMoves.Add(move);
                    }}}}}
    return allPieceMoves;
}
```

Рис. 2.46. Визначення усіх можливих ходів усіх фігур

Цей метод відноситься до класу Minimax, але він задля зручності був створений саме в класі GameManager. Детальний опис класу Minimax представлено в підрозділі 2.4.8.

- AddPiece() – це метод, власне, додає фігуру на раніше створену та ініціалізовану дошку pieces (рис. 2.47).

```
public void AddPiece(GameObject prefab, Player player, int col, int row) {
    player.pieces.Add(board.AddPiece(prefab, col, row));
    pieces[col, row] = board.AddPiece(prefab, col, row);
}
```

Рис. 2.47. Додавання фігури

Саме додавання фігури на сцену гри відбувається за допомогою однойменного методу з класу Board.

- MovesForPiece() – цей метод визначає усі можливі ходи для певної фігури, перевіряє їх на коректність та видаляє некоректні ходи (ті, що виходять за межі дошки), а також видаляє ходи, які відбуватимуться на клітинах, на яких вже стоять дружні фігури (рис. 2.48).

```
public List<Vector2Int> MovesForPiece(GameObject pieceObject) {
    Vector2Int gridPoint = GridForPiece(pieceObject);
    List<Vector2Int> locations = pieceObject.GetComponent<Piece>().MoveLocations(gridPoint);
    locations.RemoveAll(gp => gp.x < 0 || gp.x > 7 || gp.y < 0 || gp.y > 7);
    locations.RemoveAll(gp => FriendlyPieceAt(gp));
    return locations;
}
```

Рис. 2.48. Визначення усіх можливих ходів для певної фігури

- Move() – цей метод виконує, власне, сам хід. Здійснює переміщення фігури (рис. 2.49).

```
public void Move(GameObject piece, Vector2Int gridPoint) {
    if (piece.GetComponent<Piece>().type == PieceType.Pawn && !HasPawnMoved(piece)) movedPawns.Add(piece);
    Vector2Int startGridPoint = GridForPiece(piece);
    pieces[startGridPoint.x, startGridPoint.y] = null; pieces[gridPoint.x, gridPoint.y] = piece;
    int score = boardArr[startGridPoint.x, startGridPoint.y];
    boardArr[startGridPoint.x, startGridPoint.y] = 0; boardArr[gridPoint.x, gridPoint.y] = score;
    board.MovePiece(piece, gridPoint);
}
```

Рис. 2.49. Виконання ходу

Тут, так само, як у методі AddPiece переміщення відбувається на дошці pieces. Саме переміщення фігури на сцені здійснює метод MovePiece() з класу Board.

- PawnMoved() та HasPawnMoved() – допоміжні методи (рис. 2.50).

PawnMoved() у разі переміщення котрогось з пішаків додає їх у список переміщених пішаків movedPawns.

HasPawnMoved() перевіряє чи певний пішак вже переміщувався шляхом аналізу списку movedPawns.

```
public void PawnMoved(GameObject pawn) => movedPawns.Add(pawn);

public bool HasPawnMoved(GameObject pawn) => return movedPawns.Contains(pawn);
```

Рис. 2.50. Допоміжні методи

- CapturePieceAt() – це метод, основне завдання якого - полягає в захопленні ворожої фігури. Також він керує текстовими об’єктами WhiteScore та BlackScore, а також об’єктом Win тільки у разі стану перемоги (коли був захоплений ворожий король) (рис. 2.51).

```
public void CapturePieceAt(Vector2Int gridPoint)
{
    if (GameManager.instance.CurrPlayer() == "White") => GameManager.instance.WhiteScore.text = "Білі:" + whiteScore;
    else GameManager.instance.BlackScore.text = "Чорні: " + blackScore;
    GameObject pieceToCapture = PieceAtGrid(gridPoint);
    if (pieceToCapture.GetComponent<Piece>().type == PieceType.King) {
        Win = GameObject.Find("WinMassage");.GetComponent<Text>();
        if (currentPlayer.name == "White") Win.text = "Білі перемогли!!!";
        else Win.text = "Чорні перемогли!!!";
        Destroy(board.GetComponent<TileSelector>()); Destroy(board.GetComponent<MoveSelector>());
    }
    currentPlayer.capturedPieces.Add(pieceToCapture);
    pieces[gridPoint.x, gridPoint.y] = null; boardArr[gridPoint.x, gridPoint.y] = 0;
    Destroy(pieceToCapture);
}
```

Рис. 2.51. Захоплення фігури

- SelectPiece() та DeselectPiece() – методи, які слугують для виділення певної фігури, та для скасування цього самого виділення (рис. 2.52).

```
public void SelectPiece(GameObject piece) => board.SelectPiece(piece);

public void DeselectPiece(GameObject piece) => board.DeselectPiece(piece);
```

Рис. 2.52. Захоплення фігури

Ці методі працюють завдяки наведених у підрозділі 2.4.3 однойменних методів класу Board.

- DoesPieceBelongToCurrent Player() – метод, який перевіряє чи належить фігура поточному гравцеві (рис. 2.53).

```
public bool DoesPieceBelongToCurrentPlayer(GameObject piece) => return currentPlayer.pieces.Contains(piece);
```

Рис. 2.52. Перевірка фігури

- PieceAtGrid() та GridForPiece() – допоміжні методі, які повертають фігуру за координатами, та координати за фігурою (рис. 2.53).

```
public GameObject PieceAtGrid(Vector2Int gridPoint) {
    if (gridPoint.x > 7 || gridPoint.y > 7 || gridPoint.x < 0 || gridPoint.y < 0) return null;
    return pieces[gridPoint.x, gridPoint.y];
}

public Vector2Int GridForPiece(GameObject piece) {
    for (int i = 0; i < 8; i++)
        for (int j = 0; j < 8; j++)
            if (pieces[i, j] == piece)
                return new Vector2Int(i, j);
    return new Vector2Int(-1, -1);
}
```

Рис. 2.53. Допоміжні методи

- FriendlyPieceAt() – метод, який перевіряє чи немає на певній клітинці дружньої фігури (рис. 2.54).

```
public bool FriendlyPieceAt(Vector2Int gridPoint) {
    GameObject piece = PieceAtGrid(gridPoint);
    if (piece == null) return false;
    if (otherPlayer.pieces.Contains(piece)) return false;
    return true;
}
```

Рис. 2.54. Перевірка фігури

- CurrPlayer() та NextPlayer() – методи, робота яких пов’язана з гравцем (рис. 2.55).

```
public string CurrPlayer() => return currentPlayer.name;

public void NextPlayer() {
    Player tempPlayer = currentPlayer;
    currentPlayer = otherPlayer;
    otherPlayer = tempPlayer;
    if (currentPlayer.name == "White") CCPlayer.text = "Хід білих!!!";
    else CCPlayer.text = "Хід чорних!!!";
}
```

Рис. 2.55. Зміна гравця

CurrPlayer() повертає ім’я поточного гравця. NextPlayer() визначає наступного гравця та керує текстовими об’єктом CCPlayer, який під час гри повідомляє, чий наразі хід.

2.4.6 Функції класу TileSelector

Функції класу TileSelector відповідають за вибір та виділення фігури, а також показує усі можливі та коректні ходи цієї фігури. Саме тут починається використання алгоритму «Мінімакс» - алгоритм вибирає фігуру з найкращим ходом та передає її в клас MoveSelector.

У класі TileSelector реалізовані такі методи:

- EnterState() – це метод за допомогою властивості enabled вмикає об’єкт даного класу, так званий компонент. Після цього об’єкт класу стає активним (2.56).

```
public void EnterState() => enabled = true;
```

Рис. 2.56. Активація об’єкту

- Start() – цей метод проводить початкову ініціалізацію сітки, на якій будуть виводитися ходи, об’єкта, який репрезентує найкращий хід фігури та

```
void Start ()
{
    Vector3 point = Geometry.PointFromGrid(Geometry.GridPoint(0, 0));
    tileHighlight.SetActive(false);
}
```

Рис. 2.57. Перевірка фігури

тайла-індикатора, який буде слугувати підсвіткою самої фігури та її ходів, а також фігур, яких можна побити (рис. 2.57).

- Update() – це основний метод даного класу. Він виконує аналіз файлу GameMode.txt і на основі збереженого там значення виконує різні інструкції (рис. 2.58).

```
using (StreamReader sr = new StreamReader(@"C:\GameMode.txt"))
{
    GameManager.instance.GameMode = Convert.ToInt32(sr.ReadToEnd());
}
```

Рис. 2.58. Аналіз файлу GameMode.txt

Якщо значення GameMode дорівнює 0, що означає гру з людиною, то виконується стандартні інструкції, які реалізують вибір фігури та її виділення жовтим тайлом-індикатором. Також після вибору фігури відбувається виклик кінцевого *першого* методу ExitState() (рис. 2.59).

```
if (GameMode == 0)
{
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition); RaycastHit hit;
    if (Physics.Raycast(ray, out hit)) {
        Vector3 point = hit.point; Vector2Int gridPoint = Geometry.GridFromPoint(point);
        tileHighlight.SetActive(true);
        if ((gridPoint.x >= 0 && gridPoint.x <= 7) && (gridPoint.y >= 0 && gridPoint.y <= 7))
            tileHighlight.transform.position = Geometry.PointFromGrid(gridPoint);
        else tileHighlight.SetActive(false);
        if (Input.GetMouseButtonDown(0)) {
            GameObject selectedPiece = GameManager.instance.PieceAtGrid(gridPoint);
            if (GameManager.instance.DoesPieceBelongToCurrentPlayer(selectedPiece)) {
                GameManager.instance.SelectPiece(selectedPiece);
                ExitState(selectedPiece);
            }}}
    else tileHighlight.SetActive(false);
}
```

Рис. 2.59. Дії при значенні 0

Якщо значення GameMode дорівнює 1, що визначає гру з комп’ютером, то виконується перевірка гравця (рис. 2.60).

```
else if (GameMode == 1) {
    if (GameManager.instance.CurrPlayer() == "White") {
        Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition); RaycastHit hit;
        if (Physics.Raycast(ray, out hit)) {
            Vector3 point = hit.point; Vector2Int gridPoint = Geometry.GridFromPoint(point);
            tileHighlight.SetActive(true);
            if ((gridPoint.x >= 0 && gridPoint.x <= 7) && (gridPoint.y >= 0 && gridPoint.y <= 7))
                tileHighlight.transform.position = Geometry.PointFromGrid(gridPoint);
            else tileHighlight.SetActive(false);
            if (Input.GetMouseButtonDown(0)) {
                GameObject selectedPiece = GameManager.instance.PieceAtGrid(gridPoint);
                if (GameManager.instance.DoesPieceBelongToCurrentPlayer(selectedPiece)) {
                    GameManager.instance.SelectPiece(selectedPiece);
                    ExitState(selectedPiece);
                }}}
        else tileHighlight.SetActive(false);
    } else if (GameManager.instance.CurrPlayer() == "Black") {
        Minimax.GameState state = new Minimax.
                GameState(GameManager.instance.boardArr,GameManager.instance.AllPossibleMoves());
        using (StreamReader sr = new StreamReader(@"C:\GameDifficulty.txt"))
            int depth = (Convert.ToInt32(sr.ReadToEnd()) == 0) ? 3 : 5;
        bestMove = Minimax.FindBestMove(state, depth);
        ExitState(bestMove);
    }
}
```

Рис. 2.60. Дії при значенні 1

Якщо поточним гравцем є біла сторона, тобто це ми, то інструкції нічим не відрізняються від звичайних. Але якщо поточним гравцем є чорна сторона, тобто наш противник, то починається робота алгоритму «Мінімакс», який шукає фігуру з найкращим ходом, та відправляє дані ходу в *другий* метод ExitState(). Робота алгоритму «Мінімакс» детально описується в підрозділі 2.4.8.

- ExitState(piece) та ExitState(move) – кінцеві методи, які вимикають поточний компонент класу, вимикають підсвітку та відправляють дані в клас MoveSelector для подальшої роботи.

Чому цих методів два? Тому що перший метод викликають стандартні інструкції, а другий викликають інструкції алгоритму «Мінімакс». Простими словами, перших метод приймає фігуру, а другий приймає хід, визначений алгоритмом (рис. 2.61).

```
private void ExitState(GameObject movingPiece) {
    this.enabled = false; tileHighlight.SetActive(false);
    GetComponent<MoveSelector>();.EnterState(movingPiece);
}

private void ExitState(Minimax.Move bestMove) {
    this.enabled = false;tileHighlight.SetActive(false);
    GetComponent<MoveSelector>();.EnterState(bestMove);
}
```

Рис. 2.61. Кінцеві методи

Обидва методи відправляють дані в *методи* EnterState() класу MoveSelector.

2.4.7 Функції класу MoveSelector

Функції цього класу полягають у виборі та здійсненні ходу. В залежності від відісланих значень методів ExitState() класу TileSelector виконуватимуться ті чи інші інструкції.

У класі MoveSelector реалізовано такі методи:

- EnterState(piece) та EnterState(move) – початкові методи, за допомогою яких вмикається компонент даного класу, ініціалізуються об’єкти тайлів-індикаторів, аналізується файл GameMode.txt та у разі необхідності відбувається скасування ходу з подальшим перенаправленням роботи в клас TileSelector задля повторного вибору фігури.

Отож, у першому методі спочатку в залежності від збереженого раніше значення у файлі GameMode.txt відбувається виконання різних інструкцій (рис. 2.62).

```
using (StreamReader sr = new StreamReader(@"C:\GameMode.txt"))
{
    int GameMode = Convert.ToInt32(sr.ReadToEnd());
}
```

Рис. 2.62. Аналіз файлу GameMode.txt

Якщо значення GameMode дорівнюють 0, то виконуються стандартні інструкції, які виконують: увімкнення компонента даного класу, ініціалізацію фігури, яка буде пересуватися, активацію синіх тайлів-індикаторів, які підсвічують ходи фігури та у разі необхідності відбувається дозвіл на скасування ходу (рис. 2.63).

```
if (GameManager.instance.GameMode == 0) {
    movingPiece = piece;this.enabled = true;
    moveLocations = GameManager.instance.MovesForPiece(movingPiece);
    locationHighlights = new List<GameObject>();
    if (moveLocations.Count == 0) CancelMove();
    foreach (Vector2Int loc in moveLocations) {
        GameObject highlight;
        if (GameManager.instance.PieceAtGrid(loc))
            highlight = Instantiate(attackLocationPrefab, Geometry.PointFromGrid(loc),
                Quaternion.identity, gameObject.transform);
        else
            highlight = Instantiate(moveLocationPrefab, Geometry.PointFromGrid(loc),
                Quaternion.identity, gameObject.transform);
        locationHighlights.Add(highlight);
    }
}
```

Рис. 2.63. Дії при значенні 0

У разі, коли значення GameMode дорівнює 1 – виконуються інструкції, які полягають у додатковій перевірці гравця. Якщо гравцем є біла сторона, то виконуються стандартні інструкції. Якщо ж гравцем є чорна сторона, то

відбувається проста ініціалізація фігури, яка пересуватиметься, та активація компоненту класу (рис. 2.64).

```
else if (GameManager.instance.GameMode == 1) {
    if (GameManager.instance.CurrPlayer() == "White") {
        movingPiece = piece; this.enabled = true;
        moveLocations = GameManager.instance.MovesForPiece(movingPiece);
        locationHighlights = new List<GameObject>();
        if (moveLocations.Count == 0) CancelMove();
        foreach (Vector2Int loc in moveLocations) {
            GameObject highlight;
            if (GameManager.instance.PieceAtGrid(loc))
                highlight = Instantiate(attackLocationPrefab, Geometry.PointFromGrid(loc),
                    Quaternion.identity, gameObject.transform);
            else
                highlight = Instantiate(moveLocationPrefab, Geometry.PointFromGrid(loc),
                    Quaternion.identity, gameObject.transform);
            locationHighlights.Add(highlight);
        }
    }
    else if (GameManager.instance.CurrPlayer() == "Black") {
        movingPiece = piece; this.enabled = true;
    }
}
```

Рис. 2.64. Дії при значенні 1

У другому ж методі, який викликається при грі з комп'ютером, відбувається ініціалізація ходу, фігури, яка пересуватиметься, та проводиться активація компонента даного класу (рис. 2.65).

```
public void EnterState(Minimax.Move move) {
    bestMove = move;
    movingPiece = GameManager.instance.MovingPiece(bestMove.fromX, bestMove.fromY);
    this.enabled = true;
}
```

Рис. 2.65. Активація компонента

- Start() – тут проводиться початкова ініціалізація тайлів-індикаторів, тобто підсвітки (2.66).

```
void Start ()
{
    this.enabled = false;
    tileHighlight = Instantiate(tileHighlightPrefab, Geometry.PointFromGrid(new Vector2Int(0, 0)),
        Quaternion.identity, gameObject.transform);
    tileHighlight.SetActive(false);
}
```

Рис. 2.66. Ініціалізація індикаторів

- Update() – основний метод класу. Тут, власне, й відбувається пересування фігури.

Спочатку аналізується файл GameMode.txt і вже на основі збереженого раніше значення здійснюються різного роду інструкції (рис. 2.67).

```
using (StreamReader sr = new StreamReader(@"C:\GameMode.txt"))
{
    int GameMode = Convert.ToInt32(sr.ReadToEnd());
}
```

Рис. 2.67. Аналіз файлу GameMode.txt

Якщо значення GameMode дорівнює 0, то виконуються стандартні інструкції: активується підсвітка ходів, виконується вибір ходу та, власне, здійснюється переміщення фігури, а в разі необхідності й захоплення ворожої фігури. Також можливе скасування ходу фігури(рис. 2.68).

```
if (GameMode == 0) {
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition); RaycastHit hit;
    if (Physics.Raycast(ray, out hit)) {
        Vector3 point = hit.point; Vector2Int gridPoint = Geometry.GridFromPoint(point);
        tileHighlight.SetActive(true); tileHighlight.transform.position = Geometry.PointFromGrid(gridPoint);
        if (Input.GetMouseButtonDown(0)) {
            if (Input.GetMouseButtonDown(0)) CancelMove();
            if (!moveLocations.Contains(gridPoint)) return;
            if (GameManager.instance.PieceAtGrid(gridPoint) == null)
                GameManager.instance.Move(movingPiece, gridPoint);
            else {
                GameManager.instance.CapturePieceAt(gridPoint);
                GameManager.instance.Move(movingPiece, gridPoint);
            }
            ExitState();
        }
    } else tileHighlight.SetActive(false);
}
```

Рис. 2.68. Дії при значенні 0

У разі, коли значення GameMode дорівнює 1, то проводиться додаткова перевірка гравця. Якщо гравцем є білі, то виконуються стандартні інструкції. Якщо ж гравцем є чорні, тобто ворожі фігури, то виконується ініціалізація координат на основі фігури з найкращим ходом, та подальше переміщення цієї фігури (рис. 2.69).

```
else if (GameMode == 1) {
    if (GameManager.instance.CurrPlayer() == "White") {
        movingPiece = piece; this.enabled = true;
        moveLocations = GameManager.instance.MovesForPiece(movingPiece);
        locationHighlights = new List<GameObject>();
        if (moveLocations.Count == 0) CancelMove();
        foreach (Vector2Int loc in moveLocations) {
            GameObject highlight;
            if (GameManager.instance.PieceAtGrid(loc))
                highlight = Instantiate(attackLocationPrefab, Geometry.PointFromGrid(loc),
                    Quaternion.identity, gameObject.transform);
            else
                highlight = Instantiate(moveLocationPrefab, Geometry.PointFromGrid(loc),
                    Quaternion.identity, gameObject.transform);
            locationHighlights.Add(highlight);
        }
    } else if (GameManager.instance.CurrPlayer() == "Black") {
        movingPiece = piece; this.enabled = true;
    }
}
```

Рис. 2.69. Дії при значенні 1

В кінці кожного броку з інструкціями відбувається виклик кінцевого методу ExitState().

- CancelMove() – метод, який відповідає за скасування ходу (рис. 2.70).

```
private void CancelMove() {
    this.enabled = false;
    foreach (GameObject highlight in locationHighlights) Destroy(highlight);
    GameManager.instance.DeselectPiece(movingPiece);
    TileSelector selector = GetComponent<TileSelector>();
    selector.EnterState();
}
```

Рис. 2.70. Скасування ходу

У разі скасування ходу відбувається активація компонента класу TileSelector задля повторно вибору фігури та ходу.

- ExitState() – кінцевий метод даного класу(рис. 2.71).

```
private void ExitState() {
    this.enabled = false; tileHighlight.SetActive(false);
    GameManager.instance.DeselectPiece(movingPiece); movingPiece = null;
    GameManager.instance.NextPlayer();
    GetComponent<TileSelector>();.EnterState();
    foreach (GameObject highlight in locationHighlights) Destroy(highlight);
}
```

Рис. 2.71. Кінцевий метод

Тут відбувається деактивація усіх використаних об’єктів, в тому числі підсвітки, перемикання на іншого гравця та активація компонента класу TileSelector.

2.4.8 Функції класу Minimax

Алгоритм «Мінімакс» є досить простим, якщо порівнювати його з іншими відомими алгоритмами. Але він дуже універсальний. Різні його варіації, як от наприклад, «Максимін», «Діпмакс», «Альфа-бета відсічення» чудово працюють для інших ігор та програм та мають багато власних переваг. Однак разом з цим з'являються декілька побічних помилок, для кожного алгоритму вони свої (велике навантаження через рекурсивність, складність підлаштовування до своєї гри, або підвищена неточність). Але для нескладних та неглобальних проєктів алгоритм «Мінімакс» чудово підходить.

Також були варіанти використання вже готових та навчених шахових рушіїв. Наприклад Stockfish, Komodo, Leela Chess Zero, Ethereal, asmFish, Fire, Crafty, Senpai, Rodent, Cinnamon, Toga, Houdini, Shreder, Laser, Lc0, Leelenstein, Antifish, Allie тощо. І це тільки початок. На сайті Вікіпедії з програмування шахів [5] можна знайти величезний список відомих шахових рушіїв і також є можливість завантажити їх на свій комп’ютер. Також існує дуже багато так званих «аматорських» рушіїв написаних простими ентузіастами. Всі вони зберігаються на платформі Github і чемно чекають свого використання.

Отож, Мінімакс – це свого роду алгоритм зворотного відстеження, який використовується в процесі прийняття рішень і теорії ігор, щоб знайти оптимальний хід для гравця, з припущенням, що ваш опонент також грає оптимально. Він широко використовується в покрокових іграх для двох гравців, таких як: Шахи, Хрестики Нулики, Нарди, Манкала тощо.

Він здійснює пошук усіх можливих ходів до фіксованої глибини, оцінює всі кінцеві позиції та використовує ці оцінки для відстеження результатів аж до кореня дерева пошуку.

Ідея полягає в тому, що гравці перебирають усі можливі ходи у своєму стані, а потім вибирають, відповідно, той, який робить значення позиції якомога вищим або ж якомога нижчим. В алгоритмі Мінімакс таких гравців називають максимізатор (maximizer) і мінімізатор (minimizer). Максимізатор намагаться отримати найвищу кількість очок, а мінімізатор робить навпаки – намагається отримати найнижчу кількість очок.

Звідки ж беруться очки, які потрібні для оцінки ходів? Це так звана вартість фігур. У підрозділі 2.4.5 вже були зазначені числа-відповідники кожній фігурі, а саме: 10 – пішаки, 30 – слони, 35 – коні, 50 – тура, 90 – королева, 900 – король. Всі ці значення зберігають на окремій дошці очок, яка змінюється разом з основною.

Так от. В чому заключається, власне, робота даного алгоритму в нашій грі? Все починається з методу FindBestMove(). Цей метод відповідає за ініціалізацію об’єкту найкращого ходу та активацію самого алгоритму, що реалізовано в методі MinimaxAlgorithm() (рис. 2.75).

```
public static Move FindBestMove(GameState state, int depth) {
    Move bestMove = new Move(); int bestVal = int.MinValue;
    foreach (Move move in state.validMoves) {
        GameState nextState = MakeMove(state, move);
        int val = MinimaxAlgorithm(nextState, depth - 1, true);
        if (val > bestVal) bestVal = val; bestMove = move;
    }
    return bestMove;
}
```

Рис. 2.75. Ініціалізація та активація алгоритму

Це метод як аргументи приймає поточний стан дошки state та глибину перебирання depth. Це значення й відповідає за точність вибору ходу. Чим більше значення depth, тим точніше і краще працює алгоритм. Щоправда при великих значеннях depth йде дуже велике навантаження на комп’ютер, бо це рекурсивний алгоритм, тому ми використовуватимо невеликі значення depth.

Рівень складності гри з комп’ютером визначає саме значення глибини depth, оскільки чим більше значення, то тим більшим формується дерево пошуку і відповідно знаходиться кращий хід. Тому в класі TileSelector перед

викликом цього методу відбувається зчитування значення з файлу GameDifficulty.txt, в якому попередньо були записані дані про умовну складність гри. І на основі нього формується значення глибини depth. Якщо рівень складності «Легко», то значенням depth буде 3, а якщо рівнець складності «Нормально» - то 5 тощо.

За хід відповідає клас Move, в якому зберігаються координати початкової позиції фігури, та координати кінцевої позиції фігури (рис. 2.76).

```
public class Move {
    public int fromX, fromY; public int toX, toY;
    public Move(int fX = 0, int fY = 0, int tX = 0, int tY = 0){
        fromX = fX; fromY = fY;toX = tX; toY = tY;
    }
}
```

Рис. 2.76. Виконання умовного ходу

За визначення стану відповідає клас GameState, який зберігає стан усієї дошки та список усіх можливих та коректних ходів усіх фігур (рис. 2.77).

```
public class GameState {
    public int[,] boardArr; public List<Move> validMoves;
    public GameState(int[,] arr, List<Move> moves) {
        boardArr = arr; validMoves = moves;
    }
}
```

Рис. 2.77. Визначення стану

Список ходів формується методом AllPossibleMoves() (рис. 2.49) класу GameManager, що описаний у підрозділі 2.4.5.

Далі йде перебирання усіх цих ходів. Зі списку по порядку обирається хід та аналізується. На рис 2.78 добре показано процес перебору ходів.

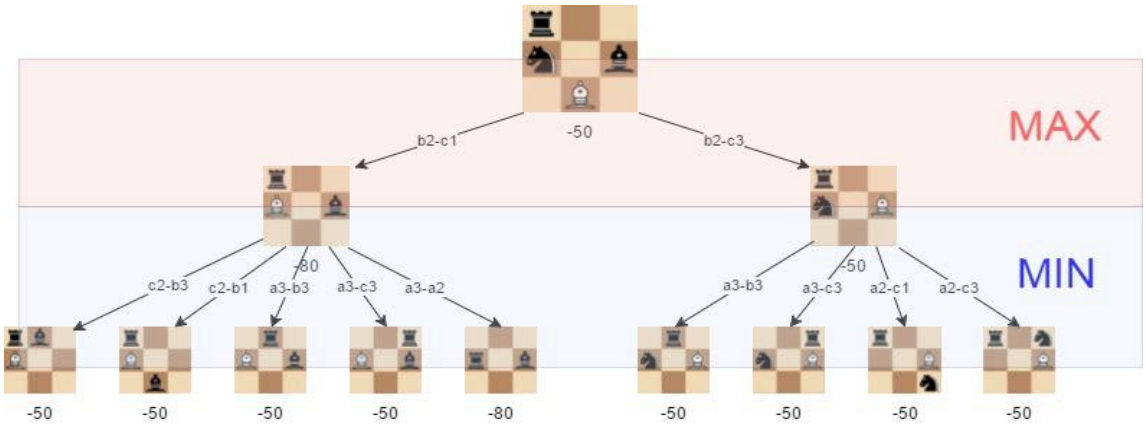

Рис. 2.78. Процес перебору ходів

Також в роботі приймає участь метод Evaluate(), який і відповідає за оцінку ходів. На вхід методу подається стан дошки і робиться підрахунок очок, тобто підраховується кількість білих фігур та їхню загальну вартість, та чорних фігур з іхньою вартістю, і шукається різниця між ними. Якщо різниця буде від'ємною, то це означає, що це досить хороший хід, оскільки було вбито певну білу фігуру противника або білих фігур просто напросто менше ніж чорних, і навпаки, якщо ж різниця додатня, то це означає, що або білі захопили певну чорну фігуру, або чорних фігур менше ніж білих (рис. 2.79).

```
public static int Evaluate(GameState state) {
    int whiteScore = 0, blackScore = 0;
    foreach (var item in state.boardArr){
        if (item > 0) whiteScore += item;
        else if (item < 0) blackScore += -item;
    }
    return blackScore - whiteScore;
}
```

Рис. 2.79. Оцінка ходів

Ще є метод MakeMove() і він, власне, робить усі ходи, тобто переміщує очки фігур у потрібні позиції а також заново формує список ходів (рис. 2.80).

```
public static GameState MakeMove(GameState state, Move move) {
    int score = state.boardArr[move.fromX, move.fromY];
    state.boardArr[move.fromX, move.fromY] = 0; state.boardArr[move.toX, move.toY] = score;
    state.validMoves = GameManager.instance.AllPossibleMoves();
    return state;
}
```

Рис. 2.80. Виконання ходів

І ось, власне, метод MinimaxAlgorithm(). Це рекурсивний метод, тобто при роботі він викликає самого себе (рис. 2.81).

```
public static int MinimaxAlgorithm(GameState state, int depth, bool maximizingPlayer) {
    if (depth == 0 || state.validMoves.Count == 0) return Evaluate(state);
    if (maximizingPlayer) {
        int maxVal = int.MinValue;
        foreach (Move move in state.validMoves) {
            GameState nextState = MakeMove(state, move);
            maxVal = Math.Max(maxVal, MinimaxAlgorithm(nextState, depth - 1, false));
        }
        return maxVal;
    } else {
        int minVal = int.MaxValue;
        foreach (Move move in state.validMoves) {
            GameState nextState = MakeMove(state, move);
            minVal = Math.Min(minVal, MinimaxAlgorithm(nextState, depth - 1, true));;
        }
        return minVal;
    }
}
```

Рис. 2.81. Алгоритм Мінімакс

В залежності від значення maximizingPlayer по різному проводиться робота. Відбувається хід і відразу ж аналізується наступний шляхом повторного виклику методу MinimaxAlgorithm(), але значення depth, тобто глибина, декрементується, аналіз проводиться на новому стані дошки, та змінюється гравець з максимізатора на мінімізатора, або навпаки.

Коли значення depth стає 0, тобто коли алгоритм дійшов до потрібної глибини дерева відбувається оцінка за допомогою методу Evaluate(). Цей блок знаходиться на початку методу і перевіряється при виклику.

І далі рекурсія обривається та відбувається повернення до початкового стану, але з кінцевою оцінкою ходу. Якщо значення maximizingPlayer дорівнює true, тобто коли хід робиться за максимізатора, шукається, як було написоно вище, максимальне значення оцінки. При false відповідно мінімальне значення.

Це значення відправляється назад у початковий метод FindBestMove() і вже там робиться остаточне порівняння ходів. Якщо оцінка поточного ходу вища за оцінку минулого ходу, то найкращий хід змінюється. Якщо ж оцінка менша, то й хід не змінюється.

І ось після всієї зробленої роботи, після аналізу всіх можливих ходів усіх фігур знаходиться найкращий і вже на основі нього проводиться пересування фігури на основній дошці.

Алгоритм «Мінімакс» це свого роду база для інших алгоритмів. Є купа набагато оптимізованіших та складніших в плані обробки ходів алгоритмів. Але їх усіх не було б без цього алгоритму. Зв’язок Мінімаксу та цих алгоритмів чимось схожий на зв’язок перцептрона та розвинених нейронних мереж – другого не було б без першого. Тому не потрібно відкидати його використання, адже все нове – добре забуте старе.

ВИСНОВКИ

Після проведення аналізу програмного забезепечення та різних версій комп’ютерних шахів, створених на ньому, а також після вивчення наукових матеріалів було розроблено комп’ютерну візуалізацію гри «Шахи».

В ході розробки було:

- змодельовано тривимірні зразки шахових фігур та шахової дошки засобами пакету Blender та графічного редактору Adobe Photoshop;
- створене початкове меню з налаштуваннями гри засобами ігрового рушія Unity;
- реалізована гра «людина з людиною» засобами ігрового рушія Unity та інтегрованого середовища розробки Microsoft Visual Studio;
- реалізована гра «людина з комп’ютером» засобами інтегрованого середовища розробки Microsoft Visual Studio;

В ході написання письмової роботи було:

- досліджено та описано коротку історію виникнення гри «Шахи»;
- досліджено та описано історію написання комп’ютерних шахів;
- описано сучасне програмне забезпечення та їх інструменти, а саме: Unity, Blender, Microsoft Visual Studio та Adobe Photoshop;
- описано повний процес розробки, та пояснення коду гри;
- досліджено, описано та пояснено алгоритм «Мінімакс»;

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Chess projects. *Scratch*. URL: https://scratch.mit.edu/search/projects?q=chess (дата звернення: 09.12.2022).
2. DenshiVideo. The Most Impressive Scratch Projects, 2022. *YouTube*. URL: https://youtu.be/tpGzqzpjmRg (дата звернення: 09.12.2022).
3. DevBanana. Making A Rhythm Game Inside PowerPoint…, 2022. *YouTube*. URL: https://youtu.be/YhQaYyDkNJc (дата звернення: 09.12.2022).
4. Donovan T. Replay: The history of video games. East Sussex, England : Yellow Ant, 2010. 501 p.
5. Engines – Chess programming wiki. URL: https://www.chessprogramming.org/Engines (дата звернення: 23.05.2023).
6. Guevarra E. T. M. Modeling and Animation Using Blender. Berkeley, CA : Apress, 2020. 312 p.
7. Hocking J. Unity in Action, Third Edition. Manning Publications Co. LLC, 2021. 419 p.
8. List of game engines. *Wikipedia*. URL: https://en.wikipedia.org/wiki/List_of_game_engines (дата звернення: 10.12.2022).
9. Murray H. J. R. A history of chess. Oxford : Clarendon Press, 1913. 945 p.
10. Rob Price. Minecraft in MS Excel. *The Daily Dot*. URL: https://www.dailydot.com/parsec/minecraft-in-microsoft-excel/ (дата звернення: 09.12.2022).

11. SethEric. I Made A Game in PowerPoint in 24 Hours! | Devlog, 2022. *YouTube*. URL: https://youtu.be/eHCxgy4Gxkw (дата звернення: 09.12.2022).
12. Surya Excel Software. Play Chess Game in MS-EXCEL, 2020. *YouTube*. URL: https://youtu.be/A-s8LjUqgUc (дата звернення: 09.12.2022).
13. UnityScript's long ride off into the sunset. *Unity Blog*. URL: https://blogs.unity3d.com/2017/08/11/unityscripts-long-ride-off-into-the-sunset/ (дата звернення: 10.12.2022).
14. Visual Studio. *Wikipedia*. URL: https://en.wikipedia.org/wiki/Visual_Studio (дата звернення: 10.12.2022).
15. 田中信秋. Doom runs on Excel, 2022. *YouTube*. URL: https://youtu.be/J2qU7t6Jmfw (дата звернення: 09.12.2022).