math_2020_078

Міністерство освіти і науки України

Чернівецький національний університет

імені Юрія Федьковича

Факультет математики та інформатики

(повна назва інституту/факультету)

Кафедра математичного моделювання

(повна назва кафедри)

Розробка мультиплеєрної гри

Дипломна робота

Рівень вищої освіти - другий (магістерський)

Виконав:

студент 6 курсу, групи 607

спеціальності 124 – Системний аналіз

(назва спеціальності)

Вишиванюк Ярослав Ігорович

(прізвище, ім’я та по-батькові)

Керівник д.ф.-м.н., доцент Клевчук І.І.

(науковий ступінь, вчене звання, прізвище та ініціали)

До захисту допущено:

Протокол засідання кафедри № 7

від „ 15 ” грудня 2020 р.

зав. кафедри _________ проф. Черевко І.М.

Чернівці – 2020

Зміст

Вступ 4

Розділ 1. Основні поняття про предметну область 5

1.1 Відеоігри 5

1.2 Розробка комп'ютерних ігор 7

1.3 Індустрія відеоігор 9

Розділ 2. Теоретична частина 11

2.1 Метод Кидання променів 11

2.2 Алгоритм художника 15

Розділ 3. Технології та середовище розробки 18

3.1 Мова програмування С++ 18

3.2 Середовище розробки – Visual Studio 21

3.3 Ігровий рушій 23

Розділ 4. Підготовча частина 25

4.1 Постановка задачі 25

4.2 Вивчення літератури 26

Розділ 5. Практична частина 27

Висновки 39

Література та використані джерела 40

Додатки 41

Анотації

У рамках дипломної роботи розроблено ігровий рушій, на основі якого створено мультиплеєрну гру в жанрі шутер. Основу цього ігрового процесу складає стрільба зі зброї по цілях. Дану програму розроблено на основі алгоритму Ray casting (кидання променів), реалізовано мовою програмування C++.

Вступ

В даний час відеоігри в нашому житті називають комп'ютерними ігровими додатками. Але розробники цих ігор та дослідники використовують термін “відеогра”. Відеоігри появилися завдяки прогресу програмованих комп'ютерних технологій формування зображення на моніторах електронних пристроїв. Першими і найпопулярнішими відеоіграми є ігри для персонального комп'ютера, яку можна встановити на персональному комп'ютері. Також її називають просто комп'ютерна гра. Механіка відеогри визначається розміром ігрового процесу, а також правилами, за якими гравець проходить відеогру. Основним у грі є об'єкти, такі як гравець, об'єкти, якими вони можуть керувати, текстури. Для того, щоб гравець якимось чином зміг керувати своїм персонажем або ігровим світом є механіка управління.

Розділ 1. Основні поняття про предметну область

1.1 Відеоігри

Появі мультиплеєрних ігор передували розвиток технологій та комп'ютерних програм, які відображали зображення на моніторах електронних пристроїв. Деякі механічні або електронні ігрові пристрої з'явилися ще в першій половині XX століття, але не були доступні кожному. Перша успішна спроба створення електронного пристрою, який використовується для зв'язку із гравцем у відеогрі, належить Вільяму Гіґінботаму. У 1958 було створено Tennis For Two.

У 2000-ні роки ігри в плані графіки ставали все реалістичнішими, розвинулися online-ігри, які можна грати по локальній мережі, що дає можливість багатьом гравцям грати в реальному часі. Наприкінці 2010-х відбувся розвиток та вплив соціальних мереж. Відеоігри на фізичних носіях перестали бути такими популярними, поступаючись цифровим. Це зумовило поширення indi-ігор, відеоігор від незалежних програмістів чи навіть окремих людей. Самі гравці стали розробляти, створювати і модифікувати відеоігри шляхом моддингу, конструювання інтерактивних творів у спеціальних середовищах. Популярним став випуск бібліотек для доповнення до ігор, у тому числі платних, запроваджувати мікротранзакції, для того, щоб полегшувати гру. Швидко підвищилися якість графіки після виходу 8-го покоління ігрових систем, що зумовило поширення 4K-дисплеїв.

Значно збільшилася кількість ігор і пристроїв з підтримкою VR-реальності. Також розвинулася індустрія додатків для мобільних пристроїв, зокрема смартфонів. Стримінг ігрового процесу став частиною культури відеоігор. Розробка гри має декілька етапів, їх є три: написання програмного (джерельного) коду, впровадження контенту (малюнки, моделі) та розробка ігрового рушія. Їх проектування включає —

створення ідеї щодо майбутньої гри, впровадження жанру, тематики, особливостей ігрового рушія, створення сценарію та образів гравців з оточенням. Менеджер керує діями різних людей, які залучені до розробки, складає план їхньої роботи, визначає термін їх виконання, планує витрати на створення. Відеогра в свою чергу проходить деякі етапи перевірки, в ході яких потрапляє до людея і підтримує інтерес до проходження. Відеоігри охоплюють багато людей з різних країн.

У 2011-ці відеоігри стали офіційним видом мистецтва, але у світі визнання їх мистецтвом залишається під питанням.

Першим, соціальним аспектом розгляду відеоігор є модель поведінки суспільства та людей. Проводячи дослідження про поведінку в стандартних і нестандартних ігрових ситуаціях, також можна прогнозувати як людина буде діяти в інших сферах життя. Другим, гуманітарним аспектом відеоігор є складова сучасної культури суспільства. Він визначає їх зв'язок з різними структурами культури людини й мислення та іншими видами мистецтва. Третім способом ігри вивчають з точки зору як їх потрібно розробляти. Цей напрям вивчає проблеми у створенні успішних відеоігор, вирішенні проблем, пов'язаних з ними.

1.2 Розробка комп’ютерних ігор

Розробка ігор почалась у 1970-ті роки XX сторіччя, разом з появою та використанням аркадних автоматів та гральних приставок. Через незначну потужність перших комп'ютерних обладнань, ігри були простими у створенні і не потребували багато часу на розробку. Їх міг створювати і успішно завершувати один розробник. У зв’язку з зростанням обчислювальних потужностей гравцям було замало таких простих відеоігор. Тому зросла і складність розробок, для яких потрібна робота цілих команд розробників. Розробка відеогри проходить у декілька складних етапів. Спочатку створюється загальний демо-проєкт майбутньої гри, описується дизайн-документ яким буде всесвіт, ігровий процес і сюжет, створюються концепти рівнів графіки і персонажів у грі, які визначатимуть жанр гри і які слугуватимуть завданням для художників та дизайнерів. Дизайн-документ та малюнки також використовуються для привернення уваги видавців.

Якщо вирішено проблему з фінансуванням, починається повноцінна робота над відеогрою, яка включає створення чи налаштування під потреби функціональності гри та ігрового рушія, створення графічного, 2-3-вимірного, та інтеграція аудіо у ігри, впровадження та тестування для налагодження ігрових механік (ігровий дизайн). До своєї останньої версії гра йде через бета-альфа версії, які ретельно перевіряються і тестуються розробниками, іноді проводять тестування самі гравці (відкриті альфа/бета-тести). Через деякий час після останніх перевірок гри та тестувань функціоналу, гра виходить, тобто починає продаватись для людей. Після виходу даної гри триває підтримка, протягом деякого часу гру вдосконалюють і доповнюють.

Кодом гри є комп'ютерна програма, її спосіб використання, технічні можливості, графіка та ігровий час забезпечується програмним кодом.

Розробка відеогри включає деякі етапи, як і розробка програмного коду, але з набагато більшою роботою над контентом і створенням ігрового функціоналу. Відеоігри здебільшого написані на готових програмних бібліотекам — ігрових рушіях, де вже зроблені базові функції, здатні зв'язувати в одне ціле графіку, звук, об'єкти і їх дії.

Ігровий функціонал визначає наскільки ігровий процес буде насиченим, правила певної відеогри. Основою механіки є ігрові персонажі, які пов’язані з об'єктами або з якими вони можуть маніпулювати. Частиною ігрового світу є управління, яким чином гравець керує своїм персонажем або іншими об’єктами в ігровому світі. Наприклад, як вказується напрям руху, як гравець взаємодіє з віртуальними об’єктами. Крім того, на етапі розробки функціоналу створюється ігровий інтерфейс, який в свою чергу інформує гравця і дозволяє взаємодію персонажа із світом гри.

Зазвичай після виходу відеогри в ній можуть бути досі невідомі помилки. Гравцю може не подобатись певний аспект гри, до прикладу, дизайн. Розробники, які створили гру, в такому разі розробляють патчі, які виправляють помилки і додають якісь вдосконалення. Розробка і випуск нових бібліотек не приносять великих прибутків, але можуть збільшити продажі відеогри, оскільки вона позбавляється недоліків.

1.3 Індустрія відеоігор

Розвиток ігор відбувся в середині 70-х років як рух розробників і за швидкий час виріс з невеликого ринку продажів в мейнстрім. Сучасні персональні комп'ютерні системи дали багато нових ігрових індустрій. До числа найвідоміших відносять звукові та графічні відеокарти, CD і DVD - приводи, Unix та Linux центральні процесори. Також операційна система Unix розроблена для запуску комп'ютерної гри про дослідження глибокого космосу.

Спочатку звукові карти були створені для інтегрування чіткого цифрового звуку в комп'ютерні відеоігри, і тільки потім звукові пристрої були вдосконалені під потреби меломанів.

Графічні відеокарти, які на етапі комп'ютерної епохи розвинулись у напрямі збільшення кількості доступних кольорів, пізніше також стали розвиватися для графічного інтерфейсу користувача та відеоігор. GUI зіграв роль у збільшенні роздільної здатності екрану, а ігри — в 3D прискорення і винаходи таких технологій, як SLI і CrossFire. Використання CD і DVD дисків для розповсюдження ігор породив необхідність у збільшенні обсягів і швидкості читання.

В ігровій розробці часто працюють ті, хто вже має хоч якийсь досвід у подібних до неї професіях (таких як програміст, художник, музикант), але деякі люди мають досвід у роботі стосовно лише до відеоігрової індустрії. Багато із тих, кого наймають фірми-розробники, займаються продажем комп'ютерних ігор. Однак є багато розробників, які розробляють ігри самостійно, а потім виставляють їх на продаж.

Дослідження, які мали відбуватися в українській відеоігровій індустрії, тривалий час не проводилися, а за цей час ігри стали популярними. Це питання стали піднімати тільки в другій

половині 2000-х. На початку це було зумовлено засекреченням українськими програмістами ділової інформації щодо вартості своїх проектів. Але багато хто знав, що розробка відеоігор в Україні коштує значно менше. В нашій країні програмувалося багато різноманітних замовлень, але озвучення і анімація персонажів не розроблялися. Причиною цього вважалися вимоги замовників і відсутність потрібних технологій.

Розділ 2. Теоретична частина

2.1 Метод Кидання променів

Метод Ray cast (рейкастинг) або кидання променів полягає у перетині променів з поверхнею для обчислення та розв'язування різних математичних задач у комп'ютерній графіці та геометрії. Рейкастинг може стосуватися різних прикладних задач та деяких методів:

- загальна задача знаходження першого об'єкта, який перетинається випущеним променем;
- метод усунення прихованої не в полі зору поверхні, що базується на знаходженні першого перетину променя, який починається з точки огляду через кожен піксель зображення;
- алгоритм побудови території без рекурсивного трасування променів, який включає тільки первинні промені;
- метод прямого об'ємного відображення території (об'ємне кидання променя), в якому промінь "проштовхує" об'єкт, а 3-вимірне скалярне поле, що має відображатися, збирається вздовж променя всередині даного об'єкта (у цьому методі не відбувається утворення вторинних променів).

Отже, поняття "кидання променів" та "трасування променів" в ранній комп'ютерній графіці часто використовувалися як тотожні. Останнім часом їх пробують відрізняти. Відмінність полягає в тому, що кидання променів — це метод відтворення рендерингу, при якому рекурсивно ніколи не відстежуються всі вторинні промені, тоді як інші алгоритми відображення на основі трасування променів можуть це зробити.

Кидання променів є найбільш зручним і простим з багатьох методів рендерингу в комп'ютерній графіці, які базуються на геометричному методі трасування променів. Алгоритми відображення території на базі трасування променів працюють в **image order** для відтворення 3-вимірних сцен на 2-вимірних екранах монітора. Геометричні промені випускаються з ока гравця для зразка світла (сяйва), що рухається до спостерігача від напрямку випущеного променя. Швидкість і легкість кидання променів забезпечується всім обчисленням кольору світла без рекурсивного відстеження всіх додаткових променів, які відбивають сяйва, що падають на дану точку попадання випущеного променя. Це дає можливість точного рендерингу відбиття променя, заломлення або природного падіння тіней на карту. Але всі ці елементи можливо будуть певною мірою підроблені шляхом зміни творчого використання текстурних карт або інших алгоритмів. У свою чергу висока швидкість обчислення зробила кидання променів зручним алгоритмом відображення території в ранніх 3D-відеоіграх реального часу.

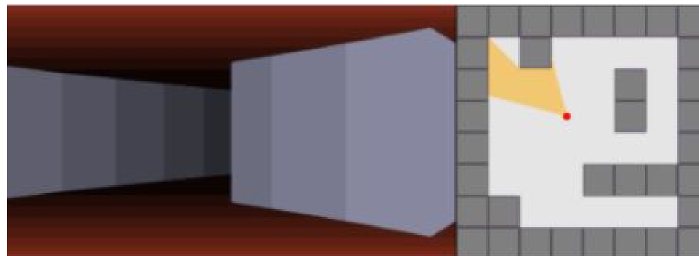


Рис. 1 (Приклад методу Кидання променів)

У нашій природі джерело світла випускає промінь світла, який, у свою чергу, рухається до поверхні, яка й зупиняє його хід. Ми можемо уявити цей “промінь” як потік певної кількості фотонів, що рухаються вздовж тієї самої траєкторії. На його шляху може бути поєднання трьох речей з цим випущеним світловим променем: поглинання, відбиття та рефракція. Деякі поверхні можуть відбити весь або частину світлового

променя в одному напрямку або зразу в декількох. Поверхня також може поглинути частину світлового променя, що приведе в результаті до втрати інтенсивності відбитого (та|або) заломленого світла. Якщо поверхня, на яку попав промінь, має будь-які прозорі або напівпрозорі властивості, вона заломлює деяку частину світлового променя в собі. В іншому випадку відбудеться поглинання або зміна кольору. Між поглинанням, відбиттям та заломленням повинно враховуватися все світло, яке прийшло. Деяка поверхня, наприклад, не може відбити 60-66% вхідного світлового випущеного проміння, і відобразити 50%. Отже, відбиті (та|або) заломлені промені можуть "вражати" інші можливі поверхні, де їх поглинаючі, рефракційні та відбиваючі значення знову підраховуються на основі вхідних променів. Деякі з цих променів рухаються так, що потрапляють в очі, змушуючи нас бачити певну дійсність і цим самим сприяють появі зображення. Спроба імітувати цей реальний процес випромінювання світлових променів за допомогою електронної машини може вважатися надзвичайно марною, оскільки лише мала частка фізичних променів світла дійсно зможуть потрапити у видимість.

Територія побудована з квадратної сітки стін однакової висоти, які контактують з суцільними підлогами і стелями. Отже, щоб промалювати територію, відстежується один випущений промінь для кожного стовпця пікселів на 2D екрані, де вертикальний фрагмент даної стінки вибирається та малюється відповідно до того, як даний промінь натикається на стіну території і яку відстань він рухався перед цим зіткненням.

Створення певних рівнів, побудованих на сітці, проводиться за два етапи: зіткнення випущеного променю зі стіною можна знайти швидше, оскільки потенційні зіткнення є більш очевидними та це потребує набагато

менше ресурсів для роботи. Проте моделювання широких відкритих областей займає деякий додатковий простір

Також в геометрії питання рейкастингу також відоме як проблема **Ray shooting** і може бути зазначене як наступна поставлена задача. Враховуючи великий набір об'єктів у n-вимірному просторі, перша обробка структури всіх даних є такою, щоб для кожного випущеного променя можна було досить швидко знайти початковий стан, на який потрапляє промінь. Одним з методів вирішення цього питання є Sparse Voxel Octree.

2.2 Алгоритм художника

Алгоритм художника, який ще називають пріоритетним малюванням, є одним з найпростіших і зручних методів розв'язування проблем які можуть виникнути в комп'ютерній 3D графіці. При рендеренгу 3D-сцени на 2D площину, необхідно в якийсь момент часу знати, які багатокутники будуть видимими, а які прихованими. Назва "алгоритм художника" також відноситься до деякої техніки, яка використовується багатьма відомими художниками живопису для віддалення якоїсь частини сцени, яка перекривається іншими частинами, розташованими ближче до спостерігача. Алгоритм художника сортує всі об'єкти на сцені за їх глибиною і потім починає малювати їх у порядку від найвіддаленішого до найближчого. Алгоритм буде замальовувати ті частини, які не буде видно глядачу. Тому не треба малювати віддалені об'єкти, які і так не будуть видимими. Алгоритм використовує поняття "порядок глибини", тобто числову відстань до всіх частин сцени. Головна властивість цієї черги, полягає в тому, що якщо один з об'єктів заступає частину іншого об'єкта, то перший об'єкт буде замальований після об'єкта, який він буде приховувати. Отже, такий порядок звісно може бути зображений як топологічне впорядкування орієнтованого ациклічного графа, що знає місце розташування об'єктів.

Недолік в тому, що не завжди алгоритм може розташувати об'єкти в деяких випадках. Такий випадок можливий при циклічному перекритті або відображенні багатокутників. У разі циклічного перекриття багатокутників, як показано на малюнку (Рис.2), багатокутники A, B, і C перекривають один одного так, що неможливо знати, який багатокутник найближчий, а який ні. У даному випадку багатокутники повинні бути розрізані для подальшого сортування. Також є випадок "проколювання" багатокутників, коли один багатокутник перетинає частину площини

іншого. Як і з циклічним перекриттям, дану проблему можна вирішити за рахунок розрізання відповідного багатокутника.

В багатьох основних реалізаціях алгоритм художника може бути не повністю ефективним. Це дає системі розглядати кожну точці площини та кожний багатокутник у видимій зоні, навіть якщо ця модель багатокутника є в готовій сцені. Отже, це означає, що для отримання чітких сцен, алгоритм художника може бути складним для комп'ютерного обладнання.

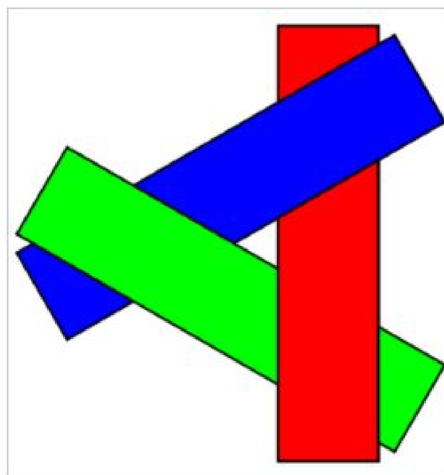


Рис. 2 (Немає найближчого багатокутника)

Зворотний алгоритм художника повертає впорядковані об'єкти, починаючи з найближчого і закінчуючи найвіддаленішим. Використовується правило, що фарба не повинна бути використана до тої частини зображення, яка вже є замальована. При побудові будь-якого зображення, це може бути дуже ефективним методом, оскільки відпадає потреба в обчисленні зображення (для цього потрібен вирахувати освітлення, накладання текстур та інше) для частин більш віддалених від точки зору, які закриті ближчими об'єктами. Однак у цього алгоритму також є недоліки, як і у стандартного.

Ці а також інші недоліки алгоритму змусили перейти до розвитку методу Z-буфера, який можна розглядати як наступну версію алгоритму художника. Цей алгоритм попіксельно розв'язує питання глибини кожного

об'єкта. Навіть в таких складних системах іноді використовують алгоритм художника. Оскільки при використанні Z-буфера застосовують сталу точність регістрів буферу, яку реалізовано на низькому апаратному рівні. Тому виникають можливі проблеми видимості через помилки округлення об’єктів. Щоб уникати цієї помилки, деякі графічні реалізації “накладають” ребра таких об’єктів даних многокутників так само як і в алгоритмі художника. Це означає, що деякі елементи текстур будуть обчислені два рази (так само як і в алгоритмі художника), але це буде лише використовуватися на невеликих ділянках певного зображення і мало буде впливати на продуктивність всього алгоритму.

Розділ 3. Технології та середовище розробки

3.1 Мова програмування C++

C++ одна з мов програмування високого рівня з підтримкою кількох базових парадигм програмування: об'єктно орієнтованої, узагальненої та процедурної. Дана мова розроблена Б'ярном Страуструпом в 1979-му році. У 1990-х роках C++ стала однією з найбільш використовуваних мов програмування для загального призначення і різних задач. Мову використовують для системного програмування на низьких рівнях системи, розробки програмного забезпечення, написання драйверів для відео та аудіо, потужних серверних, клієнтських програм та хмарних обчислень, а також для розробки розважальних програм, наприклад, відеоігор або додатків. Також мова C++ суттєво вплинула на інші популярні тепер мови програмування такі як C# та Java.

Перед тим, як створювали C++, прагнули зберегти сумісність з мовою попередником C. Більшість програм на C справно працюватимуть і з компілятором для C++. Мова програмування має синтаксис, заснований на синтаксисі C.

Нововведеннями в мові C++ є:

- підтримка об'єктно-орієнтованого програмування через класи;
- підтримка узагальненого програмування через шаблони;
- доповнення до стандартної бібліотеки;
- додаткові типи даних;
- обробка винятків;
- простори імен;
- вбудовані функції;
- перевантаження операторів;

- перевантаження імен функцій;
- посилання і оператори управління вільно розподіленою пам'яттю.

Мова програмування С++ багато в чому є надмножиною мови С. Багато нових можливостей С++ включають оголошення у вигляді виразів, перетворення примітивних типів у вигляді функцій, а також оператори **new** і **delete**, тип **bool**, посилання, розширене поняття константності так як змінних, функції, що підставляються, аргументи за замовчуванням, перевизначення, простори імен (**namespace**), тип **class** (включає і всі пов'язані з ним класи можливості, такі як успадкування, методи, віртуальні функції, абстрактні класи і конструктори), перевизначення операторів, шаблони, оператор **::**, всі обробки винятків, динамічну ідентифікацію і багато що інше. Мова С++ є також мовою строгого використання типів і накладає більше вимог щодо дотримання типів, порівняно з мовою С.

Деякі особливості С++ пізніше були перенесені в С, наприклад ключові слова **const** і **inline**, оголошення в циклах **for** і коментарі в стилі С++ (“//”). У пізніших реалізаціях С також були представлені можливості, яких немає в С++, наприклад макроси **vararg** і покращена робота з масивами-параметрами.

Переваги мови С++

- Швидкодія. Швидкість роботи програм на С++ практично не поступається програмам на С, хоча програмісти отримали в свої руки нові можливості і нові засоби.
- Масштабованість. На мові С++ розробляють програми для найрізноманітніших платформ і систем.

- Можливість роботи на низькому рівні з пам'яттю, адресами, портами (що при необережному використанні може легко перетворитися на недолік).
- Можливість створення узагальнених алгоритмів для різних типів даних, їхня спеціалізація і обчислення на етапі компіляції з використанням шаблонів.
- Підтримуються різні стилі та технології програмування, включаючи традиційне директивне програмування, ООП, узагальнене програмування, метапрограмування (шаблони, макроси).

3.2 Середовище розробки – Visual Studio

Microsoft Visual Studio — серія продуктів фірми Майкрософт, які включають інтегроване середовище розробки програмного забезпечення та низку інших інструментальних засобів. Ці продукти дозволяють розробляти як консольні програми, так і програми з графічним інтерфейсом, в тому числі з підтримкою технології Windows Forms, а також веб-сайти, веб-застосунки, веб-служби.

Маючи понад 250 нових функціональних можливостей, Visual Studio пропонує істотні поліпшення кожної з версій:

- **Language Integrated Query (LINQ)** заповнює прогалину між об'єктним програмуванням і даними та дозволяє розробникам зосередитися не на доступі до даних, а на роботі з ними.
- **Visual Studio Team System** підтримує управління збіркою програм, включаючи виконання планових збірок та збірок в результаті процесу безперервної інтеграції. Team Build забезпечує інтегровану підтримку статичного аналізу коду під час виконання збірки і проведення контрольних випробувань збірки.
- Значне спрощення розробки для Web завдяки новій техніці обміну інформацією з веб-сервером для веб-сайтів, які підтримують AJAX/JSON. Нові елементи управління ASP.NET передбачають покращене управління сторінками і шаблони, а Windows Communication Foundation передбачає вбудовану підтримку RSS і REST.
- **.NET Framework 3.5** також містить декілька нових функціональних можливостей, серед яких можна назвати можливості для Web 2.0, сервіс-орієнтовану архітектуру (Service-Oriented Architecture, SOA) та програми на базі технології ПЗ + Сервіси (Software+Services). Сервіси з підтримкою послідовності операцій надають нові класи моделі

програмування, які спрощують створення сервісів з підтримкою послідовності операцій за рахунок використання Windows Communication Foundation і Windows Workflow Foundation. Це дозволяє розробникам на .NET Framework створювати бізнес-логіку сервісу, використовуючи WF, та організовувати обмін повідомленнями з цим сервісом за допомогою WCF.

- Підтримка додаткових протоколів веб-сервісів у Windows Communication Foundation, включаючи протоколи Web Services Atomic Transaction (WS-AtomicTransaction) 1.1, WS-ReliableMessaging 1.1, WS-Secure Conversation та Web Services Coordination (WS-Coordination) 1.1.
- **Text Template Transformation Toolkit** шаблонно орієнтований генератор коду, включений як частина середовища.

Інструменти Visual Studio допоможуть не тільки в створенні звичних програм для мобільних телефонів і персональних комп'ютерів, але в розробці хмарних застосунків. При цьому процес тестування і розгортання програм в “хмарі” аналогічний створенню .NET-застосунків. Іншим важливим доповненням в Visual Studio є інструменти для розробки з використанням як некерованого коду, так і .NET Framework.

3.3 Ігровий рушій

Програмний рушій, центральна програмна частина будь-якої відеогри, яка відповідає за всю її технічну сторону, дозволяє полегшити розробку гри шляхом уніфікації та систематизації її внутрішньої структури. Важливим значенням рушія є можливість створення багатоплатформових ігор (сьогодні найчастіше одночасно для ПК, PS4 та Xbox One).

Основну функціональність гри зазвичай забезпечує її рушій, до якого входить рушій рендерингу (“візуалізатор”), фізичний рушій, звук, система скриптів, анімація, ігровий штучний інтелект, мережевий код, керування пам’яттю, багатонитевість і граф сцени. Часто на процесі розробки можна заощадити шляхом повторного використання одного рушія гри для створення декількох різних ігор.

Подальше вдосконалення ігрових рушіїв призвело до сильного розділення між рендерингом, скриптингом, художнім дизайном і дизайном рівнів. Зараз для типової команди розробників ігор є цілком звичайним мати в складі стільки ж художників, скільки й програмістів. Ігрові рушії також використовуються в іграх, спочатку розроблених для гральних консолей; наприклад, рушій RenderWare використовується у франчайзах Grand Theft Auto III та Burnout.

Сучасні ігрові рушії — одні з найскладніших у написанні застосунків, що найчастіше складаються з десятків різних компонентів, кожен з яких можна налаштовувати окремо під потреби гри. На сайті Future Game Coders є різні теми про підсистеми сучасних ігор. На додаток до багаторазово використовуваних програмних компонентів, ігрові рушії надають набір візуальних інструментів для розробки. Ці інструменти зазвичай складають інтегроване середовище розробки для спрощеної, швидкої розробки ігор на зразок потокового виробництва. Такі

рушії іноді називають “ігровим підпрограмним забезпеченням”, тому що, з погляду бізнесу, вони надають гнучку й багаторазово використовувану програмну платформу з усією необхідною функціональністю для розробки грального застосунка, скорочуючи витрати, складність і час розробки — усі критичні фактори в сильно конкурентній індустрії відеоігор.

Деякі ігрові рушії надають тільки можливості 3D рендерингу в реальному часі замість усієї функціональності, необхідної іграм. Ці рушії довіряють розробнику гри реалізацію іншої функціональності або її складання на основі інших гральних компонентів ППЗ. Такі типи рушіїв зазвичай відносять до “графічних рушіїв”, “рушіїв рендерингу” або “3D рушіїв” замість терміну “ігровий рушій”. Однак ця термінологія використовується суперечливо: багато повнофункціональних гральних 3D рушіїв згадані просто як “3D рушії”.

Найчастіше 3D рушії або системи рендерингу в ігрових рушіях побудовані на графічному API, такому як Direct3D або OpenGL, що забезпечує програмну абстракцію GPU або відеокарти. Низькорівневі бібліотеки, наприклад, DirectX, SDL і OpenAL, також використовуються в іграх, тому що забезпечують апаратно-незалежний доступ до іншого апаратного забезпечення комп’ютера, такого як пристрої введення (миша, клавіатура й джойстик), мережеві й звукові карти. До появи апаратно прискорюваної 3D графіки використовувалися програмні візуалізатори. Програмний рендеринг усе ще використовується в деяких інструментах моделювання для рендеринга зображень, для яких візуальна достовірність важливіша за продуктивність (кількість кадрів за секунду) або коли апаратне забезпечення комп'ютера не задовольняє вимоги, наприклад, не підтримує шейдери.

Розділ 4. Підготовча частина

4.1 Постановка задачі

Темою моєї дипломної роботи є “Розробка мультиплеєрної гри”. Для розробки гри напишемо власний ігровий рушій без використання сучасних технологій та бібліотек які спрощують написання гри. На основі нього створимо мультиплеєрну гру у жанрі шутер. Гра буде написана в консолі Windows, в якій буде створено ілюзію 3D.

Основною ідеєю у створенні гри було добавлення території, по якій може рухатись гравець. Перед гравцем добавлено перешкоди, стіни, підлогу, дзеркала, на які накладаються текстури для реалізації ігрового світу. У гравця є певна кількість здоров'я, швидкість пересування, а також зброя, з якої можна зробити постріли. І для того, щоб гра була повноцінною в жанрі шутер, реалізуємо можливість грати по локальній мережі з іншим гравцем один-на-один або команда на команду.

Для написання ігрового рушія і самої гри будемо використовувати:

- метод кидання променів (Ray casting);
- алгоритм художника;
- мову програмування С++ у середовищі Visual Studio.

Мета проекту - написання власного ігрового рушія, на основі якого буде створено мультиплеєрну гру, в якій гравці зможуть змагатись між собою в жанрі шутер.

4.2 Вивчення літератури

Для виконання магістерської роботи використано книги **Ray Tracing: A Tool for All** і **Комп'ютерна графіка. Моделі і алгоритми**, а також різні навчальні матеріали у мережі Інтернет.

Книга **Ray Tracing** автора Jon Peddie описує алгоритм "Кидання променів" немов би "з очей" глядача, крізь кожен піксел екрана, і знаходження найближчого об'єкта, що блокує хід такого променя. Термін "кидання променів" був вперше використаний у комп'ютерній графіці в 1982 році в роботі Скотта Рота, який застосував його для опису методу рендерингу CSG-моделей.

У книзі **Комп'ютерна графіка. Моделі і алгоритми** автора Євгенія Нікуліна детально викладаються математичні та алгоритмічні основи сучасної комп'ютерної графіки: моделі графічних об'єктів на площині і в просторі (точки, вектори, лінії і поверхні включаючи складові, поліедри, суцільні), геометричні завдання візуалізації - комплекс алгоритмів 2D- і 3D - відсікання і видалення, алгоритми афінних і проективних перетворень, методи зображення поверхонь, включаючи текстурування.

Розділ 5. Практична частина

Формування каркаса гри

Основою буде є клас “World”, в якому будуть зберігатися всі об'єкти на карті. Важливо відзначити той факт, що всі об'єкти являють собою набір з точок в двовимірному просторі. Тобто реально ніякого 3D не буде, але далі створимо ілюзію тривимірного зображення і гравцеві буде здаватися, що він бігає по тривимірній карті.

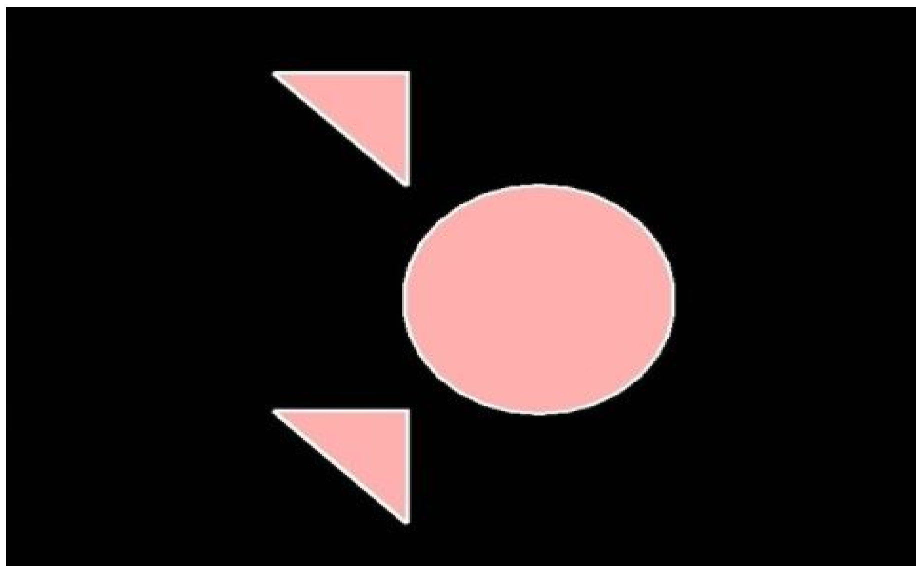


Рис. 3

По суті клас “World” готовий і тепер можна створити невелику 2D сцену з декількох об'єктів. Добре б також додати на сцену камеру і прописати управління з клавіатури.

2D карта, камера і управління

Камера повинна відображатися так, щоб було видно область огляду. Тобто потрібно дати камері ліхтар, який буде світити в напрямку погляду.

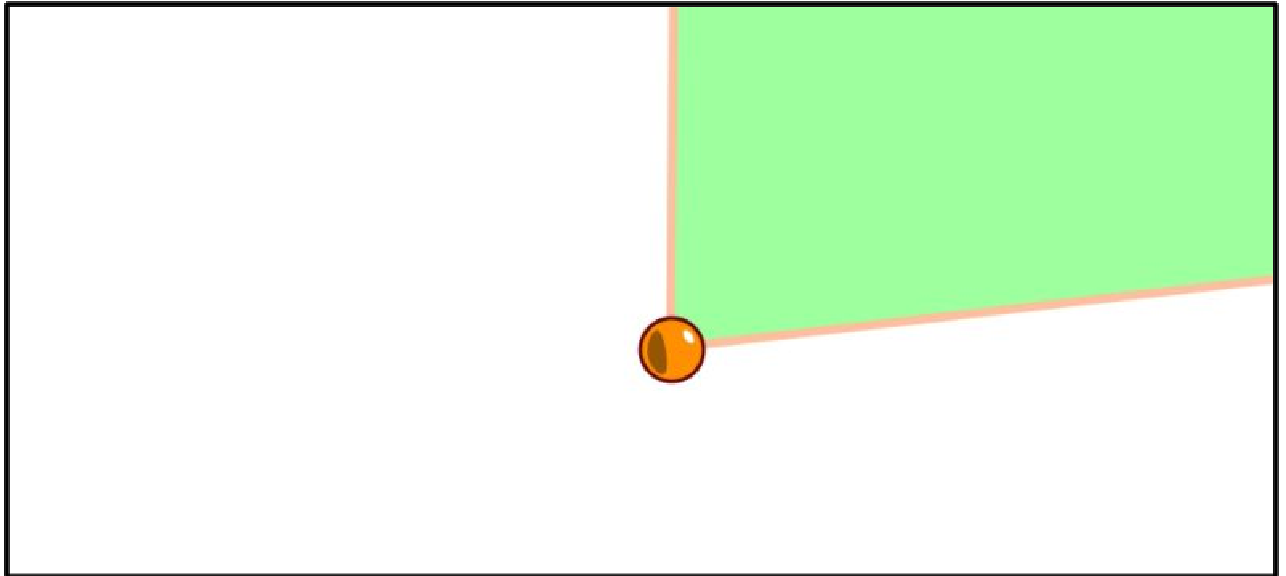


Рис. 4

Клавішами вперед і назад камера буде пересуватися вздовж напрямку погляду, а клавіші “вліво” і “вправо” будуть повертати гравця.

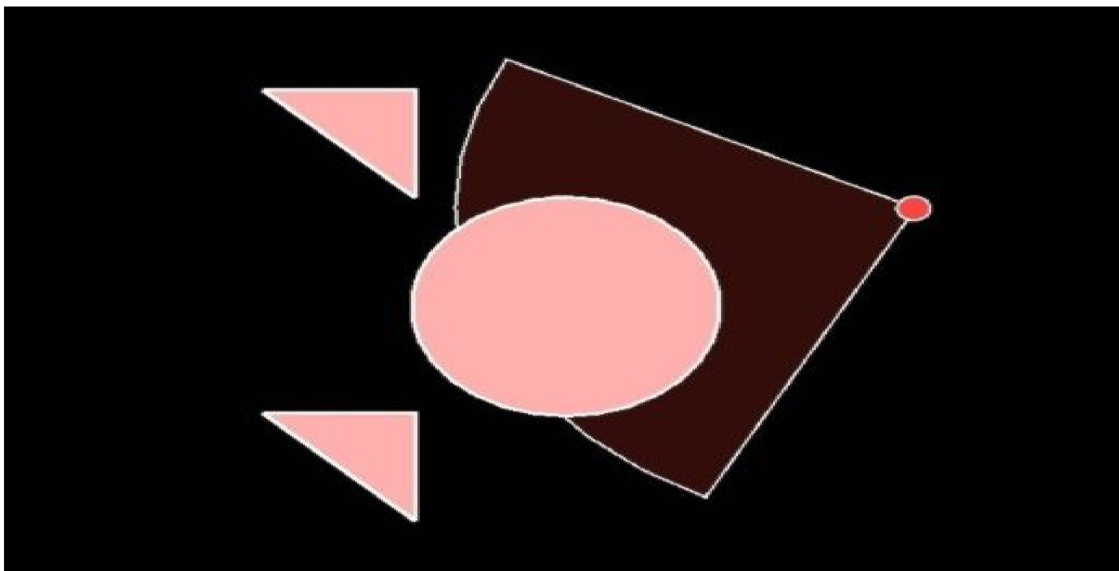


Рис. 5

Ray-cast і отримання 3D зображення

Тепер найголовніше - рушій гри. Потрібно зробити так, щоб камера могла малювати зображення на екран, і щоб це виглядало, як справжнє 3D. Для цього ми будемо використовувати алгоритм ray-cast.

Його суть полягає в наступному: Гравець пускає промінь до стіни. Якщо ми знайдемо точку перетину променя і перешкоди, то зможемо визначити відстань від камери до стінки. Так от якщо стінка близько, то ми в цьому напрямку намалюємо велику смужку, а якщо стінка далеко, то і смужка повинна бути маленькою.

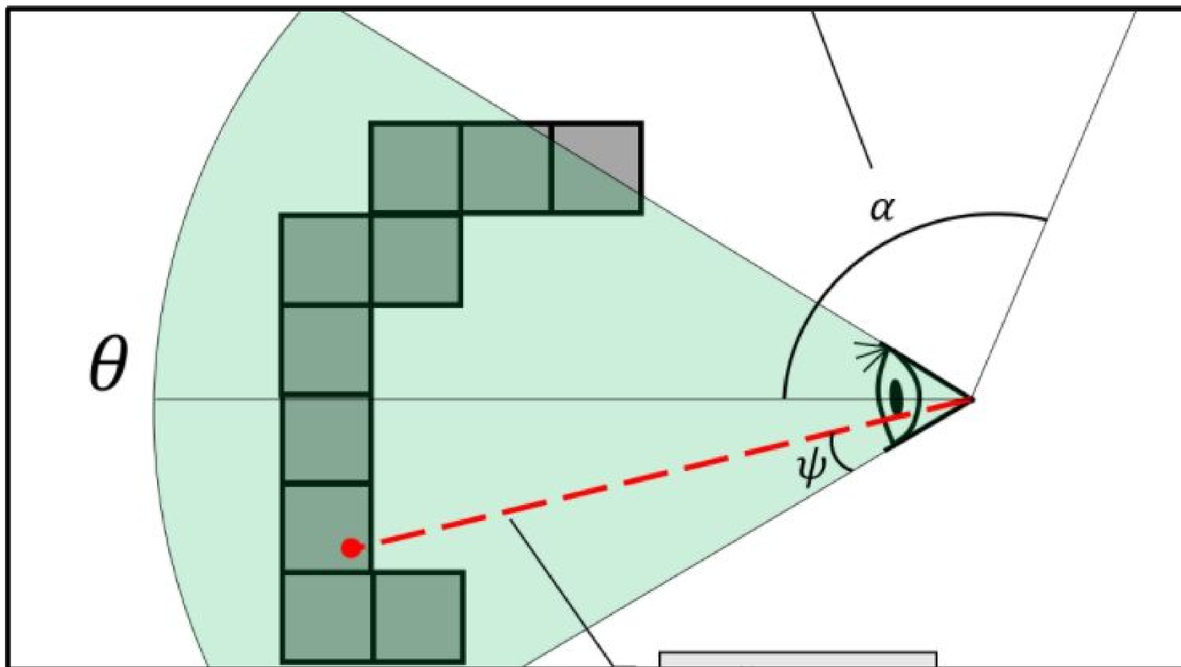


Рис. 6

Відстань до стіни будемо шукати ітераційно: на початку ми маємо напрямок, в якому хочемо перевірити наявність перешкоди, після цього ми циклічно, невеликими кроками йдемо в цьому напрямку. Зрештою можливо два результати: або промінь зіткнувся з перешкодою, або він відправився в нескінченну подорож.

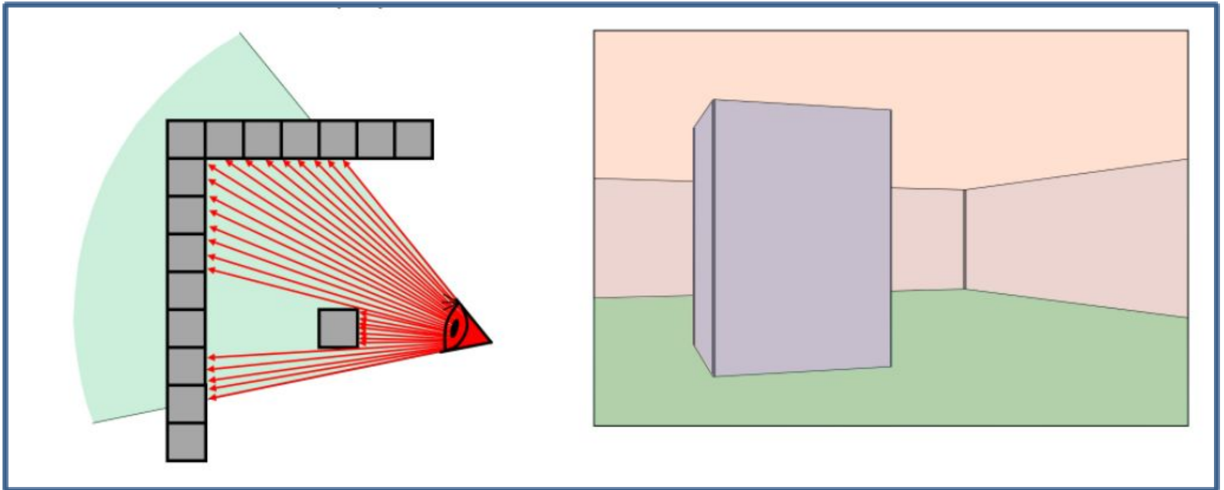


Рис. 7

Щоб не потрапляти в нескінченний цикл, потрібно обмежити радіус видимості. Якщо стінка не потрапила в радіус видимості, то ми будемо сприймати це як відсутність перешкоди в даному напрямку.

Виглядає наша програма вже непогано, але орієнтуватися по карті все одно важко. Добре б додати відображення ребер стін. Але як ми зрозуміємо, що дивимося саме на ребро? Якщо ми визначили, що відбулося зіткнення зі стіною, то ми відразу ж можемо (точно) визначити її місце розташування, а значить і місце розташування її чотирьох ребер.

Тобто у нас є чотири вектори, які спрямовані від спостерігача точно в ребра стінки. Якщо кут між випущеним променем і одним з цих векторів стає маленьким, то ми будемо сприймати цю частину стінки за ребро і малювати її іншим символом.

Настільки простий алгоритм дає можливість створити ілюзію реального 3D.

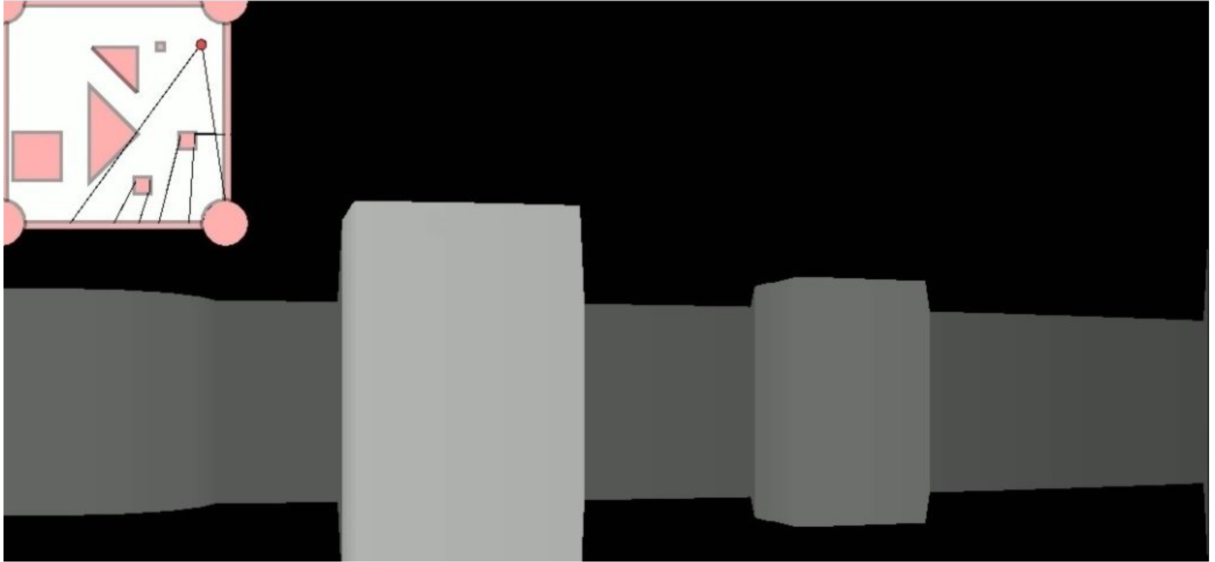


Рис. 8

Текстурування

Добре б тепер зробити так, щоб стінки могли мати якусь текстуру. Припустимо, ми маємо текстуру стіни. Тепер ми будемо шматувати її на вертикальні смужки і кожну з цих смужок накладаємо на відповідну смужку стіни.

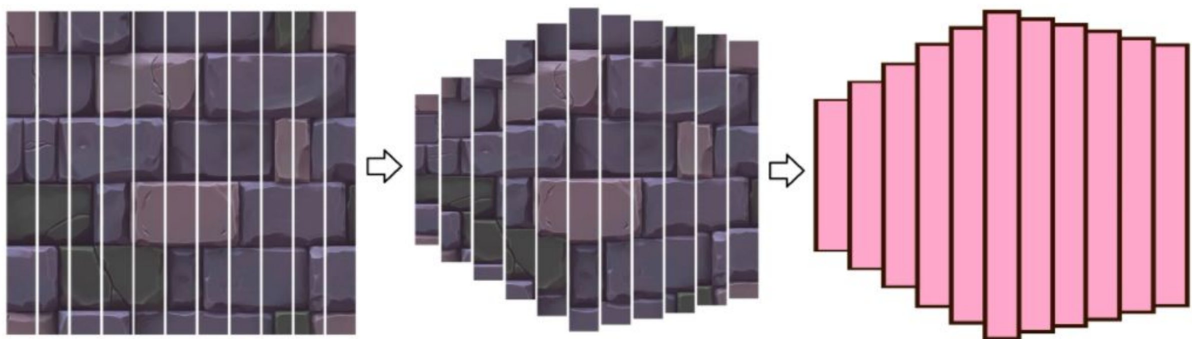


Рис. 9

В принципі, це і є весь алгоритм. Дуже простий і зрозумілий. Тепер кожен об'єкт має свою текстуру як властивість і кожна точка на поверхні об'єкту тепер відповідає конкретному зрушенню в структурі.

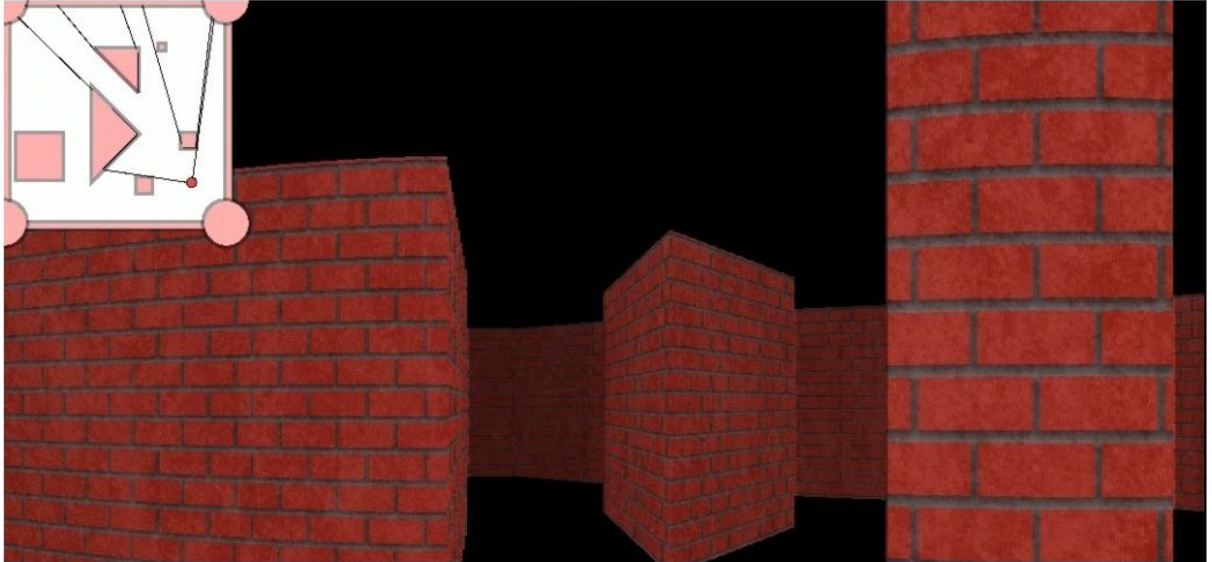


Рис.10

Природно, більш далекі ділянки текстури ми будемо малювати темнішими. Тепер можна знайти, наприклад, текстуру колони і застосувати її для бічних колон.

Тут потрібно звернути увагу на те, що при моєму підході, малюючи вертикальні смужки, ми не зможемо зробити красиву підлогу. Небо можна зробити, просто малюючи текстуру, яку ми будемо рухати при повороті камери. У підсумку вийде ефект паралакса і обсягу, а підлогу я вирішив поки що просто зробити однотонною.

Важливо, щоб текстура неба була безшовна, тобто при її склейці не повинно бути розривів. Тоді початок і кінець без видимих смуг буде з'єднуватися один з одним.

Зброя

Який шутер може обійтися без зброї? Спочатку напишемо клас "Weapon", що містить всю необхідну логіку, на зразок пострілів, підрахунку кількості патронів і швидкості перезарядки. Клас "Camera" (тобто гравець) буде містити в собі масив зброї та індекс обраної зброї.

Рис.11

Також було намальовано просту текстуру прицілу, яку потрібно буде просто помістити в центр екрану. Природно, у різних видів зброї буде різний приціл.

Щоб постріл виглядав красиво, потрібно додати анімацію перезарядки. Для цього в фотошопі я розрізав текстуру зброї на три складові, одну з яких трохи домалював. Отже, рука буде завжди знаходитися над стволом, а приклад і рушниця над рукою. Переміщаючи руку, можна створити ілюзію перезарядки. Швидкість стрільби зброї можна буде міняти (наприклад, прокачувати спритність гравця) і тому швидкість анімації теж буде змінюватися.

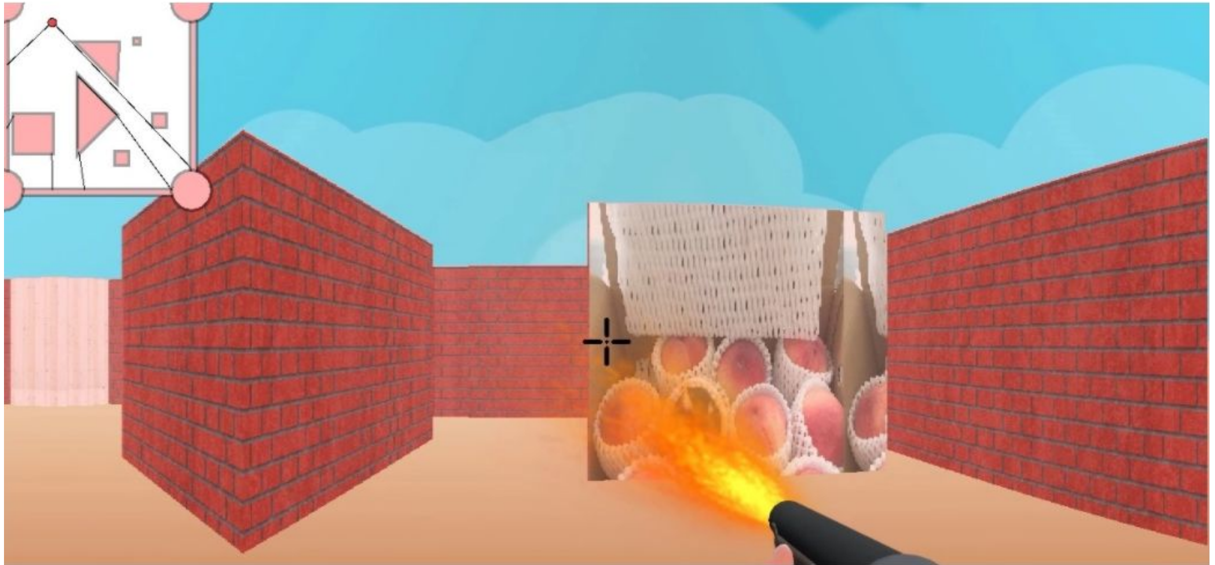


Рис.12

Колізія камери зі стінками

Додамо колізію камери зі стінами, щоб людина не могла ходити крізь перешкоди.

Тепер замість того, щоб кидати промені тільки в напрямку погляду кидаємо їх у всі боки. Навіщо це потрібно? Якщо виявляємо об'єкти, які потенційно можуть зазнати колізію з гравцем, то додаємо їх в масив «collision» потім при зсуві проходимо по всім потенційним колізіях.

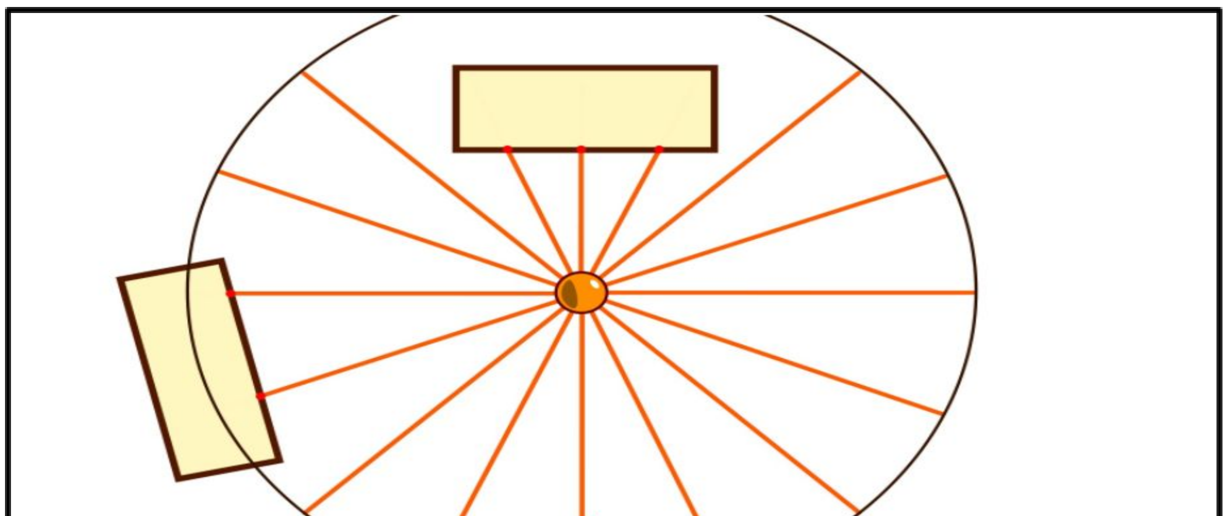


Рис.13

Для кожної колізії обчислюється проекція переміщення на вектор нормалі до поверхні перешкоди. Якщо ця проекція виявилась більшою, ніж розмір гравця, то трохи зменшуємо її і зберігаємо підсумковий вектор переміщення. Так робимо для кожної потенційної колізії і в кінці кінців вектор переміщення може дуже сильно змінитися.

Процедура більш ресурсномістка, але зате дуже гнучка і в теорії дозволить гравцеві ковзати уздовж стіни. Але тепер потрібно задавати точки для стін строго за годинниковою стрілкою. Напрям визначається вектором нормалі. Неправильно задані нормалі приведуть до того, що ми зможемо потрапити всередину стіни, а вийти вже не зможемо.

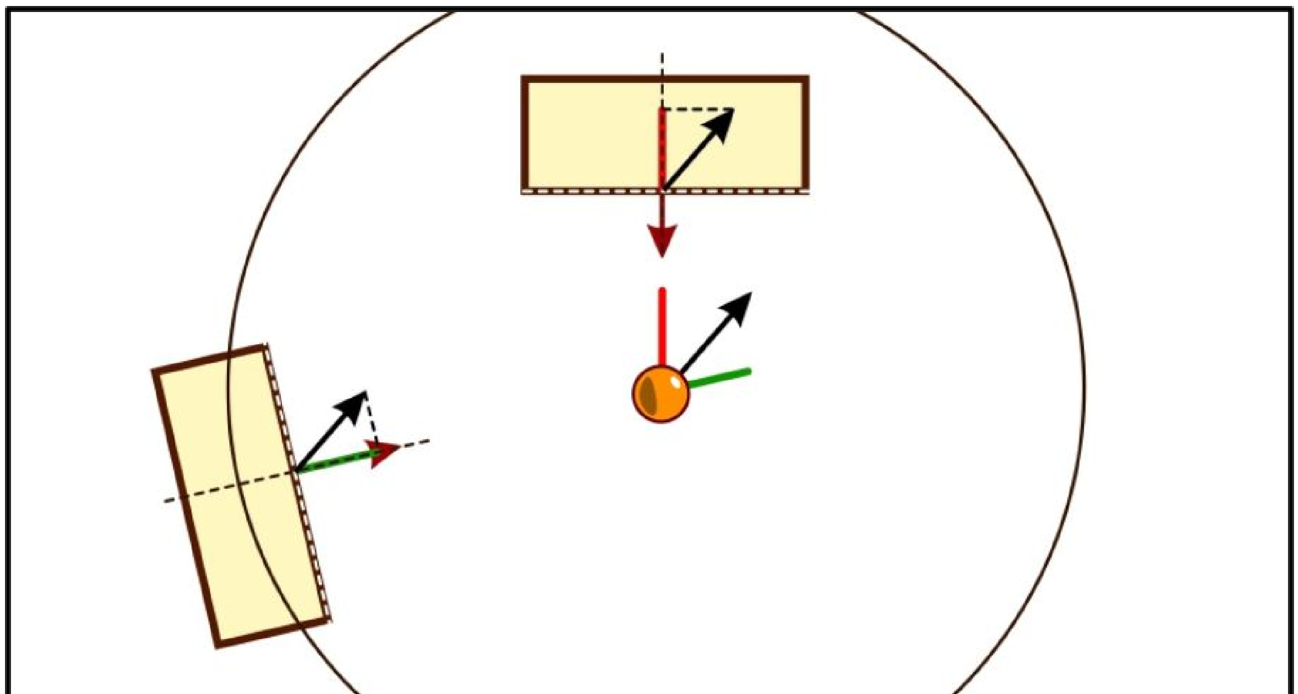


Рис.14

Дзеркала і стіни різної висоти

Ray-cast дозволяє зробити дзеркало. Якщо стінка є дзеркалом, то потрібно просто відобразити промінь і рекурсивно продовжити обчислення.

Єдиний нюанс, який потрібно зрозуміти - це як правильно відобразити промінь від даної поверхні з відомим вектором нормалі.

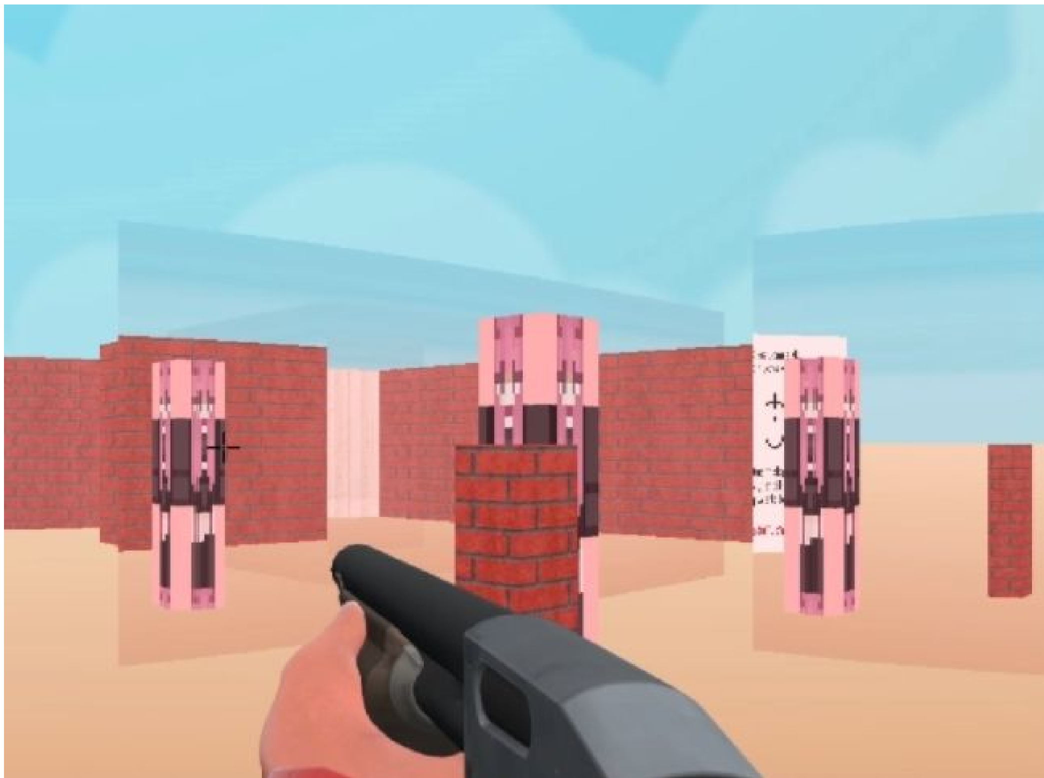


Рис.15

Для того, щоб додати стінки різної висоти, потрібно використати ray-cast не тільки до найближчого предмета, а й для всіх предметів в даному напрямку і зберігати всі колізії в вектор. Тепер зберігаємо всі відстані до всіх об'єктів, які потрапили в радіус видимості.

Обробка пострілів

Для того, щоб зробити обробку пострілів, потрібно після пострілу пускати промінь в напрямку погляду. Якщо цей промінь перетинає якусь камеру, то ми зменшуємо у неї кількість здоров'я.

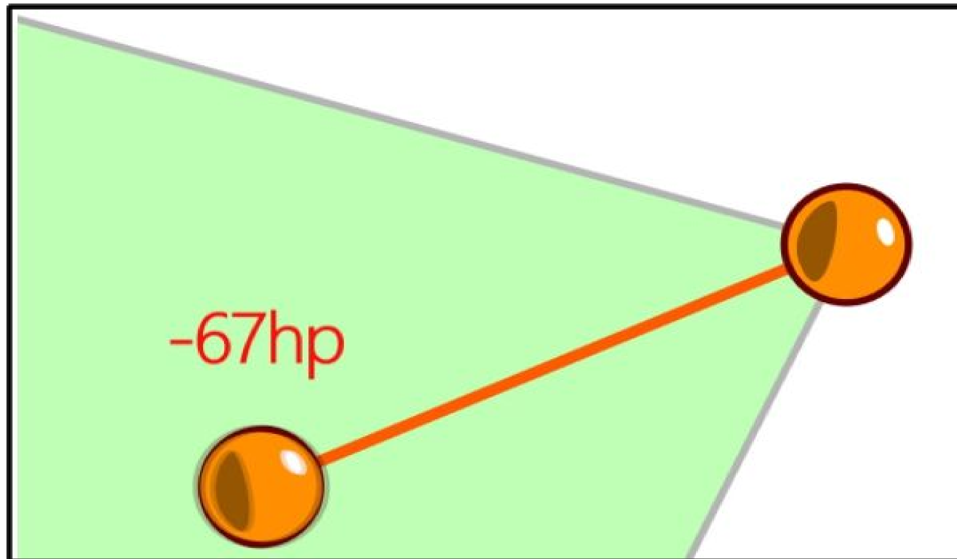


Рис.16

Але при зустрічі на шляху дзеркала промінь рекурсивно відбивається. Якщо використати ту ж функцію, то і кулі також будуть відбиватися від дзеркал.

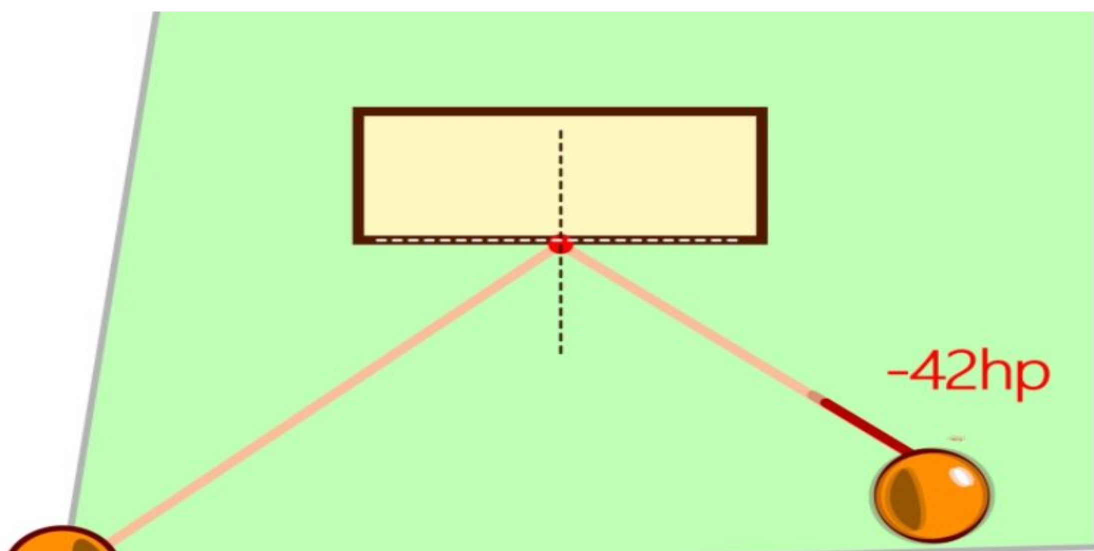


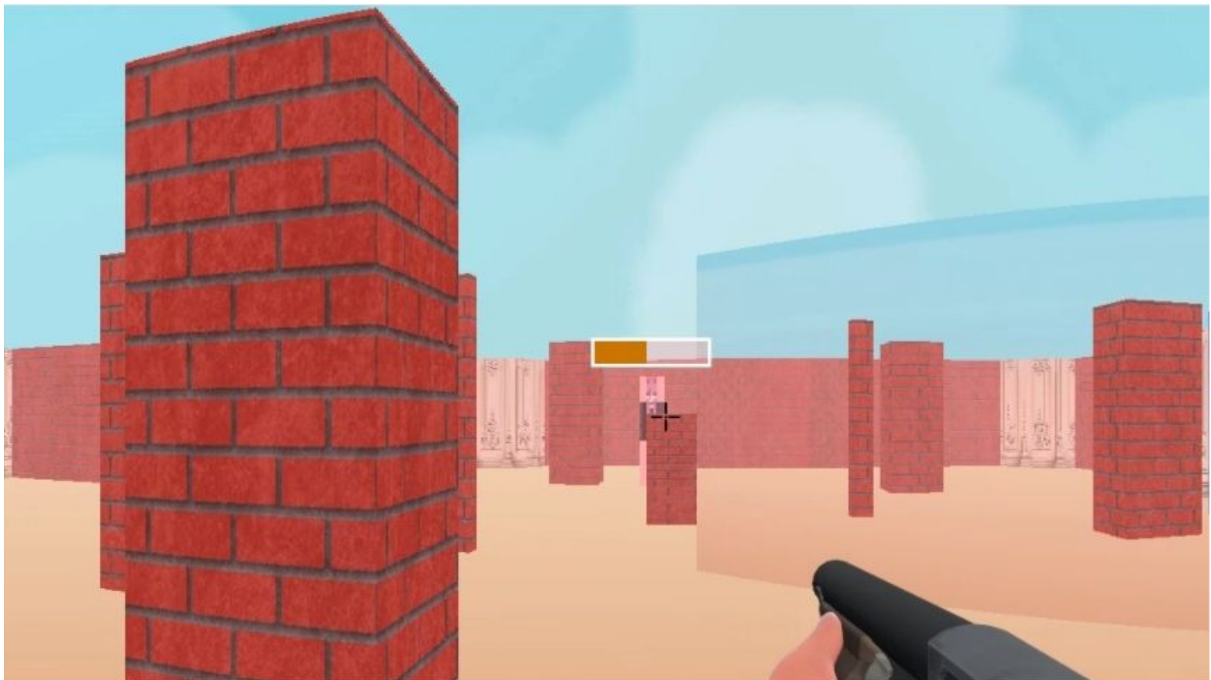
Рис.17

Це вводить нову цікаву механіку в гру.Якщо тепер користуватися дзеркалами, можна вбивати ворогів, стріляючи прямо в задзеркаллі. Втрати від куль залежать від відстані до цілі. Тобто якщо стояти впритул, то ворог відразу гине, а якщо стріляти здалеку або, наприклад, через дзеркало, шкоди буде куди менше.

Multiplayer

По суті, зараз вже можна грати, але проблема в тому, що ні з ким. Додамо можливість грати декільком гравцям на одній карті. У SFML є вбудовані сокети, що дозволяють обмінюватися повідомленнями між клієнтами по протоколу TCP або UDP.

Додавши смужку здоров'я, щоб гравець розумів, як він близький до поразки і зробив так, щоб швидкість нормувалося на повне здоров'я. Тобто тепер, коли гравцеві завдають шкоди, він не може швидко пересуватися.



Висновки

У роботі створено мультиплеєрну гру з використанням мови програмування C++ в середовищі Visual Studio. Для створення відеогри було написано ігровий рушій з використання алгоритму кидання променів. Даний метод зручно використовувати для рендерингу території, по якій гравець буде рухатися, текстури (стіни, підлога, дзеркала, перешкоди) і іншу функції для відображення 3D гри. Також на основі даного рушія написано мультиплеєрну відеогру в жанрі шутер. В цій грі можна змагатися один-на-один або команда на команду між собою, використовуючи зброю для перемоги.

Література та використані джерела

[1] https://habr.com/ru/post/187720

[2] https://litportal.ru/avtory/evgeniy-nikulin/kniga-kompyuternaya-geometriya-i-algoritmy-mashinnoy-grafiki-688211

[3] https://www.youtube.com

[4] https://visualstudio.microsoft.com

Додатки

Settings

```
#include <SFML/Graphics.hpp>

#define PI 3.14159265358979323846264338327950

#define SCALE 40 //How much pixels = 1 unit of distance measure

#define SCALE_WINDOW 500


// OBJECTS
#define OUTLINE_THICKNESS 3
#define OUTLINE_COLOR {0, 0, 0, 100}
#define FILL_COLOR {255, 174, 174, 255}

#define CIRCLE_CONVEX_NUMBER 10

// CAMERA
#define OUTLINE_CAMERA_THICKNESS 1
#define OUTLINE_CAMERA_COLOR {0, 0, 0, 255}
#define FILL_CAMERA_COLOR {255, 67, 67, 255}
#define RADIUS_CAMERA SCALE/10
#define CONVEX_NUMBER 100
#define FILED_OF_VEW_COLOR {255, 67, 67, 0}
#define RED_COLOR {255, 0, 0}
#define SKIN "texture/hero.png"

// FOR COLLISION DETECTION
#define COLLISION_DISTANCE 0.1
#define COLLISION_AREA 1

#define OPTICAL_HEIGHT 100

//RENDER SETTINGS

#define FILL_RENDER_COLOR {255, 255, 255, 255}

// SCREEN SIZE
#define SCREEN_WIDTH 1280
#define SCREEN_HEIGHT 720

#define DISTANCES_SEGMENTS 640 // How much vertical sectors we use in ray cast

//BACKGROUND
#define OUTLINE_BACK_THICKNESS 5
#define OUTLINE_BACK_COLOR {255, 255, 255, 255}
#define FILL_BACK_COLOR {255, 255, 255, 255}

//TEXTURES

#define WALL_TEXTURE "texture/brickWall.jpg"
```

```
#define WALL_TEXTURE2 "texture/brickWall2.jpg"
#define COLUMN_TEXTURE "texture/column2.jpg"
#define VOLODYA_TEXTURE "texture/volodya.jpg"
#define FROZEN_TEXTURE "texture/frozen.jpg"
#define DOG_TEXTURE "texture/dog.jpg"
#define DINOZAVR_TEXTURE "texture/dinozavr.png"
#define WHITE_TEXTURE "texture/white.jpg"
#define BLACK_TEXTURE "texture/black.jpg"

#define SKY_TEXTURE "texture/back.jpg"
#define FLOOR_TEXTURE "texture/floor.jpg"

#define PLAYGAME_PRESSED_TEXTURE "texture/buttons/playGameSelected.jpg"
#define SETTINGS_PRESSED_TEXTURE "texture/buttons/settingsSelected.jpg"
#define ABOUT_PRESSED_TEXTURE "texture/buttons/aboutSelected.jpg"
#define QUIT_PRESSED_TEXTURE "texture/buttons/quitSelected.jpg"

#define PLAYGAME_TEXTURE "texture/buttons/playGame.jpg"
#define SETTINGS_TEXTURE "texture/buttons/settings.jpg"
#define ABOUT_TEXTURE "texture/buttons/about.jpg"
#define QUIT_TEXTURE "texture/buttons/quit.jpg"

#define ABOUT_INFO "texture/info.jpg"

#define TEXTURING_SELECT "texture/buttons/texturing.jpg"
#define SMOOTHING_SELECT "texture/buttons/smoothing.jpg"
#define COLLISION_SELECT "texture/buttons/collosion.jpg"

#define TEXTURING_SELECT_S "texture/buttons/texturingSelected.jpg"
#define SMOOTHING_SELECT_S "texture/buttons/smoothingSelected.jpg"
#define COLLISION_SELECT_S "texture/buttons/collosionSelected.jpg"

#define INSTRUCTION_TEXTURE "texture/instruction.jpg"

// WINDOWS

#define TIMEQUANTA 0.1

// WEAPONS

#define SHOTGUN_TEXTURE "texture/weapons/shotgun.png"
#define SHOTGUN_ARM_TEXTURE "texture/weapons/shotgun_arm.png"
#define SHOTGUN_HANDLE_TEXTURE "texture/weapons/shotgun_handle.png"
#define SHOTGUN_TRUNK_TEXTURE "texture/weapons/shotgun_trunk.png"

#define AIM_TEXTURE "texture/weapons/aim.png"
#define FIRE_SHOTGUN_TEXTURE "texture/weapons/gunBlast.png"

#define GUN_SHOT_SOUND "sounds/gunShot.ogg"
#define WALK_SOUND "sounds/walkSound.ogg"
```

World.h

```
#include "Object2D.h"
#include <string>
#include <map>

class World : virtual public Idrawable {
private:
    mutable std::map<std::string, Object2D&> map_objects;

    double d_length = 0;
    double d_width = 0;

    sf::Texture T_sky_texture;
    sf::Texture T_floor_texture;
    mutable bool sky_texture_loaded = false;
    mutable bool floor_texture_loaded = false;
    mutable std::string s_sky_texture;
    mutable std::string s_floor_texture;
public:
    World(double length, double width, std::string sky_texture = SKY_TEXTURE, std::string
floor_texture = FLOOR_TEXTURE) : d_length(length), d_width(width),
s_sky_texture(sky_texture), s_floor_texture(floor_texture) {}

    bool addObject2D(Object2D&  object, std::string name) {
        object.setName(name);
        return map_objects.insert({name, object}).second;
    }
    Object2D& findObject2D(std::string name) { return map_objects.at(name); }
    Object2D& findObject2D(std::string name) const { return map_objects.at(name); }
    bool isExist(std::string name) const {return map_objects.count(name) != 0; }

    bool removeObject2D(std::string name) { return map_objects.erase(name) > 0; }

    Object2D& operator[](std::string name) {
        return findObject2D(name);
    }

    Object2D& operator[](std::string name) const {
        return findObject2D(name);
    }

    double width() const { return d_width; }
    double length() const { return d_length; }

    void draw(sf::RenderWindow& window) override;

    const std::map<std::string, Object2D&>& objects() const { return map_objects; }

    const sf::Texture& skyTexture() {
        if(sky_texture_loaded) return T_sky_texture;
        sky_texture_loaded = true;
        if (!T_sky_texture.loadFromFile(s_sky_texture)) {
            sky_texture_loaded = false;
        }
        T_sky_texture.setRepeated(true);
```

```
        return T_sky_texture;
    }

    const sf::Texture& floorTexture() {
        if(floor_texture_loaded) return T_floor_texture;
        floor_texture_loaded = true;
        if (!T_floor_texture.loadFromFile(s_floor_texture)) {
            floor_texture_loaded = false;
        }
        T_floor_texture.setRepeated(true);
        return T_floor_texture;
    }
};
```

World.cpp

```
#include "World.h"
#include "settings.h"

void World::draw(sf::RenderWindow &window) {

    sf::ConvexShape background;
    background.setPointCount(4);
    background.setPoint(0, sf::Vector2f(0, 0));
    background.setPoint(1, sf::Vector2f(SCALE_WINDOW*SCALE/100, 0));
    background.setPoint(2, sf::Vector2f(SCALE_WINDOW*SCALE/100, SCALE_WINDOW*SCALE/100));
    background.setPoint(3, sf::Vector2f(0, SCALE_WINDOW*SCALE/100));
    background.setOutlineColor(OUTLINE_BACK_COLOR);
    background.setFillColor(FILL_BACK_COLOR);
    background.setOutlineThickness(OUTLINE_BACK_THICKNESS);
    background.setPosition(0, 0);
    window.draw(background);

    for(auto object : map_objects) {
        object.second.draw(window);
    }
}
```

Weapon.h

```
#include <SFML/Audio.hpp>
#include <chrono>
#include "settings.h"
#include "Object2D.h"


class Weapon : virtual public Idrawable {
private:
    double d_elapsedTime;

    sf::Texture T_weapon_arm;
    sf::Sprite S_weapon_arm;
    sf::Texture T_weapon_handle;
    sf::Sprite S_weapon_handle;
    sf::Texture T_weapon_trunk;
    sf::Sprite S_weapon_trunk;

    sf::Sprite S_aim;
    sf::Texture T_aim;
    sf::Sprite S_fire;
    sf::Texture T_fire;

    sf::SoundBuffer fireSoundBuffer;
    sf::Sound fireSound;

    double d_speed = 1;
    int i_amount = 0;
    int i_damage = 0;

    double fireShift = 0;
    double d_amplitude = 100;
    double d_fireAnimTime = 0;

    double d_lastFireTime = 0;
public:
    Weapon(int amount) : i_amount(amount) { };

    Weapon(const Weapon& weapon);

    void choiceWeapon(std::string name);

    bool fire();

    double speed() const { return d_speed; }

    void draw(sf::RenderWindow& window) override;

    int damage() const { return i_damage; }
};
```

Weapon.cpp

```
#include "Weapon.h"


Weapon::Weapon(const Weapon &weapon) {
    this->i_amount = weapon.i_amount;
    this->d_speed = weapon.d_speed;

    this->S_weapon_arm = weapon.S_weapon_arm;
    this->T_weapon_arm = weapon.T_weapon_arm;
    this->S_weapon_handle = weapon.S_weapon_handle;
    this->T_weapon_handle = weapon.T_weapon_handle;
    this->S_weapon_trunk = weapon.S_weapon_trunk;
    this->T_weapon_trunk = weapon.T_weapon_trunk;

    this->S_aim = weapon.S_aim;
    this->T_aim = weapon.T_aim;
    this->S_fire = weapon.S_fire;
    this->T_fire = weapon.T_fire;

    this->fireSoundBuffer = weapon.fireSoundBuffer;
    this->fireSound = weapon.fireSound;
    this->fireSound.setBuffer(this->fireSoundBuffer);

    this->d_amplitude = weapon.d_amplitude;

    this->i_damage = weapon.i_damage;

    S_weapon_handle.setTexture(T_weapon_handle);
}

void Weapon::choiceWeapon(std::string name) {
    if(name == "shotgun") {
        T_weapon_arm.loadFromFile(SHOTGUN_ARM_TEXTURE);
        T_weapon_handle.loadFromFile(SHOTGUN_HANDLE_TEXTURE);
        T_weapon_trunk.loadFromFile(SHOTGUN_TRUNK_TEXTURE);

        T_aim.loadFromFile(AIM_TEXTURE);
        T_fire.loadFromFile(FIRE_SHOTGUN_TEXTURE);
        d_speed = 0.4;
        S_aim.scale(.05, .05);

        fireSoundBuffer.loadFromFile(GUN_SHOT_SOUND);
        fireSound.setBuffer(fireSoundBuffer);
        fireSound.setVolume(10.f);
        i_damage = 50;
    }
}

bool Weapon::fire() {
    if((i_amount > 0) && (d_lastFireTime == 0)) {
        d_lastFireTime = d_speed;
        d_fireAnimTime = .1;
        --i_amount;
```

```
        fireShift = d_amplitude;
        fireSound.play();
        return true;
    }
    return false;
}

void Weapon::draw(sf::RenderWindow &window) {
    auto tp = std::chrono::system_clock::now();
    std::chrono::duration <double> elapsedTime = tp.time_since_epoch();
    double diff = elapsedTime.count() - d_elapsedTime;
    d_elapsedTime = elapsedTime.count();


    if(fireShift > 0)
        fireShift -= 2*d_amplitude*diff/d_speed;
    else
        fireShift = 0;
    double shift = 15*(1 + cos(3*d_elapsedTime));

    Point2D weaponPosition = {SCREEN_WIDTH - S_weapon_handle.getTextureRect().width + shift
+ fireShift, SCREEN_HEIGHT - S_weapon_handle.getTextureRect().height + shift + fireShift};

    // FIRE EFFECT HERE
    d_lastFireTime -= diff; // Time when you can't fire
    if(d_lastFireTime < 0)
        d_lastFireTime = 0;

    d_fireAnimTime -= diff;
    if(d_fireAnimTime > 0) {
        S_fire.setTexture(T_fire);
        S_fire.setPosition(sf::Vector2f(weaponPosition.x - 490, weaponPosition.y - 792));
        window.draw(S_fire);
    }

    // All about weapon

    double d_armShift = 50*sin(PI*2*d_lastFireTime/d_speed);
    if(d_lastFireTime <= 0 || d_lastFireTime > d_speed/2)
        d_armShift = 0;

    S_weapon_arm.setTexture(T_weapon_arm);
    S_weapon_handle.setTexture(T_weapon_handle);
    S_weapon_trunk.setTexture(T_weapon_trunk);
    S_weapon_arm.setPosition(sf::Vector2f(weaponPosition.x + d_armShift, weaponPosition.y +
d_armShift)); // Shift arm when we shoot
    S_weapon_handle.setPosition(sf::Vector2f(weaponPosition.x, weaponPosition.y));
    S_weapon_trunk.setPosition(sf::Vector2f(weaponPosition.x, weaponPosition.y));

    window.draw(S_weapon_trunk);
    window.draw(S_weapon_arm);
    window.draw(S_weapon_handle);

    S_aim.setTexture(T_aim);
    S_aim.setPosition(sf::Vector2f(SCREEN_WIDTH/2 - S_aim.getTextureRect().width/50,
SCREEN_HEIGHT/2 - S_aim.getTextureRect().height/50)); // абсолютная позиция
    window.draw(S_aim);
```

```
}
```

Object2D.h

```
#include <utility>
#include <vector>
#include <cmath>
#include "Idrawable.h"
#include "settings.h"


struct Point2D {
    double x = 0;
    double y = 0;

    Point2D& operator+=(const Point2D& point2D) { this->x += point2D.x; this->y +=
point2D.y; }
    Point2D& operator=(const Point2D& point2D) { this->x = point2D.x; this->y = point2D.y;
return *this; }
    Point2D& operator*(double number) { this->x *= number; this->y *= number; }
    double   operator*(const Point2D& point2D) { return x*point2D.x + y*point2D.y; }
    Point2D operator-(const Point2D& point2D) const { return {this->x - point2D.x, this->y
- point2D.y}; }
    Point2D operator+(const Point2D& point2D) const { return {this->x + point2D.x, this->y
+ point2D.y}; }

    Point2D normalize() { return Point2D{this->x/abs(), this->y/abs()};}
    double abs() {return sqrt(x*x + y*y); }
};

class Object2D : virtual public Idrawable {
protected:
    std::vector<Point2D> v_points2D;

    Point2D p_position;
    Point2D p_velocity;

    std::string s_name;
    double d_height = 1;
    bool b_isMirror = false;

    sf::Texture T_texture;
    std::string s_texture;
    bool texture_loaded = false;
public:
    Object2D(Point2D position = {0, 0}, std::vector<Point2D> points = {}, double height =
1, std::string texture = WALL_TEXTURE, Point2D velocity = {0, 0})
    : p_position(position), v_points2D(points), p_velocity(velocity),
s_texture(std::move(texture)), d_height(height) {}

    Object2D(const Object2D& object2D) { // copy constructor
        v_points2D = object2D.v_points2D;
        p_position = object2D.p_position;
        p_velocity = object2D.p_velocity;
        s_name = object2D.s_name;
        d_height = object2D.d_height;
        b_isMirror = object2D.b_isMirror;
```

```
        T_texture = object2D.T_texture;
        s_texture = object2D.s_texture;
        texture_loaded = object2D.texture_loaded;
    }

    const double x() { return p_position.x; }
    const double y() { return p_position.y; }

    void setPosition(Point2D position) { p_position = position; };

    void shift(Point2D vector) { p_position += vector; }

    std::vector<Point2D>& nodes() { return v_points2D; }

    static bool segments_crossing(std::pair<Point2D, Point2D> segment1, std::pair<Point2D,
Point2D> segment2, Point2D& point);

    void setName(std::string name) {s_name = name;}
    std::string getName() const { return s_name; }

    Point2D position() const { return p_position; }

    bool isMirror () const { return b_isMirror; }
    void makeMirror() { b_isMirror = true; }
    void makeItNotMirror() {b_isMirror = false; }

    void draw(sf::RenderWindow& window) override;

    double height() const { return d_height; }

    sf::Texture& loadTexture() {
        if(texture_loaded) return T_texture;
        texture_loaded = true;
        if (!T_texture.loadFromFile(s_texture)) {
            texture_loaded = false;
        }
        T_texture.setRepeated(true);
        return T_texture;
    }

    virtual int type() { return 0; }
    void setPoints2D (std::vector<Point2D> points2D) {
        v_points2D = std::move(points2D);
    }
};
```

Object2D.cpp

```
#include "Object2D.h"
#include "settings.h"

int sign(double number) { return number >= 0 ? 1 : -1; }

double cross(Point2D p1, Point2D p2) {
    return p1.x*p2.y - p1.y*p2.x;
```

```
}

bool Object2D::segments_crossing(std::pair<Point2D, Point2D> segment1, std::pair<Point2D,
Point2D> segment2, Point2D& point)
{
    // {v11 = segment1.first, v12 = segment1.second}, {v21 = segment2.first, v22 =
segment2.second}
  Point2D cut1 = segment1.second - segment1.first;
  Point2D cut2 = segment2.second - segment2.first;
  double prod1;
  double prod2;

  prod1 = cross(cut1, (segment2.first-segment1.first));
  prod2 = cross(cut1, (segment2.second-segment1.first));

   if(sign(prod1) == sign(prod2) || (prod1 == 0) || (prod2 == 0)) // Отсекаем
также и пограничные случаи
     return false;

   prod1 = cross(cut2, (segment1.first - segment2.first));
   prod2 = cross(cut2, (segment1.second - segment2.first));

   if(sign(prod1) == sign(prod2) || (prod1 == 0) || (prod2 == 0)) // Отсекаем
также и пограничные случаи
     return false;

   point.x = segment1.first.x + cut1.x*std::abs(prod1)/std::abs(prod2-prod1);
   point.y = segment1.first.y + cut1.y*std::abs(prod1)/std::abs(prod2-prod1);

   return true;
 }

void Object2D::draw(sf::RenderWindow& window) {
    sf::ConvexShape polygon;
    polygon.setPointCount(nodes().size());
    int i = 0;
    for(auto p : nodes()) {
        polygon.setPoint(i++, sf::Vector2f(p.x*SCALE, p.y*SCALE));
    }
    polygon.setOutlineColor(OUTLINE_COLOR);
    polygon.setFillColor(FILL_COLOR);
    polygon.setOutlineThickness(OUTLINE_THICKNESS);
    polygon.setPosition((float)x()*SCALE, (float)y()*SCALE);
    window.draw(polygon);
}
```

Menu.h

```
#include <SFML/Audio.hpp>
#include <SFML/Graphics.hpp>
#include <map>
#include <iostream>
#include <chrono>
#include "settings.h"
#include "Poligon2D.h"
#include "Circle2D.h"
#include "Object2D.h"
#include "Camera.h"

struct Button {
    sf::Texture T_Texture;
    sf::Texture T_PressedTexture;
    sf::Sprite button;
    std::string name;

    bool selected = false;
    bool pressed = false;

    void select() {
        if(!selected && !pressed) {
            button.setTexture(T_PressedTexture);
            selected = true;
            button.scale({1.01, 1.01});
        }
    }
    void unSelect() {
        if(selected && !pressed) {
            button.setTexture(T_Texture);
            selected = false;
            button.scale({1/1.01, 1/1.01});
        }
    }

    void press() {
        if(!pressed) {
            button.setTexture(T_PressedTexture);
            pressed = true;
        } else {
            button.setTexture(T_Texture);
            pressed = false;
        }
    }
};

class Menu {
private:
    std::vector<Button> buttons;

    bool b_pause = true;

    bool b_settings = false;
```

```
    bool b_about = false;

    bool b_textures = true;
    bool smooth = false;
    bool b_collision = true;
    double lastTime = 0;
public:
    Menu();

    void drawMenu(sf::RenderWindow& window, double elapsedTime);

    void settings(sf::RenderWindow& window);

    void about(sf::RenderWindow& window);

    bool isPaused() { return b_pause; }
    void setPause() { b_pause = true; }

    bool isSmooth() { return smooth; }
    void switchSmooth() { smooth = !smooth; }
    bool isCollision() { return b_collision; }
    void switchCollision() { b_collision = !b_collision; }
    bool isTextures() { return b_textures; }
    void switchTextures() { b_textures = !b_textures; }
};
```

Menu.cpp

```
#include "Menu.h"


Menu::Menu() {

    buttons.assign(7, {});

    buttons[0].name = "PLAYGAME";
    buttons[1].name = "SETTINGS";
    buttons[2].name = "ABOUT";
    buttons[3].name = "QUIT";
    buttons[0].T_Texture.loadFromFile(PLAYGAME_TEXTURE);
    buttons[1].T_Texture.loadFromFile(SETTINGS_TEXTURE);
    buttons[2].T_Texture.loadFromFile(ABOUT_TEXTURE);
    buttons[3].T_Texture.loadFromFile(QUIT_TEXTURE);

    buttons[0].T_PressedTexture.loadFromFile(PLAYGAME_PRESSED_TEXTURE);
    buttons[1].T_PressedTexture.loadFromFile(SETTINGS_PRESSED_TEXTURE);
    buttons[2].T_PressedTexture.loadFromFile(ABOUT_PRESSED_TEXTURE);
    buttons[3].T_PressedTexture.loadFromFile(QUIT_PRESSED_TEXTURE);

    buttons[4].name = "TEXTURING";
    buttons[5].name = "SMOOTHING";
    buttons[6].name = "COLLISION";
    buttons[4].T_Texture.loadFromFile(TEXTURING_SELECT);
    buttons[5].T_Texture.loadFromFile(SMOOTHING_SELECT);
```

```
    buttons[6].T_Texture.loadFromFile(COLLISION_SELECT);

    buttons[4].T_PressedTexture.loadFromFile(TEXTURING_SELECT_S);
    buttons[5].T_PressedTexture.loadFromFile(SMOOTHING_SELECT_S);
    buttons[6].T_PressedTexture.loadFromFile(COLLISION_SELECT_S);

    buttons[4].press();

    for(int i = 0; i < buttons.size(); i++) {
        buttons[i].button.setTexture(buttons[i].T_Texture);
        if(i < 4)
            buttons[i].button.setPosition(SCREEN_WIDTH/2 - 170, 50 + 150*i);
        else
            buttons[i].button.setPosition(SCREEN_WIDTH/2 - 170, 50 + 150*(i-4));
    }
}

void Menu::drawMenu(sf::RenderWindow &window, double elapsedTime) {

    sf::ConvexShape back;
    back.setPointCount(4);
    back.setPoint(0, sf::Vector2f(0, 0));
    back.setPoint(1, sf::Vector2f(SCREEN_WIDTH, 0));
    back.setPoint(2, sf::Vector2f(SCREEN_WIDTH, SCREEN_HEIGHT));
    back.setPoint(3, sf::Vector2f(0, SCREEN_HEIGHT));
    back.setPosition({0, 0});
    back.setFillColor({255, 255, 255});
    window.draw(back);

    for(int i = 0; i < buttons.size(); i++) {

        if(buttons[i].button.getGlobalBounds().contains(sf::Mouse::getPosition(window).x,
sf::Mouse::getPosition(window).y)) {
            buttons[i].select();
            if (sf::Mouse::isButtonPressed(sf::Mouse::Left))
            {
                buttons[i].unSelect();
                if(!b_settings) {
                    if (buttons[i].name == "PLAYGAME") {
                        b_pause = false;
                    }
                    if (buttons[i].name == "SETTINGS") {
                        b_settings = true;
                    }
                    if (buttons[i].name == "ABOUT") {
                        b_about = true;
                    }
                    if (buttons[i].name == "QUIT") {
                        window.close();
                    }
                } else {
                    if (buttons[i].name == "TEXTURING") {
                        buttons[i].press();
                        b_textures = buttons[i].pressed;
                    }
                    if (buttons[i].name == "SMOOTHING") {
                        buttons[i].press();
```

```
                    smooth = buttons[i].pressed;
                }
                if (buttons[i].name == "COLLISION") {
                    buttons[i].press();
                    b_collision = buttons[i].pressed;
                }
            }
        }
    } else {
        buttons[i].unSelect();
    }


    if(!b_settings && !b_about && i < 4)
        window.draw(buttons[i].button);
}

settings(window);
about(window);
}

void Menu::about(sf::RenderWindow &window) {
    if(!b_about) return;

    sf::Texture T_Texture;
    T_Texture.loadFromFile(ABOUT_INFO);
    sf::Sprite button;
    button.setTexture(T_Texture);

    button.scale((double)SCREEN_WIDTH/1920, (double)SCREEN_WIDTH/1920);

    window.draw(button);
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Escape))
        b_about = false;
}

void Menu::settings(sf::RenderWindow &window) {
    if(!b_settings) return;

    for(int i = 4; i < buttons.size(); i++) {
        window.draw(buttons[i].button);
    }

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Escape))
        b_settings = false;
}
```

Camera.h

```
#include <algorithm>
#include "Object2D.h"
#include "World.h"
#include "settings.h"
#include "Menu.h"
#include "Weapon.h"

struct RGB {
    int RED = 255;
    int GREEN = 255;
    int BLUE = 255;
};

struct RayCastStructure {
    double distance;    // How far is this texture
    double progress;    // progress defines the point of texture we should load
    std::string object; // particular object. We need this to show particular texture.
    double height;      // objects has different height

    std::vector<RayCastStructure> v_mirrorRayCast; // When we have mirror, we should know
about all objects we can see
};

struct CollisionInformation {
    double distance;
    Point2D collisionPoint;
    std::pair<Point2D, Point2D> edge;
};

class Camera : public Circle2D {
private:
    std::vector<std::vector<RayCastStructure>> v_distances;

    //For collision detection
    std::vector<CollisionInformation> allCollisions;

    double d_direction;
    double d_fieldOfView;
    double d_depth;

    double d_walkSpeed;
    double d_viewSpeed;

    double i_health;

    bool b_collision = true;

    World& W_world;

    sf::Vector2i localMousePosition;

    bool b_textures = true;
    bool b_smooth = false;
```

```
    std::vector<Weapon> v_weapons;
    int i_selectedWeapon = 0;

    sf::SoundBuffer walkSoundBuffer;
    sf::Sound walkSound;

    std::string s_lastKill;

    void objectsRayCrossed(std::pair<Point2D, Point2D> ray, std::vector<RayCastStructure>&
v_rayCastStruct, const std::string& name, int reflections = 0);
    void drawVerticalStrip(sf::RenderWindow& window, const RayCastStructure& obj, int
shift, double s);
    void recursiveDrawing(sf::RenderWindow& window, const std::vector<RayCastStructure>&
v_RayCastStructure, int shift);
    static void recursiveIncreaseDistance(std::vector<RayCastStructure>&
v_RayCastStructure, double distance);

    static double scalarWithNormal(Point2D edge, Point2D vector);

    void fire();
    std::pair<std::string, double> cameraRayCheck(RayCastStructure& structure);

    std::map<std::string, Camera&> m_playersOnTheScreen;

    static void drawHealth(sf::RenderWindow& window, int x, int y, int width, int health);
public:
    explicit Camera(World& world, Point2D position, double direction = 0, std::string
texture = SKIN, int health = 100, double fieldOfView = PI/2, double depth = 14, double
walkSpeed = 1.5, double viewSpeed = .005)
    : W_world(world), Circle2D(COLLISION_DISTANCE, position, 0.5, texture, 4),
d_direction(direction), d_fieldOfView(fieldOfView), d_depth(depth), d_walkSpeed(walkSpeed),
d_viewSpeed(viewSpeed), i_health(health) {
        Weapon weapon1(100000);
        weapon1.choiceWeapon("shotgun");
        v_weapons.push_back(weapon1);

        walkSoundBuffer.loadFromFile(WALK_SOUND);
        walkSound.setBuffer(walkSoundBuffer);
        walkSound.setLoop(true);
        walkSound.setVolume(50.f);
    }

    Camera(const Camera& camera) : W_world(camera.W_world) { // copy constructor
        d_height = camera.d_height;
        v_points2D = camera.v_points2D;
        T_texture = camera.T_texture;
        s_texture = camera.s_texture;
        p_position = camera.p_position;
        v_distances = camera.v_distances;
        allCollisions = camera.allCollisions;
        d_direction = camera.d_direction;
        d_depth = camera.d_depth;
        d_fieldOfView = camera.d_fieldOfView;
        d_walkSpeed = camera.d_walkSpeed;
        d_viewSpeed = camera.d_viewSpeed;
        i_health = camera.i_health;
```

```
        b_collision = camera.b_collision;
        b_textures = camera.b_textures;
        b_smooth = camera.b_smooth;
        localMousePosition = camera.localMousePosition;
        v_weapons = camera.v_weapons;
        i_selectedWeapon = camera.i_selectedWeapon;
        walkSoundBuffer = camera.walkSoundBuffer;
        walkSound = camera.walkSound;
        setName(camera.getName());
    }

    void updateDistances(const World& world);
    void drawCameraView(sf::RenderWindow& window);

    void draw(sf::RenderWindow& window) override;

    bool keyboardControl(double elapsedTime, sf::RenderWindow& window);

    void shiftPrecise(Point2D vector);

    bool isSmooth() { return b_smooth; }
    void switchSmooth() { b_smooth = !b_smooth; }
    bool isCollision() { return b_collision; }
    void switchCollision() { b_collision = !b_collision; }
    bool isTextures() { return b_textures; }
    void switchTextures() { b_textures = !b_textures; }

    void previousWeapon();
    void nextWeapon();

    double health() const { return i_health; }
    bool reduceHealth(double damage = 0);
    void fullHealth () { i_health = 100; }
    void setHealth(double h) {i_health = h; }

    int type() override { return 1; }
    std::string lastKill() { return s_lastKill;}
    void cleanLastKill(){s_lastKill = "";}
};
```

Camera.cpp

```
#include <cmath>
#include <windows.h>
#include "Camera.h"



using namespace std;

void Camera::recursiveIncreaseDistance(std::vector<RayCastStructure>& v_RayCastStructure,
double distance) {
    for(auto& rcs : v_RayCastStructure) {
        rcs.distance += distance;
        recursiveIncreaseDistance(rcs.v_mirrorRayCast, distance);
```

```
    }
}

void Camera::objectsRayCrossed(pair<Point2D, Point2D> ray, std::vector<RayCastStructure>
&v_rayCastStruct, const std::string& name, int reflections) {
    std::string obj;
    double len = 0;
    double progress = 0;
    for(auto object : W_world.objects()) {
        Point2D crossPoint = {0, 0};
        obj = "";
        Point2D nearCross = ray.second;
        std::vector<RayCastStructure> v_reflectedRayCastStructure;
        if(object.first == name || object.second.nodes().size() < 2)
            continue;
        progress = 0;
        pair<Point2D, Point2D> segment2 = {object.second.position() +
object.second.nodes().back(), object.second.position() + object.second.nodes().front() };
        for(int k = 0; k < object.second.nodes().size(); k++) {
            if(segments_crossing(ray, segment2, crossPoint)) {
                if ((nearCross - ray.first).abs() > (crossPoint - ray.first).abs()) {
                    nearCross = crossPoint;
                    obj = object.second.getName();
                    len = 0*progress + (segment2.second - nearCross).abs();
                    // For collision detection
                    if(name == getName() && COLLISION_AREA >= (nearCross -
position()).abs()) {
                        CollisionInformation newCollision;
                        newCollision.distance = (nearCross - position()).abs();
                        newCollision.edge = segment2;
                        newCollision.collisionPoint = nearCross;
                        allCollisions.push_back(newCollision);
                    }
                    if(object.second.isMirror() && scalarWithNormal(segment2.second -
segment2.first, ray.second - ray.first) < 0) {
                        Point2D edgeVector = segment2.second - segment2.first;
                        Point2D rayVector = ray.second - ray.first;
                        double twistAngle = 2*acos((edgeVector.x*rayVector.x +
edgeVector.y*rayVector.y)/(edgeVector.abs()*rayVector.abs()));
                        Point2D twistedRayVector = {rayVector.x*cos(twistAngle) +
rayVector.y*sin(twistAngle), -rayVector.x*sin(twistAngle) + rayVector.y*cos(twistAngle)};
                        pair<Point2D, Point2D> newRay = {crossPoint, crossPoint +
twistedRayVector};

                        if(reflections < 40) {
                            objectsRayCrossed(newRay, v_reflectedRayCastStructure,
object.first,reflections + 1);
                            recursiveIncreaseDistance(v_reflectedRayCastStructure,
(ray.first - nearCross).abs());
                        }
                    }
                }
            } else {
                progress += (segment2.second - segment2.first).abs();
            }
            if(k+1 < object.second.nodes().size())
```

```
                segment2 = { object.second.position() + object.second.nodes()[k],
object.second.position() + object.second.nodes()[k+1] };
        }
        if(!obj.empty())
            v_rayCastStruct.push_back({(ray.first-nearCross).abs(), len, obj,
object.second.height(), v_reflectedRayCastStructure});
    }
    sort(v_rayCastStruct.begin(), v_rayCastStruct.end(), [](const RayCastStructure& lh,
const RayCastStructure& rh) { return lh.distance > rh.distance; });
}

void Camera::updateDistances(const World& world) {
    v_distances.clear();
    allCollisions.clear();

    for(int i = 0; i < 2*PI/d_fieldOfView*DISTANCES_SEGMENTS; i++) {
        double left = d_direction - d_fieldOfView/2;
        double right = d_direction + d_fieldOfView/2;
        // find distance to near object in that direction
        double direction = d_direction + ((double)i/DISTANCES_SEGMENTS -
0.5)*d_fieldOfView;

        pair<Point2D, Point2D> segment1 = {{x(), y()}, {x() + d_depth*cos(direction), y() +
d_depth*sin(direction)}};

        std::vector<RayCastStructure> v_rayCastStructure;

        objectsRayCrossed(segment1, v_rayCastStructure, getName());

        for(const auto& rcs : v_rayCastStructure)
            if((m_playersOnTheScreen.count(rcs.object) == 0) && (W_world[rcs.object].type()
== 1))
                m_playersOnTheScreen.insert({rcs.object,
dynamic_cast<Camera&>(W_world[rcs.object])});

        if(!v_rayCastStructure.empty())
            v_distances.push_back(v_rayCastStructure);
        else
            v_distances.push_back({{d_depth, 0, "", 1}});
    }
}

void Camera::draw(sf::RenderWindow& window) {

    if(v_distances.empty() || i_health <= 0)
        return;

    sf::CircleShape circle;
    circle.setRadius(COLLISION_DISTANCE*SCALE);
    circle.setOutlineColor(OUTLINE_CAMERA_COLOR);
    circle.setFillColor(FILL_CAMERA_COLOR);
    circle.setOutlineThickness(OUTLINE_CAMERA_THICKNESS);
    circle.setPosition((float)x()*SCALE - COLLISION_DISTANCE*SCALE, (float)y()*SCALE -
COLLISION_DISTANCE*SCALE);

    double left = d_direction - d_fieldOfView/2;
    double right = d_direction + d_fieldOfView/2;
```

```
    sf::ConvexShape triangle;
    triangle.setPointCount(CONVEX_NUMBER + 2);
    triangle.setPoint(0, sf::Vector2f(0, 0));
    for(int i = 0; i <= CONVEX_NUMBER; i++) {
        int index = v_distances[(int)i*DISTANCES_SEGMENTS/CONVEX_NUMBER].size() - 1;
        triangle.setPoint(i+1,
sf::Vector2f(v_distances[(int)i*DISTANCES_SEGMENTS/CONVEX_NUMBER][index].distance*cos(left
+ (right - left)*i/CONVEX_NUMBER)*SCALE,
v_distances[(int)i*DISTANCES_SEGMENTS/CONVEX_NUMBER][index].distance*sin(left + (right -
left)*i/CONVEX_NUMBER)*SCALE));
    }
    triangle.setOutlineColor(OUTLINE_CAMERA_COLOR);
    triangle.setFillColor(FILED_OF_VEW_COLOR);
    triangle.setOutlineThickness(OUTLINE_CAMERA_THICKNESS);
    triangle.setPosition((float)x()*SCALE, (float)y()*SCALE);

    window.draw(triangle);
    window.draw(circle);
}

std::pair<std::string, double> Camera::cameraRayCheck(RayCastStructure &structure) {
    std::pair<std::string, double> result = {"", 1};
    if(W_world[structure.object].type() == 1) {
        result.first = structure.object;
        result.second = structure.distance;
    }
    else if(!structure.v_mirrorRayCast.empty()) {
        return
cameraRayCheck(structure.v_mirrorRayCast[structure.v_mirrorRayCast.size()-1]);
    }
    return result;
}

void Camera::fire() {
    pair<Point2D, Point2D> segment1 = {{x(), y()}, {x() + d_depth*cos(d_direction), y() +
d_depth*sin(d_direction)}};
    std::vector<RayCastStructure> v_rayCastStructure;
    objectsRayCrossed(segment1, v_rayCastStructure, getName());

    if(!v_rayCastStructure.empty()) {
        std::pair<std::string, double> hitted =
cameraRayCheck(v_rayCastStructure[v_rayCastStructure.size()-1]);
        //W_world.removeObject2D(hitted.first);
        if(hitted.first == "") return;

        dynamic_cast<Camera&>(W_world[hitted.first]).updateDistances(W_world);
        double dir = 2*PI * rand() / RAND_MAX;
        dynamic_cast<Camera&>(W_world[hitted.first]).shiftPrecise({0.05*cos(dir),
0.05*sin(dir)});


if(dynamic_cast<Camera&>(W_world[hitted.first]).reduceHealth(static_cast<int>(v_weapons[i_s
electedWeapon].damage() / hitted.second))) {
            //W_world.removeObject2D(hitted.first);
            W_world[hitted.first].setPosition({2.5, 0});
            s_lastKill = hitted.first;
```

```
            dynamic_cast<Camera&>(W_world[hitted.first]).setHealth(100);
        }
    }
}

bool Camera::keyboardControl(double elapsedTime, sf::RenderWindow& window) {
    double dx = 0;
    double dy = 0;

    // left and right
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::A))
    {
        dx += sin(d_direction);
        dy += -cos(d_direction);
    }
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::D))
    {
        dx += -sin(d_direction);
        dy += cos(d_direction);
    }
    // forward and backward
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::W))
    {
        dx += cos(d_direction);
        dy += sin(d_direction);
    }
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::S))
    {
        dx += -cos(d_direction);
        dy += -sin(d_direction);
    }

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Escape))
    {
        return false;
    }
    // Mouse
    if (sf::Mouse::isButtonPressed(sf::Mouse::Left))
    {
        // левая кнопка мыши нажата, стреляем!
        if(v_weapons[i_selectedWeapon].fire())
            fire();
    }
    if(sf::Mouse::getPosition(window).x != localMousePosition.x) {
        double difference = sf::Mouse::getPosition(window).x - localMousePosition.x;
        sf::Mouse::setPosition({SCREEN_WIDTH/2, SCREEN_HEIGHT/2});
        localMousePosition = sf::Mouse::getPosition(window);
        d_direction += d_viewSpeed * difference;
    }

    if((dx*dx + dy*dy) > d_walkSpeed * elapsedTime * d_walkSpeed * elapsedTime / 10) {
       if(walkSound.getStatus() != sf::Sound::Status::Playing)
           walkSound.play();
    } else {
        walkSound.pause();
    }
```

```
    shiftPrecise({dx * d_walkSpeed * elapsedTime * ((double)i_health/100), dy * d_walkSpeed
* elapsedTime * ((double)i_health/100)});
    return true;
}

pair<double, double> heightInPixels(double distance, double height) {
    pair<double, double> h = {0, 0};
    h.first = (1 - height/distance)*SCREEN_HEIGHT/2;
    h.second = (1 + 1/distance)*SCREEN_HEIGHT/2;
    return h;
}


void Camera::drawVerticalStrip(sf::RenderWindow &window, const RayCastStructure& obj, int
shift, double s) {
    sf::ConvexShape polygon;
    polygon.setPointCount(4);

    pair<double, double> height_now = heightInPixels(obj.distance, obj.height);

    int h1 = height_now.first + s;
    int h2 = height_now.second - s;

    polygon.setPoint(0, sf::Vector2f(0, h1));
    polygon.setPoint(1, sf::Vector2f(0, h2));
    polygon.setPoint(2, sf::Vector2f(SCREEN_WIDTH / DISTANCES_SEGMENTS, h2));
    polygon.setPoint(3, sf::Vector2f(SCREEN_WIDTH / DISTANCES_SEGMENTS, h1));

    int alpha = 255 * (1 - obj.distance / d_depth);
    if (alpha > 255)
        alpha = 255;
    if (alpha < 0)
        alpha = 0;

    if (!b_textures)
        polygon.setFillColor({255, 174, 174, static_cast<sf::Uint8>(alpha)});
    else
        polygon.setFillColor({255, 174, 174, 255});

    polygon.setOutlineThickness(0); // we can make non zero thickness for debug
    polygon.setPosition(shift * SCREEN_WIDTH / DISTANCES_SEGMENTS, 0);
    if (abs(obj.distance - d_depth) > 0.001)
        window.draw(polygon);

    double scaleFactor = (double) (h2 - h1) / SCREEN_HEIGHT;
    sf::Sprite sprite;
    if (obj.object != "" && b_textures) {
        int left = obj.progress * SCREEN_WIDTH;
        int top = 0;
        if(W_world[obj.object].isMirror()) { // In case of mirror
            sprite.setTexture(W_world.skyTexture());
            left = (d_direction/10) * SCREEN_WIDTH;
            top = sprite.getTextureRect().height/2-SCREEN_HEIGHT/2;
        } else {
            sprite.setTexture(W_world[obj.object].loadTexture());
        }
```

```
        sprite.setTextureRect(sf::IntRect(left, top, SCREEN_WIDTH / DISTANCES_SEGMENTS,
SCREEN_HEIGHT));
        sprite.setPosition(sf::Vector2f(shift * SCREEN_WIDTH / DISTANCES_SEGMENTS, h1)); //
абсолютная позиция
        sprite.scale(1, scaleFactor);
        sprite.setColor({255, 255, 255, static_cast<sf::Uint8>(alpha)});
        window.draw(sprite);
    }
}

void Camera::recursiveDrawing(sf::RenderWindow& window, const
std::vector<RayCastStructure>& v_RayCastStructure, int shift) {
    for(const auto & k : v_RayCastStructure) {
        drawVerticalStrip(window, k, shift, 0);
        if(!k.v_mirrorRayCast.empty())
            recursiveDrawing(window, k.v_mirrorRayCast, shift);
    }
}

void Camera::drawHealth(sf::RenderWindow& window, int xPos, int yPos, int width, int
healthProgress) {
    sf::ConvexShape polygon1;
    polygon1.setPointCount(4);
    sf::ConvexShape polygon2;
    polygon2.setPointCount(4);

    polygon1.setPoint(0, sf::Vector2f(xPos, yPos));
    polygon1.setPoint(1, sf::Vector2f(xPos + width, yPos));
    polygon1.setPoint(2, sf::Vector2f(xPos + width, yPos + 20));
    polygon1.setPoint(3, sf::Vector2f(xPos, yPos + 20));

    polygon2.setPoint(0, sf::Vector2f(xPos, yPos));
    polygon2.setPoint(1, sf::Vector2f(xPos + width*healthProgress/100, yPos));
    polygon2.setPoint(2, sf::Vector2f(xPos + width*healthProgress/100, yPos + 20));
    polygon2.setPoint(3, sf::Vector2f(xPos, yPos + 20));

    polygon1.setFillColor({255, 174, 174, 100});
    polygon2.setFillColor({static_cast<sf::Uint8>((100 - healthProgress)*255),
static_cast<sf::Uint8>(healthProgress*255/100), 0, 255});

    polygon1.setOutlineThickness(3);
    window.draw(polygon1);
    window.draw(polygon2);
}

void Camera::drawCameraView(sf::RenderWindow& window) {

    if(v_distances.empty())
        return;
    //SKY AND FLOOR
    if(b_textures) {
        sf::Sprite sprite_sky;
        sprite_sky.setTexture(W_world.skyTexture());
        sprite_sky.setTextureRect(sf::IntRect(d_direction * SCREEN_WIDTH/2,
sprite_sky.getTextureRect().height/2-SCREEN_HEIGHT/2, SCREEN_WIDTH, 1080));
        sprite_sky.setPosition(sf::Vector2f(0,0));
        window.draw(sprite_sky);
```

```
    }

    for(int i = 0; i < DISTANCES_SEGMENTS-1; i++)
        recursiveDrawing(window, v_distances[i], i);

    //m_playersOnTheScreen
    for(const auto& player : m_playersOnTheScreen) {
        Point2D enemyDirection = (player.second.position() - position()).normalize();
        Point2D cameraLeftDirection = {cos(d_direction - d_fieldOfView/2), sin(d_direction
- d_fieldOfView/2)};
        double angle = acos(enemyDirection*cameraLeftDirection);
        if((cameraLeftDirection.x*enemyDirection.y -
enemyDirection.x*cameraLeftDirection.y) > 0) {
            int xPos = SCREEN_WIDTH * angle/d_fieldOfView;
            int yPos = heightInPixels((player.second.position() - position()).abs(),
1).first;
            auto healthProgress = (double)player.second.health();
            drawHealth(window, xPos-50, yPos, 100, healthProgress);
        }
    }
    drawHealth(window, 50, SCREEN_HEIGHT - 50, 400, health());
    v_weapons[i_selectedWeapon].draw(window);
}

double Camera::scalarWithNormal(Point2D edge, Point2D vector) {
    Point2D normal = {edge.y, -edge.x};
    normal = normal.normalize();
    double scalar = vector.x*normal.x + vector.y*normal.y;
    return scalar;
}

void Camera::shiftPrecise(Point2D vector) {

    if(!b_collision) {
        shift(vector);
        return;
    }

    for(auto c : allCollisions) {
        Point2D edgeVector = c.edge.second - c.edge.first;
        Point2D normal = {edgeVector.y, -edgeVector.x};
        normal = normal.normalize();
        double scalar = vector.x*normal.x + vector.y*normal.y;
        if(scalarWithNormal(edgeVector, vector) < 0) {
            if(c.distance - abs(scalar) < COLLISION_DISTANCE) {
                vector.x -= normal.x*scalar;
                vector.y -= normal.y*scalar;
            }
        }
    }

    shift(vector);
}

void Camera::previousWeapon() {
    if(i_selectedWeapon > 0)
        i_selectedWeapon--;
```

```
    else
        i_selectedWeapon = v_weapons.size() - 1;
}

void Camera::nextWeapon() {
    if(i_selectedWeapon < v_weapons.size() - 1)
        i_selectedWeapon++;
    else
        i_selectedWeapon = 0;
}

bool Camera::reduceHealth(double damage) {
    if(i_health > 100)
        i_health = 100;
    i_health -= damage;
    return i_health < 0;
}
```

UDPSocketConnection.h

```
#include <iostream>
#include <SFML/Network.hpp>
#include <utility>
#include "World.h"
#include "Camera.h"

class UDPSocketConnection {
private:
    World& W_world;
    Camera& C_camera;
    sf::UdpSocket socket;

    std::map<short unsigned, Camera> m_cameras;
    std::string s_ipAdress;
    short unsigned i_myPort;

    std::vector<Point2D> v_spawns = {{1.5, 1.5}, {1.5, 9}};
public:
    UDPSocketConnection(World& world, Camera& camera) : W_world(world), C_camera(camera) {}

    void bind(std::string ip, short unsigned port) {
        s_ipAdress = std::move(ip);
        socket.bind(port);
        socket.setBlocking(false);
        i_myPort = port;
        C_camera.setName(std::to_string(port));
    }

    void update() {
        double x = 0;
        double y = 0;
        int health = 100;
        std::string killedName;
```

```
        sf::Packet packet;
        sf::IpAddress sender;
        short unsigned int senderPort = 0;
        short unsigned int port = 0;

        bool ack1 = false;
        bool ack2 = false;


        while((socket.receive(packet, sender, port) == sf::Socket::Status::Done) && (port
>= 54000) && (port <= 54010)) {
            packet >> x >> y >> killedName >> ack2 >> health;
            senderPort = port;
        }
        while((socket.receive(packet, sender, port) == sf::Socket::Status::Done)) {};

        if(ack2)
            C_camera.cleanLastKill();

        if(killedName == C_camera.getName()) {
            C_camera.setPosition(v_spawns[i_myPort % v_spawns.size()]);
            C_camera.fullHealth();
            ack1 = true;
        }

        if(W_world.isExist(std::to_string(senderPort))) {
            m_cameras.at(senderPort).setPosition({x, y});
            C_camera.setHealth(health);
            m_cameras.at(senderPort).reduceHealth(-0.1);
        } else if ((senderPort >= 54000) && (senderPort <= 54010)){
            Camera camera(W_world, {2.5, 0});
            camera.setName(std::to_string(senderPort));
            m_cameras.insert({senderPort, camera});
            W_world.addObject2D(m_cameras.at(senderPort), std::to_string(senderPort));
        }


        for(int i = 54000; i < 54010; i++) {
            sf::Packet packetSend;
            int h = m_cameras.count(i) != 0 ? m_cameras.at(i).health() : 100;
            packetSend << C_camera.x() << C_camera.y() << C_camera.lastKill() << ack1 << h;
            if (i != i_myPort)
                socket.send(packetSend, s_ipAdress, i);
        }
    }
};
```

Main.cpp

```
#include <SFML/Audio.hpp>
#include <SFML/Graphics.hpp>

#include <map>
#include <iostream>
#include <chrono>
#include "settings.h"
#include "Poligon2D.h"
#include "Circle2D.h"
#include "Object2D.h"
#include "World.h"
#include "Camera.h"
#include "Menu.h"
#include "UDPSocketConnection.h"

using namespace std;

int main()
{
    sf::Music music;
    music.openFromFile("sounds/unrealSuperHero3.ogg");
    music.setVolume(10);
    music.setLoop(true);
    music.play();

    sf::Music backSounds;
    backSounds.openFromFile("sounds/backSounds.ogg");
    backSounds.setVolume(20);
    backSounds.pause();

    vector<Point2D> columnPositions = {{0,0}, {3,0}, {3,3}, {6,3}, {6,8}, {3,8}, {3,11},
{0,11}, {0,8}, {-3,8}, {-3,3}, {0,3}};
    vector<Circle2D> columns(12);
    vector<Poligon2D> walls (12);

    sf::RenderWindow window(sf::VideoMode(SCREEN_WIDTH, SCREEN_HEIGHT), "Pseudo3DEngine",
sf::Style::Fullscreen);

    World world(100, 100);
    Camera camera(world, {1.5, 1.5});

    for(int i = 0; i < columns.size(); i++) {
        columns[i].setPosition(columnPositions[i]);
        if(i != columns.size()-1)
            walls[i].setPoints2D({columnPositions[i], columnPositions[i+1]});
        else
            walls[i].setPoints2D({columnPositions[i], columnPositions[0]});

        world.addObject2D(walls[i], "wall" + std::to_string(i+1));
        world.addObject2D(columns[i], "col" + std::to_string(i+1));
    }
```

```
    UDPSocketConnection udpSocketConnection(world, camera);
    udpSocketConnection.bind("192.168.137.255", 54001); // 192.168.137.255

    //wall4.makeMirror();

    Poligon2D object1({{1, 1}, {2.05, 1}, {2, 2}}, {2.7, 5}, 1, INSTRUCTION_TEXTURE);
    Poligon2D object2({{0, 0}, {1, 0}, {1, 1}, {0, 1}}, {-1.9, 6.7}, 1, FROZEN_TEXTURE);
    Poligon2D object3({{1, 1}, {2, 2}, {1, 2}, {1, 2}}, {-0.79, 3.95});
    Poligon2D object4({{0, 0}, {.1, 0}, {.1, .1}, {0, .1}}, {1.92, 6.18});
    Poligon2D object5({{0, 0}, {.3, 0}, {.3, .3}, {0, .3}}, {1.84, 9});
    Poligon2D object6({{0, 0}, {.3, 0}, {.3, .3}, {0, .3}}, {1.11, 7}, 0);
    Poligon2D object7({{0, 0}, {.1, 0}, {.1, .1}, {0, .1}}, {1, 2}, 0);

    Poligon2D object8({{0, 0}, {.3, 0}, {.3, .3}, {0, .3}}, {3.6, 4.6}, 0.9);
    Poligon2D object9({{0, 0}, {.3, 0}, {.3, .3}, {0, .3}}, {1.5, 4.2}, 0.9);
    Poligon2D object10({{0, 0}, {.1, 0}, {.1, .1}, {0, .1}}, {2.5, 5.2}, 0.9);

    //object2.makeMirror();
    object3.makeMirror();

    //cameraw
    world.addObject2D(camera, camera.getName());

    world.addObject2D(object1, "object1");
    world.addObject2D(object2, "object2");
    world.addObject2D(object3, "object3");
    world.addObject2D(object4, "object4");
    world.addObject2D(object5, "object5");
    world.addObject2D(object6, "object6");
    world.addObject2D(object7, "object7");

    world.addObject2D(object8, "object8");
    world.addObject2D(object9, "object9");
    world.addObject2D(object10, "object10");

    Menu menu;


    auto tp1 = chrono::system_clock::now();
    auto tp2 = chrono::system_clock::now();

    while (window.isOpen())
    {
        tp2 = chrono::system_clock::now();
        chrono::duration <double> elapsedTime = tp2 - tp1;
        tp1 = tp2;
        double d_elapsedTime = elapsedTime.count();
        std::string title = "Pseudo3DEngine " +
std::to_string((double)1/elapsedTime.count()) + "fps. x:" + std::to_string(camera.x()) + ",
y:" + std::to_string(camera.y());
        window.setTitle(title);
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }
```

```
        window.clear();
        if(!menu.isPaused()) {
            udpSocketConnection.update();
            window.setMouseCursorVisible(false);
            camera.updateDistances(world);
            camera.drawCameraView(window);
            //world.draw(window);
            if(!camera.keyboardControl(d_elapsedTime, window)) {
                menu.setPause();
                music.play();
                backSounds.pause();
            };
        } else {
            window.setMouseCursorVisible(true);
            if(menu.isTextures() != camera.isTextures())
                camera.switchTextures();
            if(menu.isCollision() != camera.isCollision())
                camera.switchCollision();

            menu.drawMenu(window, d_elapsedTime);
            //FOR MUSICS
            if(!menu.isPaused()) {
                backSounds.play();
                music.pause();
            }
        }
        window.display();
    }

    return 0;
}
```