Міністерство освіти і науки України

Ніжинський державний університет імені Миколи Гоголя

ННІ природничо-математичних, медико-біологічних наук та інформаційних технологій

Кафедра інформаційних технологій, фізико-математичних та економічних наук

Освітня програма: Комп’ютерні науки

Спеціальність:122 Комп’ютерні науки

**КВАЛІФІКАЦІЙНА РОБОТА**

на здобуття освітнього ступеня *магістр*

**Створення скриптової мови програмування для ігрового рушія**

студента **Циганкова Андрія Сергійовича**

**Науковий керівник:**

Фетісов Валерій Сергійович,

кандидат економічних наук, доцент

**Рецензент:**

Казачков Іван Васильович,

доктор технічних наук, професор

Кресан Тетяна Анатоліївна,

кандидат технічних наук, доцент

**Допущено до захисту:** \_\_\_\_ \_\_\_\_\_\_ 2024 р.

Завідувач кафедри

проф. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Казачков І.В.

Ніжин − 2024

Вступ 03

РОЗДІЛ 1. СКРИПТОВА МОВА ПРОГРАМУВАННЯ

1.1 Необхідність скриптових мов програмування в ігровому рушії 06

1.2 Поєднання скриптової мови програмування з С++ кодом 09

1.2.1 Реалізація в Unreal Engine 5 10

1.2.2 Реалізація в Lua 11

РОЗДІЛ 2. ОСНОВИ СКРИПТОВОЇ МОВИ ПРОГРАМУВАННЯ

2.1 Основні вимоги до скриптової мови програмування 14

2.2 Розробка лексера 15

2.3 Правила мови програмування 19

2.4 Контейнери 24

2.5 Розробка компілятора 32

2.5.1 Замикання 33

2.5.2 Обробка помилок 34

2.5.3 Підтримка ООП 36

РОЗДІЛ 3. ВІРТУАЛЬНА МАШИНА

3.1 Особливості віртуальної машини 40

3.2 Виконання машинного коду 42

3.3 Реалізація прибиральника сміття 45

РОЗДІЛ 4. МЕХАНІЗМИ ДЛЯ СТВОРЕННЯ ІГРОВОЇ ЛОГІКИ

4.1 Реалізація оновлення стану гру в рушії 50

4.2 Огляд механізмів для створення ігрової логіки 54

4.3 Реалізація механізмів у скриптовій мові 55

РОЗДІЛ 5. ІНТЕГРАЦІЯ

5.1 Інтеграція мови програмування з ігровим рушієм Unreal Engine 5 61

5.2 Рекомендації щодо покращення скриптової мови програмування 66

ВИСНОВКИ 69

СПИСОК ВИКОРИСТАНИХ ІНФОРМАЦІЙНИХ ДЖЕРЕЛ 70

**Вступ**

**Актуальність теми.** З кожним роком все більше і більше програм стають комплексними. Вони пропонують різноманітний функціонал, який постійно розширюється. Це вимагає створення більш гнучкої системи для розробки певного продукту.

Під час проєктування систем, архітектори намагаються виокремити системи, які вимагають високої кваліфікації розробників, та більш прості системи, які можуть розробляти менш досвідчені програмісти. В результаті цього поділу деякі проєкти додають скриптові мови програмування, які легко інтегруються в основну кодову базу. Прикладом таких мов може слугувати Python, JavaScript, TypeScript, Lua. Кожна з цих мов програмування виконується у віртуальній машині та має можливість виконання С++ функцій поза віртуальною машиною.

Подібна стратегія проєктування використовується також в ігрових рушіях. Зазвичай, ігрові рушії використовують С++ як основну мову програмування та одну чи дві скриптових мов програмування для написання гнучкої логіки гри. Ці мови програмування можуть бути або відкритими, або закритими. Прикладом цього може слугувати мови програмування Lua, яка є Open Source, та WitcherScript, яка була створена спеціально для рушія студії CDProjectRed і є закритою [15]. Існує велика кількість переваг скриптових мов над С/С++, але основними є:

1. Менший поріг входу для розробника.
2. Простота та швидкість написання коду, наявність автоматичного контролю пам’яті.
3. Швидке відображення змін коду в грі.

Наприклад, рушій Unreal Engine використовував Slate UI Framework для розробки користувацького інтерфейсу. Ця система була складною і вимагала написання коду на мові програмування С++, що ускладнювало пошук програмістів, які були знайомі з системою, та можливості відлагоджування програми, бо для кожної зміни треба було збирати рушій. Для вирішення цієї проблеми, програмісти створили обгортку над Slate -UMG (Unreal Motion Graphics). Після цього стало простіше розробляти UI, бо більшість функціоналу перенесли в скриптову мову рушія та в графічний інтерфейс, який дозволяє бачити зміни на льоту [23].

Тому, враховуючи вище викладеному, нами поставлена наступна **мета** магістерської роботи:

проаналізувати особливості розробки скриптових мов в ігрових рушіях та механізми створення ігрової логіки, дослідити застосування математичних правил у граматиці мови програмування, написати необхідний код для реалізації скриптової мови програмування, який можна інтегрувати до будь-якого ігрового рушія.

Для вирішення даної мети нами сформульовані наступні **завдання:**

1. Опрацювати інформаційні джерела з теми магістерської роботи.
2. Визначити специфіку скриптових мов програмування для застосування в ігрових рушіях.
3. Опрацювати ряд вимог для скриптових мов програмування.
4. Спроектувати та реалізувати весь необхідний набір класів для реалізації сканування, компіляції та виконання коду скриптової мови програмування.
5. Реалізувати необхідні механізми для створення ігрового коду.
6. Описати кроки інтеграції скриптової мови програмування у рушій Unreal Engine.
7. Окреслити можливі способи покращення коду скриптової мови програмування.

**Об'єктом нашого дослідження** є скриптові мови програмування.

**Предметом** – реалізація необхідного коду скриптової мови програмування, яке може бути інтегроване до ігрового рушія, та додаткового функціоналу, який є необхідним для написання ігрової логіки програмістами.

**Наукова новизна одержаних результатів**. Результати, які були одержані в магістерській роботі, дають основу розуміння того, як реалізовані різноманітні скриптові мови програмування, що застосовуються в ігрових рушіях.

**Практичне значення одержаних результатів.** Практичне значення одержаних результатів полягає в написанні сканера, компілятора, інтерпретатора, який забезпечує базову реалізацію скриптової мови програмування, та реалізації необхідних інструментів для написання ігрового коду. В результаті, код проєкту можна перенести до будь якого продукту у вигляді набору файлів або динамічної бібліотеки.

**Особистий внесок дослідника**. Магістерська робота розглядає особливості ігрових рушіїв, способи взаємодії між віртуальною машиною скриптової мови програмування та мовою програмування рушія (С/С++). У практичній частині магістерської роботи був наведений приклад розробки скриптової мови програмування.

**Структура роботи**. Магістерська робота складається зі вступу, теоретичного розділу, та експериментальної частини розділів, висновків і списку використаних джерел, до складу якого входить 27 найменувань. Кваліфікаційна робота налічує 72 сторінок друкованого тексту, проілюстрована 35 рисунками.

**СКРИПТОВА МОВА ПРОГРАМУВАННЯ**

**1.1 Необхідність скриптових мов програмування в ігровому рушії**

В даний момент все більше і більше ігор розробляються для гравців на різноманітні платформи: ПК, мобільні телефони, консолі, портативні консолі, тощо. Комп’ютерна гра є цілою системою пов’язаних між компонентів що дають можливість гравцю отримати нові емоції від дослідження ігрового світу та його правил. Прикладами таких компонентів є: фізичний рушій, система скелетної анімації, система завантаження рівнів під час роботи гри (Level Streaming), система рендерингу ігрового світу на екран монітору, система штучного інтелекту в іграх, тощо. Завдяки цим системам програмісти створюють правила цього світу.

Більшість ігрових рушіїв були розроблені в різні роки з відповідними технологіями, які були доступні на той момент, та сфокусовані на конкретних типах гри (наприклад стратегії, онлайн шутери, тощо), що впливає на його підходи до реалізації. Також треба враховувати, що кожна студія має свій бюджет для розробки рушія, тому розробник буде шукати компроміс між реалізацією свого рішення або плати за ліцензію певного готового рішення та подальшого інтегрування його в рушій.

Прикладами такого рішення може слугувати фізичний рушій Havok або Nvidia Physics, які використовуються у великій кількості ігор. Проте деякі розробники вирішують розробляти свої системи. Наприклад, рушій Unreal Engine 5 перейшов з Nvidia Physics на свій фізичний рушій Chaos.

Фізичні рушії можуть використовувати свої системи такі як рефлексія, проте вони не підтримують скриптові мови програмування, та повністю реалізовані мовою програмування С++. Проте є певні рішення, які обов’язково принесуть за собою залежності у вигляді додаткових мов програмування. Наприклад бібліотека Coherent UI використовує такі мови як JavaScript, HTML, CSS.

Більша частина гри розробляється за допомогою мови програмування С++. Ця мова програмування є найбільш універсальною та максимально швидкою, так як вихідний код компілятора С++ (надалі в цій роботі буде використаний компілятор Microsoft Visual C++ Compiler) є машинним кодом, який виконується процесором напряму.

Мова С++ має свої переваги та недоліки, проте при правильному проєктуванні системи та мінімізації небезпечних практик С++ має значну перевагу над мовою С. Перша перевага в тому, що компілятор С++ за рахунок механізмів стандарту може генерувати більш оптимізований код, на відміну від компілятора С. Друга перевага в тому, що С++ має більше можливостей ніж С такі як ООП, шаблони, перевантаження функцій [4].

Проте швидкість виконання завжди вимагає складної системи компіляції та оптимізації коду. Для генерації машинного коду компілятору необхідно виконати такі кроки [7]:

1. Препроцесінг. Програма читає вихідний код і виконує певні операції над ним. Ці операції включають включення файлів, розкриття макросів і заміну директив препроцесора на їх відповідні значення. На цьому кроці генерується вихідний файл для наступного етапу.
2. Компіляція. Компілятор бере проміжний файл, створений препроцесором та компілює файл. Під час компіляції цей код проходить ряд як загальних так і платформо-залежних оптимізацій та перетворюється в IR (Intermediate Representation).
3. Асемблювання. Асемблер використовує IR, створений компілятором, і генерує машинний код в форматі, який вже може бути безпосередньо виконаний процесором.
4. Лінкування. Програма взаємодіє з асембльованими файлів та об'єднує їх у виконувану програму. Компонувальник також розпізнає зовнішні посилання між об'єктними файлами та різними динамічними бібліотеками. Результатом роботи лінкера є виконувана програма.

Таким чином, будь яка зміна в коді вимагає перекомпіляцію проєкту, що може привести як до часткової компіляції змінених файлів та інкрементального лінкування, або повної компіляції проєкту у випадку якщо змінений файл-заголовок який додається до великої кількості файлів. Це збільшує час тестування навіть при незначних змінах коду.

Процес компіляції унеможливлює відображення зміни в реальному часі. Для цього необхідно перезапускати клієнт гри використовуючи зібраний новий виконуваний файл. Частковим винятком цього правила є архітектура рушія Unreal Engine, яка підтримує модульну систему виконуваного коду, що дозволяє виносити окремі компоненти в динамічні бібліотеки, які можна вивантажувати та завантажувати в пам’ять під час роботи редактора рушія. Проте такий спосіб має певні проблеми з рефлексією рушія. Зміни типів даних та самих даних, відкритих для рефлексії рушія, можуть призвести до аварійного завершення роботи всього редактора. Також треба враховувати, що не кожна архітектура рушія є достатньо модульною для реалізації модульного підходу.

Мова програмування С++ є досить складною для вивчення та пошуку помилок в коді, що, відповідно, вимагає високий рівень кваліфікації програміста для внесення будь яких змін в код гри.

На противагу цим особливостям мови програмування С++, ігрові рушії додають підтримку скриптових мов програмування, які вирішують вище наведені проблеми рушія та дозволяють створити більш простий код ніж аналогічна реалізація мовою С++:

1. Скриптова мова програмування не вимагає високої кваліфікації для програміста. Деяку ігрову логіку можуть писати дизайнери систем, UX/UI дизайнери тощо.
2. Скриптова мова програмування є дуже гнучкою до швидкого відображення змін під час виконання програм і не вимагає перезапуску гри.
3. На відміну від С++, скриптова мова програмування мінімізує ризик створення критичних помилок, які можуть призводити до аварійного закриття гри.
4. Правильно спроєктована скриптова мова програмування має досить високу швидкість виконання ігрового коду.
5. Скриптові мови програмування дозволяють створювати логіку, яка буде виконуватися в середині віртуальної машини і не впливати на будь які інші системи чи ресурси ігрового рушія.

Залежно від рушія та його потреб при реалізації певної гри скриптові мови програмування можуть виконувати конкретні завдання, або розширювати можливості геймплею:

1. Реалізовувати логіку керування гравцями в грі. Наприклад створення дерев поведінок, або реєстрація певних обробників подій, які потім будуть викликані з нативного коду.
2. Розширювати наявні механіки, створювати певні скриптові події в грі або визначати певні параметри такі як швидкість руху персонажа, кількість балів здоров’я, тощо.
3. Налаштування правил гри у визначеному світі або ігрових режимах (якщо гра має кілька ігрових режимів такі як Multiplayer, Singleplayer, Co-op тощо).

**1.2 Поєднання скриптової мови програмування з С++ кодом**

На даний момент є кілька стратегій реалізації методів поєднання скриптових мов програмування з кодом рушія написаного на С++:

1. Рушій не містить віртуальної машини, яка працює окремо. Певний С++ код викликає допоміжні функції, які зчитують байт-код та виконують його. За необхідністю байт-код може викликати певні нативні функції.
2. Окрема віртуальна машина, яка виконує скриптовий код у захищеному середовищі незалежно від нативного коду, проте надає можливості для обміну даних між віртуальною машиною та кодом рушія.

**1.2.1 Реалізація в Unreal Engine 5**

Перший спосіб характерний саме для рушія Unreal Engine. Програміст пише код на мові Blueprints, який рушій компілює в байт-код та зберігає його в бінарному файлі. Для програміста доступний набір функцій що реалізовані у вигляді байт-коду, та тих, що реалізовані в нативному коді.

Для створення функції, яку можна викликати з байт-коду, необхідно створити відповідну плюсову функцію з параметром BlueprintCallable. Програма Unreal Header Tool сканує код перед компіляцією та генерує необхідний додатковий код, в якому визначається ця функція та її параметри і реєструється в окремому масиві функцій.

Для створення функції, яка може бути викликана у нативному коді, але реалізована у вигляді байт-коду, необхідно створити оголошення функції без її реалізації (Unreal Header Tool автоматично згенерує відповідну реалізацію цієї функції) з параметром BlueprintNativeEvent. В результаті рушій додає цю функцію в систему рефлексії, яку використовує редактор коду скриптової мови програмування рушія [12]. Рушій має велику кількість вже вбудованих функцій. Наприклад клас AActor має такі функції:

1. AActor::ReceiveBeginPlay.
2. AActor::ReceiveTick.
3. AActor::ReceiveEndPlay.

Кожна з цих функцій викликається в момент, коли рушій викликає батьківські функції (AActor::BeginPlay, AActor::Tick, AActor::EndPlay відповідно).

Такий підхід створює систему, де простіше знаходити баги та контролювати середовище виконання, проте вимагає додаткових зусиль для розробки такої системи. Додатковою особливістю Blueprints є те, що ця мова програмування є строго типізованою. Тобто є чітка вимога до всіх типів, які передаються в функцію, що помітно на етапі компіляції коду Blueprints [23].

**1.2.2 Реалізація в Lua**

Другий спосіб використовується в більшості рушіїв, які не мають можливості підтримувати рефлексію схожу на Unreal Engine, реалізовують просту спосіб додавання скриптової мови. Віртуальна машина скриптової мови програмування та С++ код працюють окремо але мають певний спосіб для комунікації між ними. Такий спосіб комунікації називають стеком. Розробник може покласти дані на стек віртуальної машини, які потім можна прочитати або в С++ коді через відповідні методи, або в скриптові мові програмування. Прикладом мови програмування, яка є широко популярною у розробці ігор та використовує такий підхід, є Lua.

На відміну від Blueprints, Lua є мовою з динамічною типізацією, тобто написання коду відбувається без врахування типів об’єктів, що ускладнює пошук помилок та додає певні додаткові вимоги до перевірки типів під час передачі даних між віртуальною машиною та С++ кодом. Якщо в строготипізованій мові перевіркою типів займається компілятор, то в динамічній мові програмування програміст сам зобов’язаний робити перевірки на типи які приходять з віртуальної машини, та передавати конкретні типи у стек віртуальної машини [19].

Віртуальна машина Lua має необхідне API для реєстрації С++ функцій та їх імен у її віртуальному середовищі. Коли програміст використовує функції, віртуальна машина зчитує назву функції, знаходить вказівник на цю функцію у віртуальній машини та викликає С++ функцію [8]. Для виклику функції у віртуальній машині необхідно виконати такі кроки: пошук, завантаження скриптової функції та її параметрів на стек і виконання віртуальною машиною [17]. Наведемо приклади реалізації способів.

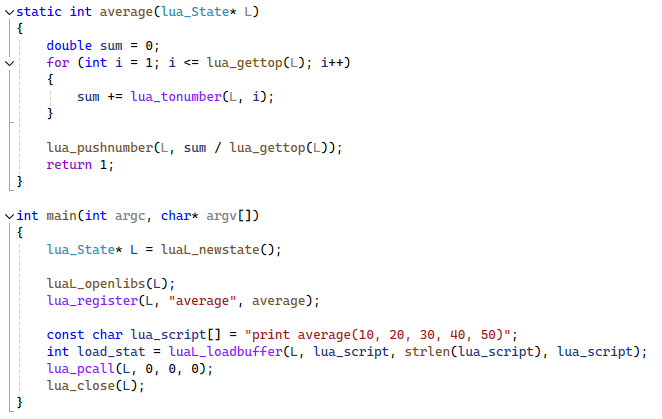


Рис. 1. Створення С++ функції для віртуальної машини

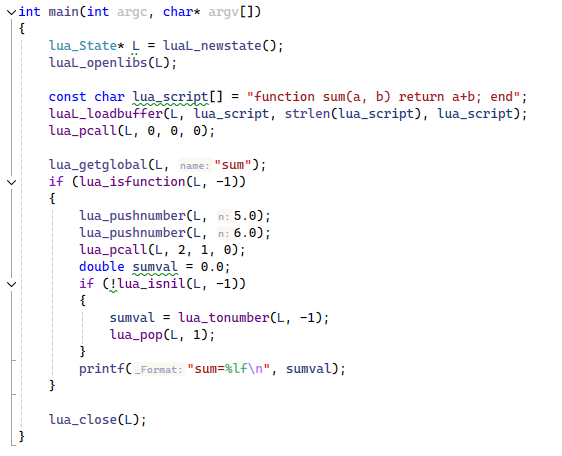


Рис. 2. Виклик Lua функції в середовищі С++

Проаналізувавши рис. 1 та рис. 2 можна побачити способи комунікації віртуальної машини та С++ коду. Відповідні функції lua\_gettop(), lua\_pushnumber(), lua\_pop(), lua\_getglobal() використовуються для завантаження чи вивантаження змінних в стек з метою обміну даних між С++ кодом та внутрішньою реалізацією віртуальної машини Lua. Схожий підхід буде використаний під час створення скриптової мови програмування в цій роботі.

Для перевірки типів в цих прикладах використовуються спеціальні функції: lua\_isfunction(), lua\_isnumber(), lua\_tonumber(), lua\_isnil(). Є велика кількість переваг мов зі статичною типізацією над мовами з динамічною типізацією, проте це накладає ряд вимог до написання компілятора скриптової мови, тому в цій роботі буде створена мова з динамічною типізацією.

**РОЗДІЛ 2. ОСНОВИ СКРИПТОВОЇ МОВИ ПРОГРАМУВАННЯ**

**2.1 Основні вимоги до скриптової мови програмування**

Є різноманітна кількість реалізованих модулів, які можуть виконувати певну частину скриптової мови програмування. Проте такі рішення не завжди легко інтегрувати в проєкт і вони можуть додатково використовувати інші залежності в собі. Наприклад різні бібліотеки Regex для реалізації сканера тексту. Наша мета – створити скриптову мову програмування, яка не буде залежати від подібних рішень, і буде мінімально використовувати сторонні бібліотеки такі як STL, Boost. Тому накладемо такий ряд обмежень для проєкту скриптової мови програмування:

1. Відмова від використання STL контейнерів, такі як std::vector, std::unordered\_map. Всі необхідні контейнери необхідно створити під конкретні завдання з урахуванням оптимізацій віртуальної машини.
2. Відмова від використання оператора new, так як деякі платформи можуть мати свої функції для алокації пам’яті. Винятком слугує «placement new», який створює об’єкт у зарезервованій пам’яті.
3. Обов’язкове використання алокатору пам’яті для мінімізації затримкою з доступом до пам’яті для нових об’єктів.
4. Відмова від використання RTTI. Деякі проєкти можуть спеціально вимикати RTTI для покращення швидкодії та мати свої способи динамічного приведення об’єктів (прикладом є Unreal Engine, який використовує свою систему рефлексії для приведення типів).
5. Створення глобальних функцій та параметрів, імплементація яких може бути перевизначена для різних платформ.

**2.2 Розробка лексера**

Код скриптової мови програмування, який пише програміст, є зручним для людського розуміння проте такий формат не є найкращим для подальшої аналізу та обробки коду компілятором. Тому першим кроком для створення компілятора скриптової мови – є створення лексера (або парсера). Цей клас сканує послідовність коду, та видає набір токенів, які вже надалі будуть використані компілятором. Для економії пам’яті лексер буде працювати за вимогою. Тобто парсер буде зчитувати новий токен коли це треба компілятору.

Першим кроком для реалізації необхідно визначити набір токенів, які будуть розпізнаватися лексером. Для цієї мови програмування, будуть доступні такі токени: {, }, (, ), “,”, “.”, -, +, “;”, /, \*, !, !=, =, ==, >=, <=, >, <, “, and, class, else, false, true, for, fun, if, nil, or, print, return, super, this, var, while, які визначені в перечисленні ETokenType [27].

Далі визначимо структуру, яка буде зберігати інформацію про токен. В цій структурі визначимо тип токена (ETokenType), початок цього символа у файлі, його довжина та номер рядка. Ці поля необхідні для того, щоб окрім типу зберігати додаткову інформацію про токен, яка надалі може бути використана. Наприклад для отримання назви змінної в коді. На рис. 3 зображена структура токена.

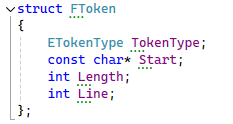


Рис. 3. Визначення структури токена

Лексер поділяє всі токени на кілька груп, які вимагають своїх правил сканування:

1. Число.
2. Текстовий рядок.
3. Визначений граматикою токен, який необхідно перевірити на точний збіг з відомими граматичними одиницями.
4. Ідентифікатор. Рядок, який починається з будь якої літери і закінчується пробілом. Ідентифікатор не може починатися з літери, але може містити цифри в кінці рядка.

Для сканування числа напишемо функцію, яка повертає токен з типом «Число» див. Рис. 4. Ця функція підтримує сканування як цілих, так і дробових значень числа. Розподільником між цілою і дробовою частиною є крапка.

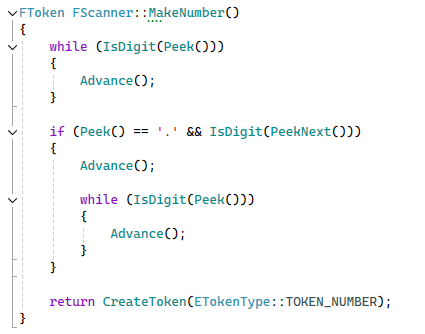


Рис. 4. Функція, яка зчитує число

Для визначення рядка необхідно знайти токен лапок з якого починається рядок і зчитувати всі наступні символи до того, як буде знайдений ще один токен лапок, який закриває цей рядок. Поточна реалізація сканера підтримує багаторядкові тексти. В деяких мовах такі рядки необхідно відмічати спеціальним токеном, який підкаже сканеру, що необхідно шукати кінець тексту в інших рядках. На рис. 5 зображено імплементацію функції для зчитування тексту.

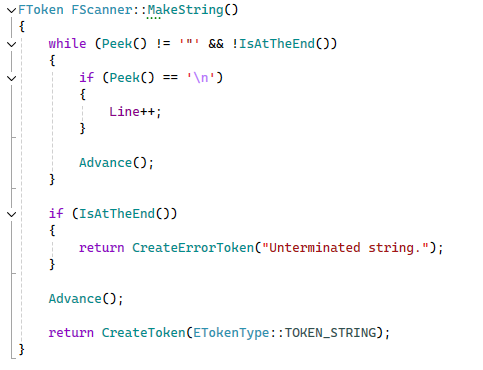


Рис. 5. Функція, яка зчитує текстовий рядок

Визначені граматикою токени можна поділити на кілька типів: символи та слова. Символи можуть містити одну літеру, або максимум 2 літери. Список таких символів є: {, }, (, ), “,”, “.”, -, +, “;”, /, \*, !, !=, =, ==, >=, <=, >, <, “. Токен, який складається з однієї літери можна визначити за допомогою простого оператора switch, але для токенів, які можуть містити один або два символи, наприклад токени > та >= (більше і більше та дорівнює), необхідно перевірити наступний символ на збіг. Якщо наступний символ співпадає, то це двохсимвольний токен, а якщо ні, то це односимвольний токен. На рис. 6 зображений код, який перевіряє односимвольні та двосимвольні токени серед набору тексту.

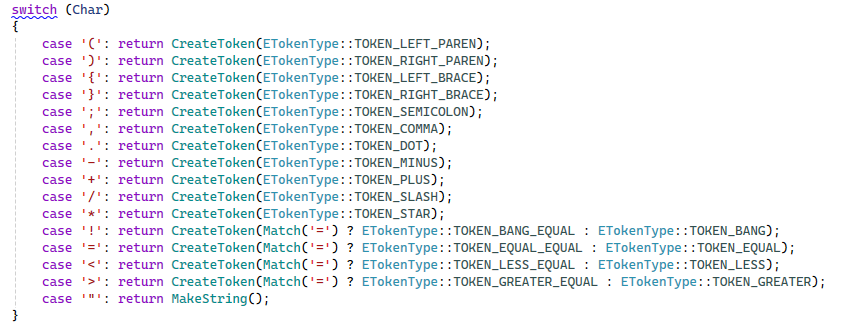


Рис. 6. Функція, яка сканує прості символи граматики

Слова є більш складними типами для сканування, так як треба враховувати визначені граматикою слова різної довжини (and, class, else, false, true, for, fun, if, nil, or, print, return, super, this, var, while), так і враховувати ідентифікатори, задані програмістом. В цьому прикладі var myTextVariable = “This is a text”; myTextVariable є ідентифікатором.

Для початку треба прочитати рядок який починається з літери, може містити числа і закінчується пробілом. Надалі треба передати цей рядок в функцію, яка реалізовує перебір серед можливих токенів. Перевірка слів використовує схожий підхід як з символами, але необхідно перевірити більшу кількість випадків. Якщо рядок не співпадає зі словами граматики, то вважаємо що це є ідентифікатором, який був заданий програмістом. На рис. 7 зображена реалізація функції перебору токенів в рядку.



Рис. 7. Функція, яка перебирає слова граматики

Отже, ця імплементація сканеру є достатньою для розпізнавання граматики простої мови програмування. Для покращення розпізнавання та створення складного граматичного синтаксису, необхідно використовувати більш просунуті сканери, проте це порушує вимогу щодо відмови від використання сторонніх бібліотек [14].

**2.3 Правила мови програмування**

Для кожної мови програмування необхідно побудувати певні правила, які будуть використані для розпізнавання синтаксису. Основне завдання компілятора – просканувати токени та визначити яка операція повинна бути виконана при певній послідовності токенів [10]. Наприклад, if ( 1 > 0 ) {} означає таку послідовність токенів: If Token, Open Parentheses Token, Identifier Token, Greater Token, Identifier Token, Close Parentheses Token, Open Brace Token, Close Brace Token. Цей набір токенів компілятор визначає як перехід за умовою, яка написана в дужках. Далі на основі токенів повернути певний результат, який буде використаний віртуальною машиною для виконання коду. Цей результат не буде містити в собі всі токени граматики, а лише необхідні інструкції та дані для виконання коду.

Компілятор буде проходити файл за файлом та зчитувати код. Тому самими першими правилами граматики, які визначають найвищу ієрархію переходів та об’єкти які складаються з інших переходів, будуть переходи зображені на рис. 8.



Рис. 8. Початкові переходи

Для спрощення написання правил визначимо такі символи: \* - необмежена кількість повторів певного переходу, ( )? – наявність або відсутність такої послідовності. Наступні переходи визначають клас, функцію, змінну та вираз зображені на рис. 9.

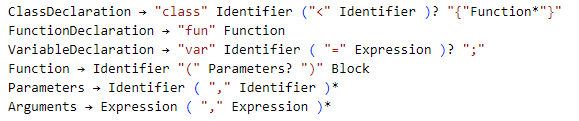


Рис. 9. Переходи класу функції, змінної та виразу.

Наступним кроком є визначення базових правил, за допомогою яких можна визначати переходи для таких виразів: for-цикл, if-else, print, return, while, block. Print є виразом для того, щоб можна було виконувати будь які операції з аргументами на етапі компіляції замість етапу виконання. На рис. 10 зображені такі переходи.

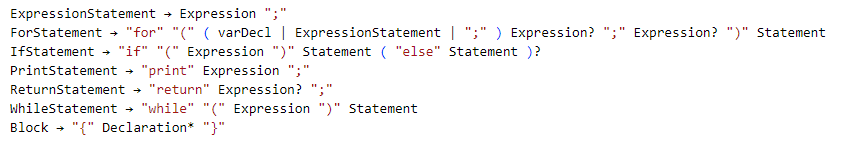


Рис. 10. Переходи для for, if-else, print, return, while, block виразів

Наступною найважливішою частиною побудови переходів є створення базових операцій з числами, булевими операціями, змінними, викликом функції, так як ці переходи мають особливості в реалізації. На рис. 11 зображені ці переходи.

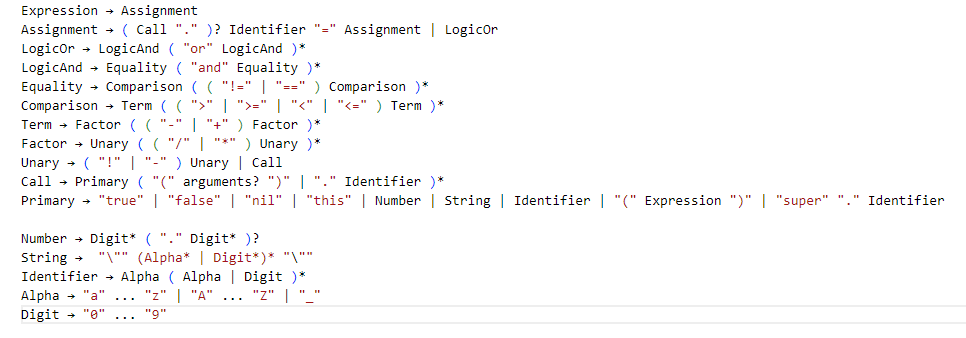


Рис. 11. Переходи для базових операцій

На рис. 8 – 10 зображені переходи, які не вимагають чітких правил пошуку переходів, так як вони починаються з конкретного токену, який визначає набір переходів. Тобто компіляцію цих переходів можна реалізувати за допомогою перевірок на послідовність токенів які зчитуються зліва-направо. Проте такий підхід не працює для переходів на рис. 11. Зчитування токенів таким чином призведе до неправильного розуміння порядку операцій. Наприклад, математична операція (3 + 4 \* (1 + 2 \* GetSomeValue())), повинна бути виконана в такому порядку:

* 1. Виклик функції GetSomeValue.
  2. Множення значення отриманого з функції на константу 2.
  3. Додавання 1 до результату множення в дужках.
  4. Множення результату на 4.
  5. Додавання до результату 3.

Зчитування зліва-направо тільки призведе до того, що компілятор спочатку спробує додати 3 до 4 і потім множити, що в результаті порушує коректність математичних операцій. Різноманітні бібліотеки мають свої підходи для розв’язання порядку операцій, проте нам заважає обмеження, яке вимагає відсутність додаткових бібліотек, тому необхідно реалізувати рішення, яке дозволяє вирішити проблему. Для цього можна використати алгоритм Vaughan Pratt «Operator Top Down Precedence» [21].

Необхідні кроки для реалізації цього алгоритму – це визначення пріоритету операцій, створення правил, які обробляють цей перехід та написання коду, який буде аналізувати токени та шукати найвищий пріоритет для операції, та виконання команди. Введемо такий порядок виконання операцій, які перерахуємо в перечисленні на рис. 12.

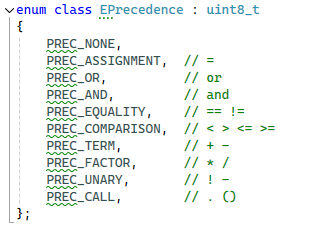


Рис. 12. Порядок виконання операцій

Визначимо формат, в якому можна задати пріоритет та методи для обробки цих токенів. Також потрібно врахувати що будь який токен може мати як пріоритетні токени перед ним, так і токени після нього. Тому для цього визначимо префікс та інфікс правило. Правила будуть визначатися як вказівник на функцію класу компілятора. На рис. 13 зображено структуру, яка визначає операції для парсингу токена.

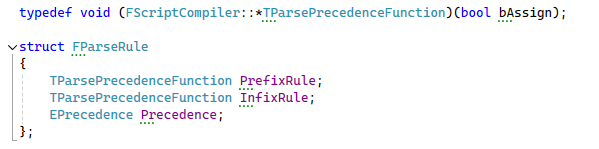


Рис. 13. Структура правила сканування токенів

Далі визначимо ці правила для токенів, які мають пріоритетність порядку виконання. Ці правила визначені в порядку зменшення пріоритетності виконання цих операцій. На рис. 14 зображено список цих операцій.



Рис. 14. Список пріоритетності операцій

Імплементація самого алгоритму не є складним завданням. Кожен токен, який має пріоритет парсингу, передає значення з перечислення пріоритетності в цю функцію. Ця функція викликає префікс правило для перечислення, проходить по списку правил і викликає всі інфікс функції, які мають вищу пріоритетність, ніж ця операція. На рис. 15 зображена імплементація цього алгоритму.

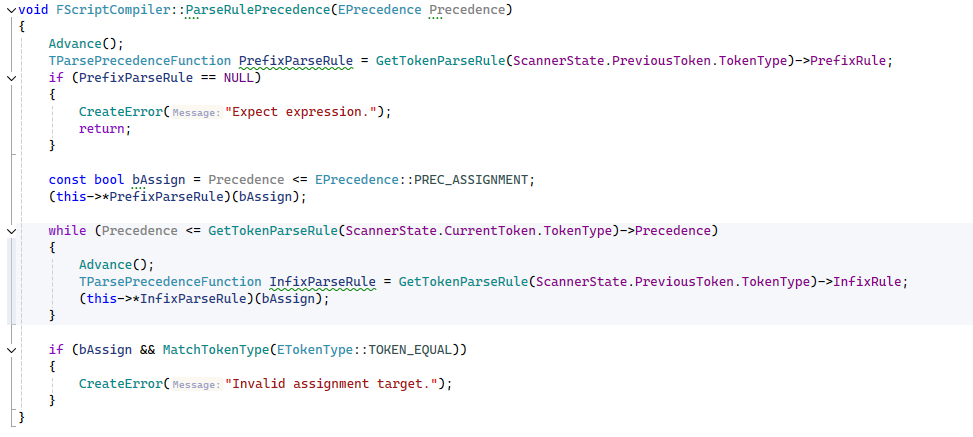


Рис. 15. Імплементація алгоритму.

**2.4 Контейнери**

Для реалізації змінних у віртуальній машині з динамічною типізацією, необхідно визначити базові типи даних, які доступні програмісту під час написання коду, і будуть інтерпретовані компілятором та виконані віртуальною машиною. На етапі компіляції типи змінних будуть визначені компілятором та будуть створені контейнери що зберігають дані та описують цей тип для подальших операцій зі змінними. Віртуальна машина підтримує такі типи даних:

1. Число. Використаємо double для того, щоб зберігати як цілі, так і дробові числа.
2. Булеві значення (істина або хиба).
3. Пусте значення (Nil).
4. Об’єкт (FGCObject та його нащадки).

Для збереження цих значень створимо спеціальний контейнер FValue. Для економії пам’яті, контейнер буде складатися з об’єднання типів та перечислення, що описує цю змінну. На рис. 16 зображено структура FValue.

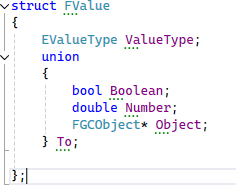


Рис. 16. Контейнер для поля віртуальної машини

Розмір цього контейнера – 16 байт з урахуванням вирівнювання пам’яті. Найбільший елемент це вказівник на об’єкт – 8 байт. Тому структура вирівнюється по найбільшому елементу. Проте таке використання пам’яті не є ефективним, так як в найгіршому випадку структура буде займати лише 9 байтів (вказівник та тип) а в найкращому - до 2 байтів (тип та булеве значення). Тому за необхідністю розмір FValue можна зменшити до 8 байт.

Зменшити розмір контейнера можна за рахунок пакування бітів у змінну. В такому випадку контейнер буде числом, яке зберігає першими двома бітами тип поля (2 біт достатньо для того щоб зберегти значення від 0 до 3, які визначені у перечисленні типів), а інші біти зберігають значення змінної. Такий підхід працює з булевим значенням і Nil, які використовують лише один байт змінної, та вказівником. Операційна система оперує вказівником об’єкту в пам’яті з зміщенням відносно виділеного блоку пам’яті, тому перші два біти вказівника завжди будуть нулем, які можна використати для збереження типу. Проте такий підхід не працює з числами так як double використовує всі 8 байтів для значення і перший біт завжди буде бітом знаку, а другий біт – першим бітом степені. Компромісом такого рішення - є використання чотирьох-байтового float замість double. Ціна такого компромісу є зменшенням точності дробового значення у віртуальній машині.

Особливістю цієї реалізації є те, що програміст зобов’язаний перевіряти за перечисленням тип, який зберігається в цьому контейнері, та взаємодіяти з ним у об’єднанні відповідно. Якщо тип не співпадає з очікуваним типом, необхідно вручну обробляти випадки передачі некоректних значень у нативні функції, аргументи функцій тощо під час виконання.

Об’єкти побудовані на наслідуванні від базового класу, з яким працює прибиральник сміття. Так як основною вимогою під час розробки була відмова від використання RTTI, тому кожен об’єкт має перечислення, яке визначає його тип. Всього існують такі типи об’єктів:

1. Bound Method - метод класу, який має доступ до вказівника на об’єкт класу.
2. Class - представлення класу у вигляді методів класу.
3. Closure - функція з замиканням, яка захоплює значення та зберігає їх протягом свого циклу життя.
4. Function - функція, яка не захоплює значення на відміну від замикання.
5. Instance – копія певного класу, який зберігає інформацію про змінні під час виконання та тримає посилання на клас з методами.
6. Native – нативна функція С++.
7. String – текстовий рядок віртуальної машини.
8. UpValue – структура, яка зберігає інформацію про значення, яке зберігається в замиканні.
9. Table – структура даних у вигляді геш-таблиці.
10. Array – структура даних у вигляді масиву.
11. Event Data – структура, яка зберігає інформацію про прив’язані елементи методом Bind.
12. Thread – структура, яка визначає поток виконання та його дані.

Аналогічно з контейнером FValue, програміст зобов’язаний вручну перевіряти тип переданого об’єкту в функцію. Надалі необхідно використати статичне приведення до типу конкретного об’єкту. Всі допоміжні функції для перевірок типу та приведення до типу знаходяться в класі FObjectCastHelpers. Динамічне приведення не підтримується через обмеження щодо використання RTTI.

Основним елементом віртуальної машини є функція. Функція є мінімальною одиницею виконання. Інші більш складні об’єкти як методи класів та замикання використовують функцію для збереження байт-коду, який виконується віртуальною машиною. На рис. 17 зображено клас функції.

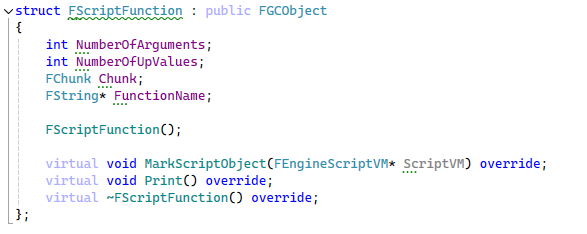


Рис. 17. Реалізація класу функції

Для збереження байт-коду функція використовує контейнер FChunk, який створюється компілятором для кожної функції. Цей контейнер зберігає масив байт коду та додаткову інформацію про функцію. Всі операційні коди, які необхідні для роботи віртуальної машини, визначені у перечисленні EOpCode і мають розмір одного байту. На рис. 18 зображено контейнер для байт-коду FChunk.

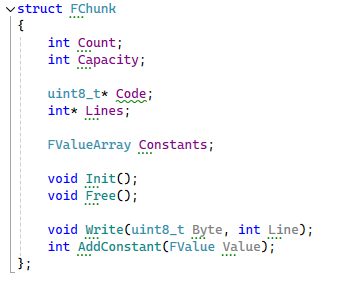


Рис. 18. Контейнер байт-коду

Функція може мати довільну кількість змінних в коді. Будь-яка змінна має довільний розмір і може повторюватися будь-яку кількість разів в коді, тому розміщення назв змінних в масиві з байт кодом не є оптимальним рішенням. Для мінімізації використання пам’яті та швидкості зчитування значень, контейнер FChunk зберігає окремо масив FValueArray з іменами цих змінних. Таким чином для взаємодії зі змінними компілятор кладе в байт-код індекс цієї змінної в масиві контейнері замість рядка.

Будь яка функція може виконувати визначені операції зі змінними, які можуть бути локальні (змінні в функції-замиканні також є локальними) та глобальними. Під час виконання віртуальна машина спробує знайти локальні дані, але якщо не вийде, то звернеться до глобального масиву змінних (глобальний простір імен).

Наступні контейнери, які використовуються у віртуальній машині є геш-таблиця та масив. Контейнери типу std::map, std::set не є оптимальними через погане використання пам’яті так як зберігають дані у вигляді дерева, що збільшує сегментацію даних.

Контейнер масив дозволяє зберігати об’єкти будь якого типу, так як всі об’єкти використовують контейнер FValue. Аналогічно з std::vector, FValueArray підтримує можливість розширення місця для даних залежно від кількості об’єктів в ньому. Для мінімізації додаткових реалокацій пам’яті, контейнер використовує окрему функцію для визначення нового розміру. Ця функція подвоює кількість елементів якщо елементів більше 8, або повертає 8 для невеликих масивів для мінімізації реалокації при невеликій кількості об’єктів. На рис. 19 наведено публічні методи масиву. Масив є необхідним контейнером як для віртуальної машини так і для розробки скриптового коду. Програміст може використовувати контейнер для збереження важливих даних у лінійному порядку.

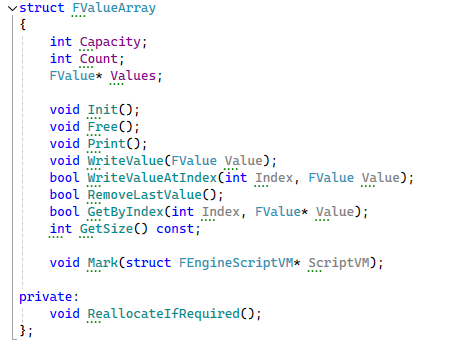


Рис. 19. Інтерфейс масиву

Так як синтаксис мови програмування не підтримує ряд операцій, які використовуються для взаємодії з масивами в будь якій іншій мові, такі як доступ до елементу за індексом (MyArray[ firstIndex ]) чи оператор розміру (MyArray.length в Javascript, #MyArray в Lua), тому реалізацію цих операцій необхідно перенести у нативні функції віртуальної машини. Програміст може взаємодіяти з таблицею через такі функції:

1. MakeArray – створює масив в пам’яті та повертає FValue.
2. WriteValue – записує значення в кінець масиву.
3. WriteValueAtIndex – записує значення за індексом.
4. RemoveLastValue – видаляє останнє значення з масиву.
5. GetByIndex – повертає FValue за індексом в масиві.
6. GetSize – повертає розмір масиву.

Геш-таблиця дозволяє зберігати значення у форматі «Ключ – Значення». В ролі ключа використовується текстовий рядок. В ролі значення контейнер FValue. Розміщення елементу визначається за допомогою геш-суми ключа. Корзина побудована за принципом зв’язаного списку для того, щоб розміщати елементи один за одним якщо геш-сума ключа вказує на одну ту саму корзину. Найбільш поширеним алгоритм підрахунку корзини - це метод залишку від ділення на розмір контейнера. Якщо розмір контейнера перевищує певну кількість об’єктів, то контейнер створює додаткові корзини.

Геш-таблиця скриптової мови програмування використовує аналогічний підхід для пошуку корзини, в яку треба покласти значення, проте особливість цієї таблиці є в тому, що в основі її реалізації лежить масив а не зв’язний список. Таким чином, це дозволяє мінімізувати зайві алокації пам’яті та пришвидшує пошук елемента в таблиці. На відміну від зв’язного списку, масив не має затримок кешу (cache miss) і підтримує можливість видалення елементу під час ітерації що є забороненим в більшості STL контейнерів так як видалення елементу призводить до інвалідації ітераторів. Для спрощення інтерфейсу контейнера ключем може бути лише текстовий рядок. За бажанням можна замінити рядок на контейнер FValue проте це вимагає написання спеціального методу, який повертає геш-суму для цього елемента в контейнері. Алгоритм пошуку елементу в таблиці побудований на таких правилах:

1. Залишок від ділення геш-суми на розмір таблиці визначає початковий індекс розміщення елементу.
2. Якщо індекс вже зайнятий іншим елементом, відбувається пошук наступної вільної корзини в масиві.
3. Якщо видаляється об’єкт, ця корзина відмічається як «мертвою» для того, щоб розмістити новий елемент.
4. Якщо геш-сума вказує на «мертву» корзину під час доступу до елементу, то контейнер намагається знайти шуканий елемент на наступній позиції якщо шуканий елемент був зміщений.

Складність такого алгоритму для додавання та видалення є O(1) в найкращому випадку коли геш-сума співпадає з елементом в корзині, та O(n) для найгіршого випадку коли необхідно шукати елемент в масиві. Якщо розмір масиву перевищує 75% від поточного розміру, таблиця робить переміщення елементів у збільшений масив та перераховує індекси корзин в масиві. На рис. 20 наведено публічні методи геш-таблиці.

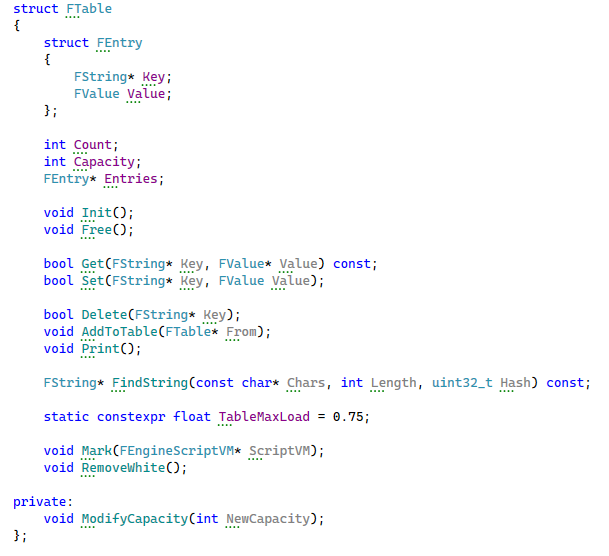


Рис. 20. Публічні методи таблиці

Аналогічно з масивом, таблиця має такі функції для взаємодії з геш-таблицею у скриптовій мові програмування:

* 1. MakeTable – створює таблицю в пам’яті та повертає FValue.
  2. AddKeyValue – додає пару ключ-значення до таблиці.
  3. RemoveKey – видалення пари ключ-значення з таблиці.
  4. GetByKey – доступ до значення за ключем.

**2.5 Розробка компілятора**

Наступний крок реалізації скриптової мови програмування – це реалізація компілятора. Компілятор це окремий клас, який аналізує набір токенів, які видає лексер на основі коду написаним програмістом, за визначеними правилами та генерує набір об’єктів, в яких описані конкретні команди. Надалі ці дані будуть використані віртуальною машиною для того, щоб виконати код.

Мета компілятора згенерувати такі команди, які віртуальна машина може виконати без жодних затримок та необхідності аналізувати текст що є досить довгою операцією. В результаті роботи компілятора, генерується байт-код, який пояснює що саме треба робити віртуальній машині. Віртуальна машина підтримує команди, які дозволяють працювати зі змінними, операціями порівняння, додавання, віднімання, множення, ділення значень, викликами різних типів функцій та виконання як прямих переходів, та переходів за умовою (if-else, while/for loop statements).

Деякі об’єкти створені компілятором під час компіляції у вигляді функцій, зв’язаних методів, класів. Інші об’єкти створені під час виконання коду віртуальною машиною такі як замикання, таблиці, геш-таблиці тощо.

Для роботи віртуальної машини необхідно визначити набір команд у вигляді перечислення, які будуть застосовані у віртуальній машині. На рис. 21 зображено перечислення цих токенів.

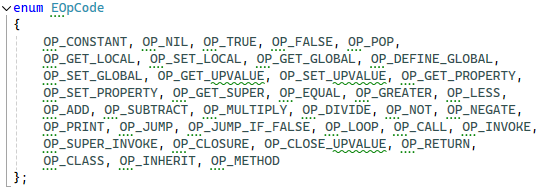


Рис. 21. Перечислення команд віртуальної машини

Ці команди вказують що саме треба робити віртуальній машині: завантажувати та вивантажувати елементи, взаємодіями зі змінними, виконувати конкретні математичні операції, створювати та видаляти об’єкти, робити прямі переходи, переходи з умовою, викликати функцію з передачею керування в цю функцію тощо [3].

**2.5.1 Замикання**

Особливістю цього компілятора є підтримка замикань (closure). Замикання є досить популярним серед мов програмування з підтримкою динамічної типізації. Замикання підтримуються також в С++, проте з певними особливостями. Замикання – це механізм, який дозволяє зберігати значення, яке може бути за межами певного блоку, та оперувати ним в будь який момент в цьому блоці.

Особливість замикання в тому, що воно тримає посилання на цей об’єкт, який знаходиться в купі віртуальної машини, тому прибиральник сміття не може видалити цю змінну поки не буде видалене замикання. На рис. 22 зображено функцію, яка повертає функцію що захопила локальну змінну та оперує з нею під час викликів внутрішньої функції [2].

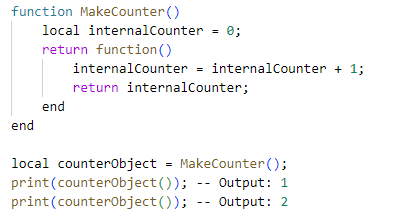


Рис. 22. Приклад замикання в Lua

Такий механізм реалізований за рахунок того, що кожна функція в представленні віртуальної машини може мати свій масив змінних, в який можуть бути додані певні змінні, шо створені в конкретному просторі змінних (наприклад на стеку функції MakeCounter).

В С++ подібне замикання зробити складніше через вимоги до синтаксису та особливостями моделі пам'яті. Мова програмування не підтримує ідею завантаження змінної в пам'ять функції, але проте можна зберегти дані в тілі лямбда-функції. Компілятор перетворює лямбду у спеціальний клас-функтор, який зберігає певний набір даних у вигляді змінних класу, які передані у блок захоплення даних лямбда-функції. Також, С++ не має прибиральника сміття, тому всі параметри передані в лямбда-функції вимагають ручного керування об’єктами, або використання розумних вказівників, які можуть вести підрахунок на цей об'єкт. На рис. 23 зображена реалізація об’єкту, який підраховує кількість його викликів [6].

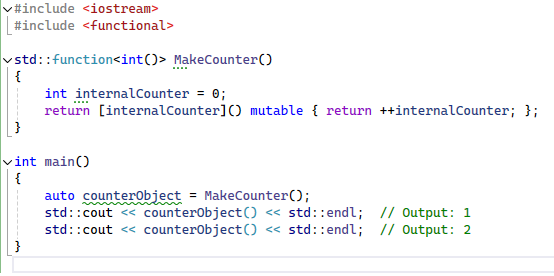


Рис. 23. Приклад замикання в С++

**2.5.2** **Обробка помилок**

Компілятор може працювати лише з правильним синтаксисом коду, але якщо код має недоліки, то компілятор повинен видати помилку у випадку невідповідності синтаксису з правилами переходів формальної граматики. Кожна мова програмування має свої способи для виявлення та повідомлення програміста про помилки синтаксису. Більш просунуті рішення мають окремі програми, які перевіряють синтаксис згідно чітко визначених правил. Скриптова мова програмування не використовує подібні програми, а робить перевірку синтаксису під час компіляції коду.

Будь яка помилка в коді призводить до неправильної роботи самої програми, тому завдання компілятора сповістити програміста належним чином про цю помилку. Мови програмування використовують різні підходи, але їх можна узагальнити до таких стратегій [24]:

1. Якщо знаходиться помилка, то зупиняється компіляція коду.
2. Якщо знаходиться помилка, то компілятор продовжує компіляцію наступного коду незважаючи на це.
3. Якщо знаходиться помилка, то компілятор пропускає проблемний блок коду та продовжує подальшу компіляцію.

Мінус першого підходу полягає в тому, що компілятор буде зупинятися після знаходження першої помилки і сповіщати про неї, полягає в тому, що в коді може бути більше однієї помилки що змусить програміста виправляти помилки поступово.

Мінус другого підходу полягає в тому, що одна помилка може збільшити кількість наступних помилок, які були створені через одну помилку, і які будуть більш складними в розумінні програмісту. Особливо цей підхід ускладнює розуміння помилок під час підстановки шаблонних функцій чи шаблонних класів. Підхід характерний для мов С та С++.

Третій підхід намагається знайти компроміс між обома підходами. Компілятор перевіряє код до поки не знайде помилку. Якщо помилка знайдена, то компілятор пропускає компіляцію певного блока, де може виникнути ця помилка, та продовжує процес далі. Таким чином компілятор виведе найбільшу кількість знайдених помилок в коді з чітким поясненням помилки. Мінусом підходу є те, що компілятор не знайде іншу помилку в блоці коду, який був пропущений компілятором. Блоком коду можна вважати: клас, функція, кінець рядку, цикл тощо.

Окрім помилок компіляції ще існують помилки виконання. Помилки виконання з’являються через помилково написаний код. Наприклад, передача неправильного аргументу у функцію, виклик не створених функцій, арифметичні операції над нечисельними типами, тощо. Деякі типи помилок можуть бути знайдені на етапі компіляції, проте це вимагає розширення самого компілятора, тому простіше такі помилки обробляти під час виконання. Якщо віртуальна машина знаходить помилку під час виконання, то виконання коду аварійно завершується [11].

Віртуальна машина не підтримує методи перехоплення помилок під час виконання такі як try-catch(-finally) в різних мовах програмування як C++, Java, Python. За необхідністю можна додати блоки коду try-catch, які будуть безпечно виконуватися та не зупиняти роботу віртуальної машини, або модифікувати віртуальну машину таким чином, що вона буде ігнорувати такі помилки під час виконання. Наприклад виводити інформацію про помилку та робити ранній вихід з функції або продовжувати виконання далі. Останній варіант є непоганим рішенням для написання коду, який не повинен аварійно завершуватися під час роботи ігрового рушія.

**2.5.3** **Підтримка ООП**

Ігрові рушії побудовані з використанням об’єктно-орієнтованого програмування з метою максимального розширення функціоналу та систем. Перші рушії використовували процедурне програмування на С, проте сама мова мала велику кількість недоліків, які створювали проблеми під час розробки: відсутність інструментів для роботи з рядками (в С++ використовується std::string), повністю ручне керування вказівниками та різними масивами (в С++ використовуються розумні вказівники для автоматичного керування або std::vector), складність поєднання об’єктів, які мають спільні риси (в С++ використовується наслідування), тощо.

Скриптові мови програмування зазвичай наслідують об’єктно-орієнтований стиль різними способами. Наприклад, Blueprints рушія Unreal Engine повністю підтримують всі концепції ООП: наслідування, абстрагування, поліморфізм, інкапсуляція, що дозволяє писати повноцінний ООП код. Мова програмування Lua, яка є дуже поширеною мовою у рушіях, частково підтримує ООП. Скриптова мова підтримує абстрагування, поліморфізм, наслідування за рахунок використання мета-таблиць, які надають можливість визначити батьківські методи класу та поля класу.

Основною одиницею ООП у віртуальній машині є клас. Різниця між структурою та класом в мові програмування С++ є мінімальною, тому додавання структури не є необхідним. За необхідністю можна зберігати дані у вигляді геш-таблиці для швидкого доступу замість класу. Для спрощення синтаксису класи не будуть мати модифікатори доступу (public, protected, private) для методів та параметрів і під час наслідування від інших класів. Всі функції класу є віртуальними автоматично. Таким чином можна перевантажувати будь який метод без зайвих модифікаторів (virtual, override), які необхідні в мові програмування С++. Проте відсутність модифікатора override унеможливлює перевірку що правильна функція перевантажена в класі нащадку. Наступною особливістю класу це доступ до полів класу через this (вказівник на дані класу, який неявно передається у кожен метод класу) та super. Super дозволяє отримати доступ до полів та методів батьківського класу.

Основне завдання по визначенню класів покладено саме на компілятор. Він повинен визначити всі можливі класи, методи класів, наслідування від інших класів. Мова С++ чітко визначає концепцію класів як набір незалежних методів та екземпляр класу, який зберігає винятково дані класу. Виклик методу класу компілятор С++ записує у вигляді «Метод Класу(Вказівник на екземпляр класу, [аргументи функції]). Таким чином компілятор мінімізує необхідність створювати копії функцій і мінімізує розміри екземплярів класу лише до даних, які змінюються під час виконання. Аналогічно з реалізацією класів в С++, розділимо класи віртуальної машини на визначення класів та екземплярами. Сам клас визначимо у вигляді об’єкту, який зберігає назву класу, методи класу. На рис. 24 наведено клас у віртуальній машини. Для виклику Super методу та мінімізації розміру класу, компілятор кладе назву класу в байт-код перед викликом батьківського методу.

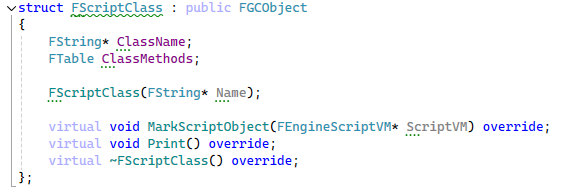


Рис. 24. Клас віртуальної машини

Сам екземпляр класу зберігає вказівник на методи класу, та всі дані цього класу. На відміну від С++, скриптова мова програмування не вимагає чітко декларувати всі поля класу на етапі компіляції і дозволяє додавати будь які змінні до класу під час виконання коду віртуальною машиною так, як віртуальна машина зберігає геш-таблицю даних класу і не вимагає чіткого розміру класів для подальшого генерування машинного коду з відомими відступами для доступу до елемента класу. Для доступу до даних класу, програмісту необхідно додавати this перед кожною зміною, тим самим прибрати ймовірність що віртуальна машина знайде глобальну змінну з аналогічною назвою. Таким чином можна додатково перевірити під час виконання чи змінна належить до класу чи ні. На рис. 25 наведено визначення екземпляру класу.

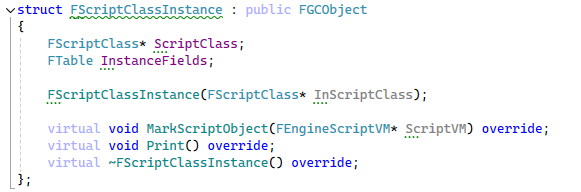


Рис. 25. Екземпляр класу віртуальної машини

**РОЗДІЛ 3. ВІРТУАЛЬНА МАШИНА**

**3.1 Особливості віртуальної машини**

Віртуальна машина це клас, який виконує байт-код, який генерується компілятором. Вона має свої особливості виконання байт-коду та використовує схожі підходи з процесором персонального комп’ютера, а подекуди має великі відмінності від процесора.

Перша особливість полягає в тому, що віртуальна машина відповідає за керування ресурсами, які створюються під час виконання коду. Це мінімізує потребу програмістом слідкувати за життям об’єктів. Аналогічний код мовою С++ вимагає чіткого контролю за об’єктами та часом їхнього знищення, слідкуванням за «мертвими» посиланнями, доступ до яких потенційно може призвести до аварійного завершення програми. Такі помилки можуть бути помічені не відразу а лише при деяких умовах, що ускладнює пошук та їхнє виправлення. Частково ці проблеми вирішують розумні вказівники, проте вони мають свої особливості використання.

Друга особливість в тому, що весь код виконується у безпечному середовищі, яке не містить в собі операцій, які дозволяють виходити за обмеження віртуальної машини [5]. Такими операціями є:

1. Маніпуляції з вказівниками характерних для С/С++. Більшість мов програмування намагаються максимального обмежити операції з вказівниками такі як Rust, Golang або прибрати їх з використання такі як Java, Python.
2. Обмеження способів комунікації віртуальної машини з операційною системою. Мови низького програмування можуть оперувати з стеком, регістрами, викликати будь-які функції, які можуть мати певні недоліки створеними розробниками операційної системи, тощо. Таким чином хакер отримує можливість перехоплення керування та виконання будь якого шкідливого коду. Віртуальна машина може виконувати лише визначений байт-код і викликати функції, які були додані до віртуальної машини методом реєстрації нативної функції.
3. Операції приведення типів. Деякі операції приведення до типів можуть бути небезпечними. Наприклад, використання static\_cast для приведення класу до одного з його нащадків, reinterpret\_cast для приведення вказівників одного типу до вказівника іншого типу. Оператор reinterpret\_cast є одним з найбільш небезпечних методів, так як він не робить жодних перевірок на можливість подібного приведення типів. Оператор static\_cast навпаки робить перевірку чи є метод, який реалізує таке приведення, та викликає його. Сonst\_cast є повністю безпечним так як він знімає константність, яка використовується лише на етапі компіляції. Оператор dynamic\_cast робить додаткові перевірки щодо можливих приведень до певного типу класу нащадка. Більшість мов програмування не підтримують таким список способів приведення до типу і мають більш прості та безпечні механізми для перевірки правильності операцій приведення.
4. Відсутність способів ручної аллокації пам’яті. Мови С/С++ не мають ніякого захисту від надмірної або неконтрольованої алокації пам’яті та способів автоматичного видалення пам’яті на відміну від віртуальної машини, яка автоматично резервує пам’ять під конкретні об’єкти без можливості створювати будь-які нестандартні блоки даних.

Третя особливість полягає в тому, що віртуальна машина може робити додаткові перевірки під час компіляції та виконання коду для забезпечення коректності коду. Мова програмування С++ має великий набір інструментів для статичного пошуку таких помилок, проте навіть вони не гарантують що весь код правильний і не має помилок під час виконання. Наприклад, код мовою С++ на рис. 26 може призвести до помилки виконання.

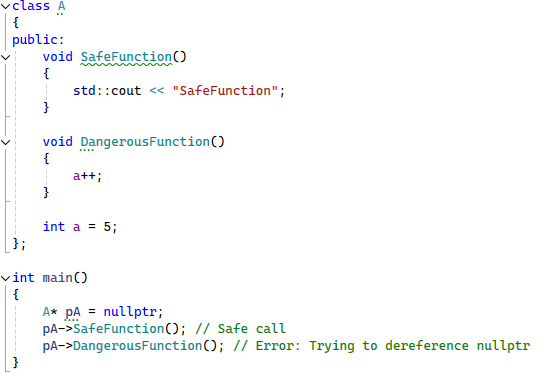


Рис. 26. Небезпечний код мовою С++

В цьому прикладі, програміст намагається викликати функцію для неіснуючого об’єкту. Перша функція буде викликана успішна так як вона не використовує this і не відбувається доступ до вказівника класу. А друга функція намагається отримати доступ до змінної класу за нульовим вказівником що призводить до помилки виконання. Віртуальна машина для мінімізації таких помилок автоматично з’єднує this з функцією та перевіряє тип функції таким чином мінімізує помилки.

**3.2 Виконання машинного коду**

Для виконання байт-коду віртуальна машина використовує схожі підходи що й процесор, проте спрощує певні аспекти та автоматизує процеси, які відбуваються з об’єктами. Віртуальна машина буде завжди повільніша за машинний код, проте правильно написаний та оптимізований код може показати досить високу продуктивність. Виконання коду процесором відбувається у вигляді такого алгоритму [1]:

* 1. Зчитування команди.
  2. Дешифрування команди.
  3. Завантаження даних в кеш процесора.
  4. Виконання інструкції.
  5. Запис результатів виконання команди.
  6. Інкрементація регістру інструкції (EIP, RIP залежно від розрядності операційної системи).

Ці етапи виконуються один за одним різними компонентами процесора (Control Unit, Arithmetic Logic Unit, Address Generation Unit, Floating Point Unit), та можуть робити окремі запити до регістрів, кешу процесора або завантажувати блоки даних в кеш процесора. Відсутність потрібних даних в кеші процесора є Cache Miss, що змушує процесор чекати завантаження даних блоку даних з оперативної пам’яті. Ще кількома важливими оптимізаціями процесора є конвеєризація та передбачення переходів. Проте такі оптимізації досить складно реалізувати у віртуальній машині так як вона може працювати лише з однією інструкцією в певний момент часу.

Віртуальна машина має схожий алгоритм виконання команди, проте етапи 3-5 (завантаження даних, виконання команди, запис результатів) об’єднані в один етап виконання. Наприкінці, віртуальна машина збільшує адресу виконання байт-коду на певну кількість кроків (аналог регістру RIP процесора) [26]. Переходи на певну кількість інструкцій використовуються у циклах, умовах, тощо. «Регістр» інструкцій віртуальної машини вказує на адресу поточного блоку байт-коду (FChunk).

Аналогічно з процесором, віртуальна машина має свій стек, в який вона кладе поточні значення цього фрейму. У випадку передачі керування, тобто виклику нової функції, віртуальна машина видає новий стековий фрейм з масиву зарезервованих стекових фреймів. Віртуальна машина підтримує максимальну глибину у вигляді 128 стекових фреймів. Перевищення цього значення призводить до помилки «Stack overflow». На відміну від процесора, віртуальна машина зберігає адресу виконання у структурі, яка визначає стековий фрейм. Віртуальна машина під час переключення контексту бере самий верхній фрейм та виконує інструкції на які посилається «регістр». Процесор навпаки зберігає адресу повернення на стеку та відновлює значення при поверненні керування [20].

Локальні дані функції зберігаються в неперервному масиві значень віртуальної машини. Кожний стековий фрейм зберігає вказівник на початок змінних, які належать йому. Після виходу з функції всі локальні змінні від початку значень фрейму до вершини стеку додаються в чергу на знищення. Об’єкт буде повноцінно знищено під час запуску прибиральника сміття. В даному випадку використовується метод відкладеного знищення об’єктів.

Віртуальна машина не підтримує будь-які Calling Convention, які використовуються в різних компіляторах та операційних системах, що дозволяє оптимізувати передачу певну кількість аргументів до функції через більш швидкі регістри та повільний стек, або через основні та додаткові регістри, або лише через стек, та способи повернення результатів виконання функції. Віртуальна машина не містить регістрів, тому передає всі змінні у функцію через стек та повертає результат виконання в стеку. Синтаксис мови програмування підтримує повернення лише одного аргументу з функції. За необхідністю для повернення більшої кількості аргументів, можна використовувати масив чи геш-таблицю.

Віртуальна машина побудована за принципом «виконання коду поки не буде викликано завершення виконання». Тобто в основі знаходиться цикл який містить switch-case statement. На рівні машинного коду switch вирази є більш оптимізованими так як вони роблять прямі переходи ніж аналогічний if-else if-else statement. Switch порівнює зчитану команду в масиві байт-коду з відомими командами та виконує операції читання та запису зі масивом змінних та стековими фреймами [25].

Також віртуальна машина намагається оптимізувати використання текстових рядків для мінімізації аллокацій пам'яті під кожну копію одного й самого рядка. Цей механізм реалізований за рахунок збереження всіх текстових рядків в окремому масиві значень віртуальної машини. Таким чином віртуальна машина перевіряє чи такий рядок існує у зарезервованій пам’яті та повертає створений рядок. Якщо такого рядку не існує, то додає до масиву.

**3.3 Реалізація прибиральника сміття**

Основною перевагою будь-якої скриптової мови програмування у порівнянні з С/С++, є присутність прибиральника сміття. Програміст обмінює максимальний контроль над об’єктами та пам’яттю на автоматичний контроль ціною часу, який займає прибиральник сміття для визначення «живих» та «мертвих» об’єктів.

Основна проблема ручного керування полягає у «висячих вказівниках» (Dangling Pointers) та витоку пам’яті (Memory Leak). Перша проблема виникає коли програміст створює об’єкт та поширює вказівник на нього серед інших об’єктів. В результаті після ручного видалення цього об’єкту, всі інші об’єкти, які тримають вказівник на нього, можуть про це не знати, що в результаті призводить до того, що об’єкт видалений, але вказівник вказує на повернену пам’ять операційній системі, або переданій у використання іншим об’єктом (якщо об’єкт був створений в певному пулі пам’яті, яким володіє аллокатор). В результаті доступ до цієї адреси може призвести до помилки виконання або аварійного завершення виконання програми. Бібліотека мови С++ пропонує використовувати розумні вказівники, які підраховують кількість посилань на об’єкт. Мова С подібних рішень немає, так як в основі розумних вказівників використовується ідіома RAII, що не підтримується компілятором мови. Розумні вказівники мають свої недоліки при неправильному користування. При неправильному проєктуванні архітектури можна створити ситуацію коли два об’єкти тримають розумний вказівник один на одного і таким чином автоматично продовжують життя один одного, що призводить до другої критичної проблеми мов С/С++ - витік пам’яті.

Проблема витоку пам’яті полягає в тому, що пам’ять, яку використовує програма, не повертається до операційної системи коли об’єкт стає непотрібним, а всі посилання на цей об’єкт втрачено, що унеможливлює подальшої видалення цієї пам’яті. В результаті накопичуються об’єкти, які вже не використовуються але займають пам’ять операційної системи, яка має обмеження. Подібна ситуація стає критичною для випадків коли від програми очікують довгий час роботи без перезапусків чи перезавантаження операційної системи. Програмісти намагаються боротися з цією проблемою використовуючи спеціальні програми «санітайзери», які вбудовуються в код та відслідковують всі аллокації та деаллокації пам’яті. В результаті виконання програміст може дослідити логи програми на витоки пам’яті. Також аллокатори можуть підраховувати всю звільнену та не звільнену пам’ять на момент закриття програми та сповіщати про витоки.

Завдання прибиральника сміття мінімізувати ці проблеми під час написання коду. Тому в скриптових мовах програмування з підтримкою прибиральника сміття відсутня можливість вручну створювати пам’ять.

Прибиральник сміття повинен працювати з певним пулом пам’яті, який мінімізує сегментацію пам’яті. Тому для вирішення цих проблем скористуємося двома фіксованими пулами пам’яті: перший пул – для об’єктів віртуальної машини, другий пул – для будь яких масивів об’єктів чи текстових рядків.

Під час виконання коду скриптової мови об'єкти можуть посилатися один на одного та створювати залежності між об’єктами. З точки зору прибиральника сміття ці об’єкти можуть створювати цикл між собою, що може призвести процес прибирання пам’яті до безкінечного переходу від одного об’єкту до іншого і навпаки. Тобто алгоритм Mark-and-Sweep не підходить для віртуальної машини саме через можливість зациклювання процесу прибирання сміття [16]. Оптимальним алгоритмом є - Tri-color marking. Цей алгоритм розподіляє об’єкти на певні групи:

1. Біла група - це група об'єктів, які є потенційними кандидатами на видалення.
2. Чорна група - це група об'єктів, які не мають вихідних посилань на об'єкти з білої групи і можуть бути досягнуті з коренів. Об'єкти з цієї групи не є кандидатами на видалення.
3. Сіра група – це група об’єктів, які доступні з коренів, але ще не перевірені на наявність посилань на «білі» об'єкти. Оскільки вони доступні з коренів, їх не можна видаляти, і після ітерації прибиральника сміття вони потраплять до чорної групи.

Надалі таке розподілення на групи дозволяє реалізувати покращений алгоритм збірки сміття. Алгоритм працює наступним чином:

1. Вибірка будь якого об’єкту з сірої групи.
2. Переміщення кожного білого об'єкту, на який є посилання від кореневого об’єкту, до сірої групи. Це гарантує, що цей об'єкт та об'єкти, на які він посилається, не можуть бути видалені. Перевірені об’єкти ігноруються.
3. Переміщення об’єкту до чорної групи.
4. Повторення цих кроків, доки сіра група не стане порожньою.
5. Ітерація по всім створеним об’єктам та перевірка стану. Чорні об'єкти доступні з коренів, а білі об'єкти – ні. Тому ці об’єкти видаляються.

Такий алгоритм може поступово дослідити всі об’єкти у віртуальній машини та знайти «мертві» об’єкти [22]. Для початку виконання алгоритму необхідно визначити початковий стан сірої групи (або кореневі об’єкти). Для віртуальної машини кореневими об’єктами, з яких починає працювати прибиральник сміття, є:

* 1. Таблиця зарезервованих текстових рядків.
  2. Стек віртуальної машини.
  3. Функції віртуальної машини.
  4. Активні функції-замикання віртуальної машини.
  5. Класи та батьківські класи.
  6. Глобальні об’єкти.
  7. Фрейми віртуальної машини.
  8. Геш-таблиці з потоками та подіями.

Такий алгоритм також має певні недоліки. Основний недолік прибиральника сміття полягає в тому, сам процес прибирання сміття зупиняє виконання байт-коду що призводить до паузи виконання коду, що може бути досить критичним для роботи з об’єктами ігрового рушія у багатопоточному середовищі. Є кілька способів для мінімізації подібних затримок:

* 1. Використання багатопоточного процесу прибирання сміття, яке дозволило б розподілити навантаження на більшу кількість потоків одночасно та зменшити затримку.
  2. Фіксований час прибирання сміття. Такий підхід виконує прибирання сміття в окремий проміжок часу, коли пауза не впливає на виконання коду та не шкодить багатопоточності рушія. Наприклад, ігровий рушій Unreal Engine може запустити прибиральник сміття в кінці тіку та продовжувати прибирання протягом кількох наступних тіків (Time Slicing).
  3. Видалення об’єктів відразу як тільки вони виходять за межу видимості. Об’єкт видаляється як тільки кількість посилань на цей об’єкт дорівнює нулю. Цей підхід використовує ідею Reference Counting.
  4. Надання можливості програмісту самостійно вирішувати коли треба видалити об’єкт. Проте цей підхід вимагає пошуку всіх посилань на об’єкт та занулення цих об’єктів.

Останні два пункти видаляють об’єкти з пам’яті миттєво, а не чекають коли вільного місця в аллокаторі стає мало і запускається прибиральник сміття. Такі підходи є більш оптимальними для відслідковування пам’яті, проте вимагають переписування віртуальної машини і більш чіткого контролю над створеними об’єктами та їхнім циклом життя на етапі компіляції. Такі підходи реалізовані на рівні бекенду компілятора в мовах програмування Swift, Rust.

Проте такі особливості прибиральника сміття додають обмеження для реалізації ідіоми RAII, так як прибиральник сміття не дає гарантій що об’єкт буде видалений як тільки він перестає бути потрібним. Тому програміст зобов’язаний думати щодо реалізації додаткових функцій, які б звільняли ресурси об’єкту перед його видаленням. Наприклад, після закінчення гри або після сповіщення про певну події, віртуальна машина викликає функцію, в якій програміст може видалити всі створені об’єкти.

**РОЗДІЛ 4. МЕХАНІЗМИ ДЛЯ СТВОРЕННЯ ІГРОВОЇ ЛОГІКИ**

**4.1 Реалізація оновлення стану гру в рушії**

Будь яка програма побудована на ідеї очікуванні певної події. Наприклад, Microsoft Word очікує на подію натиску кнопки, браузер очікує на подію руху мишки, програма очікує на подію переключення фокусу, тощо. В основі обробки події використовується цикл. Програма з певною періодичністю робить запити та обробляє результати поки не отримає запит на завершення виконання програми.

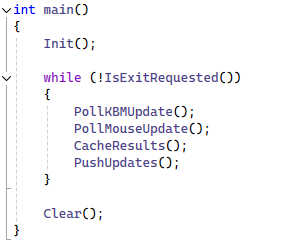


Рис. 27. Оновлення подій системи в програмі

Код наведений на рис. 27 збирає інформацію про всі натиснуті кнопки на клавіатурі, переміщення мишки та її натиснуті кнопки. Потім ці дані зберігаються в окрему структуру даних. Після цього викликається функція, яка перевіряє що нові дані відрізняються від тих, які були отримані в результаті попередньої ітерації циклу та делегує зміни на об’єкти, які очікують ці події. Також збережені дані можуть бути використані для того, щоб компоненти програми могли в будь який момент дізнатися поточний стан клавіш.

Iгровий рушій використовує такий самий підхід для оновлення стану клієнта гри та його компонентів, проте має обмеження часу одного оновлення. Рушії використовують фіксоване або гнучке число, яке визначає кількість оновлень стану гри за одну секунду. Це число визначається вимогою платформи, налаштуваннями користувача або обмежене потужністю процесора та графічної карти. Таким чином, консолі минулого покоління (PS4, Xbox One) можуть запускати гру з фіксованою частотою оновлення 30 разів в секунду, а більш сучасні ПК можуть оновлювати гру від 30 до 240 разів в разів в секунду. Надалі будь яке додаткове оновлення не має жодної користі, так як більшість моніторів не можуть оновлювати картинку швидше за 240 разів в секунду і в результаті обрахований стан гри буде ігноруватися.

Цикл в ігровому рушії є більш комплексним та оновлює велику кількість підсистем рушія з певною періодичністю, так як є системи, які вимагають підвищеної частоти оновлення, а є такі які, не вимагають високої частоти. Прикладом систем, які вимагають підвищену частоту оновлення є:

1. Система фізики. Ця система вимагає оновлення вдвічі більше за частоту відображення гри на моніторі. Така частота необхідна для того, щоб рушій правильно розраховував проміжні етапи фізичних об’єктів між кадрами.
2. Система анімацій. Аналогічно з системою фізики, анімації вимагають підвищеної частоти оновлення для коректного відображення скелетного мешу.

Вище наведені системи виконують дуже важливу роль в ігровому рушії та високу частоту оновлення, проте є системи, які не вимагають подібної частоти оновлення. Наприклад:

* 1. Система оновлення стану гри та світу. Ця система забезпечує оновлення акторів кожен кадр гри. За необхідністю певні групи акторів можуть мати свою частоту оновлення.
  2. Система мережевого коду. Ця система забезпечує отримання та відправлення пакетів. Система не має певної частоти оновлення, а лише чекає на події мережевого драйвера операційної системи та обробляє отримані результати. Надалі ці дані будуть використані системою оновлення гри.
  3. Система відображення світу. Ця система працює послідовно після оновлення стану гри та може мати кілька етапів.
  4. Система штучного інтелекту. Ця система є найбільш гнучкою з точки зору частоти оновлення, так як система використовує набір команд з низькою частотою оновлення. Операції ігрового інтелекту є складними для обрахунку та не є критично важливими, які б заважали користувачу грати в гру.

Всі ці системи використовують потоки процесора для виконання обчислень. Кожний ігровий рушій має свою реалізацію багатопоточного виконання коду залежно від архітектури рушія та систем. Багатопоточність побудована з мінімізацією блокування потоків, тому рушії визначають етапи оновлення систем. Зазвичай існують такі етапи: Pre-Physics, Physics Update, Physics End, Post Physics проте більшість рушіїв мають додаткові етапи. На рис. 28 зображено етапи оновлення рушія Unreal Engine 5.

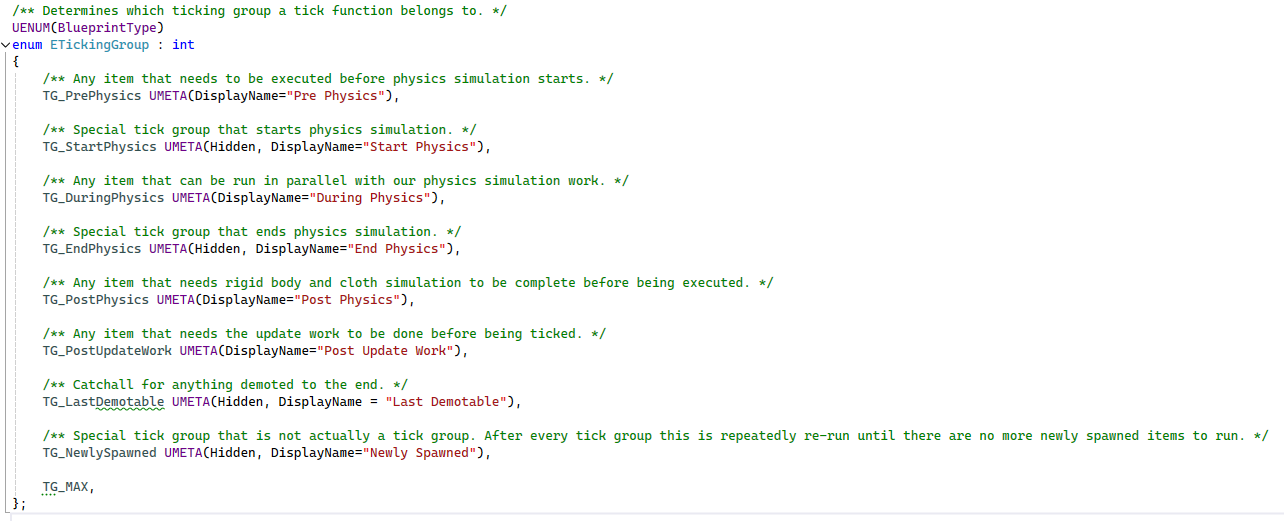


Рис. 28. Етапи оновлення систем рушія

Всі ці етапи рушія Unreal Engine використовують певну кількість потоків операційної системи. Ігровий рушій може контролювати який код виконується на потоці, проте не може керувати коли і на якому ядрі процесора виконується код. Проте є рішення, які можуть вручну керувати виконанням коду на фіксованій кількості ядер процесора та процесом переключення контексту виконання на цих ядрах. Такий підхід полягає в тому, що операційна система дозволяє програмісту зарезервувати певні ядра для рушія та підняти максимально пріоритет виконання для того, щоб ядро ОС не могло за вимогою інших програм переключати контексти виконання. Схожим рішенням подібного переключення контексту в бібліотеці STL є std::coroutine. За необхідністю розробники можуть реалізувати своє рішення з необхідними вимогами рушія.

Рушій Unreal Engine визначає такі типи потоків: Main Thread, Render Thread, Network Thread, Physics Thread. Ці потоки відповідають за різноманітні завдання рушія та делегування виконання певного завдання у пул потоків (Worker Threads).

* 1. Physics Thread перевіряє чи створено AABB Tree та розподіляє математичні обрахунки певної під гілки дерева на пул потоків. В результаті велика кількість завдань виконується паралельно, а пізніше результати обрахунків будуть оброблені на наступному етапі оновлення стану ігрового рушія [9].
  2. Main Thread відповідає за оновлення стану гри, таймерів, оновлення всіх об’єктів в світі (виклик функції Tick) тощо. Такий поток не використовує додаткові потоки з пулу, проте програміст може додати паралельні обчислення в коді.
  3. Network Thread відповідає за отримання та відправлення пакетів на сервер та очікування нових пакетів. Такий поток працює в режимі очікування і не вимагає додаткових потоків.
  4. Render Thread підготовлює певний набір об’єктів, які необхідно обробити, та передає цей набір до робочих потоків, які вже підготують дані для графічної карти.

Таким чином рушій розподіляє оновлення стану гри, яке відбувається більше ніж 30 разів на секунду використовуючи певні оптимізації коду та паралельне виконання систем.

**4.2 Огляд механізмів для створення ігрової логіки**

Проаналізувавши систему багатопоточного оновлення стану гри, необхідно визначити де саме виконується код віртуальної машини. Мова програмування використовується для розширення ігрового коду, який виконується на основному потоці рушія разом з оновленням стану світу. Тому потенційними варіантами виконання коду віртуальної машини є:

* 1. Оновлення віртуальної машини послідовно з виконанням коду рушія. Проте такий підхід має мінус в тому, що оновлення одного об’єкту в світі може зайняти двічі: оновлення об’єкту у віртуальній машині та оновлення цього об’єкту в ігровому світі.
  2. Оновлення віртуальної машини відбувається на окремому потоці та на початку кожного потоку відбувається синхронізація результатів виконання скриптів.

В цій роботі реалізований другий варіант, який дозволяє передавати дані та виконувати певну логіку між кодом рушія та віртуальною машиною, за допомогою таких методів:

* 1. Нативні функції, які реєструються у віртуальній машині та доступні для виклику під час виконання байт-коду.
  2. Пересилання подій між віртуальною машиною та С++.
  3. Потоки виконання з можливістю налаштування періодичності виклику потоку.

Нативні функції є основним способом комунікації між віртуальною машиною та середовищем рушія. За допомогою таких функцій можна реалізувати передачу даних з віртуальної машини в рушій, та навпаки, отримання даних рушія, які надалі можуть бути використані у віртуальній машині. Перевага нативних функцій у порівнянні з функціями віртуальної машини полягає в тому, що вони виконуються процесором на відміну від віртуальної машини, що зменшує час виконання коду. Таким чином можна перенести важкий код з віртуальної машини в С++ та отримати приріст продуктивності.

Події необхідні для пересилання сповіщень як між об’єктами у віртуальній машині, так між подіями рушія та скриптами. На відміну від нативних функцій, такий підхід дозволяє сповіщати віртуальну машину коли саме треба виконати певний код, замість того, щоб перевіряти в тіку факт зміни стану. Кожен об’єкт або функція у віртуальній машині може підписатися на очікування певної події та відписатися коли немає в цьому потреби.

Потоки використовуються для того, щоб виконувати певний код з чітко визначеною періодичністю. Це необхідно для того, щоб не перевантажувати скриптовий код та перенести обрахування часу оновлення виклику функції у код віртуальної машини. Поток виконання підтримує виконання замикання та функції класу. Основна різниця між ними - неявний параметр this, який вказує на структуру з даними цього класу. Але так як віртуальна машина працює лише на одному потоці і не підтримує паралельність, тому потоки реалізовані як послідовне виконання функцій, які відповідають умові часу. Ці потоки не мають чіткого порядку виконання, проте за необхідністю можна реалізувати систему пріоритетності виконання потоків у віртуальній машині за допомогою певних тегів чи перечислення груп оновлення [13].

**4.3 Реалізація механізмів у скриптовій мові**

Реалізація нативних функцій є найпростішою в порівнянні з іншими механізмами. Нативна функція реєструється в спеціальному масиві на етапі ініціалізації віртуальної машини що дає можливість викликати її під час виконання коду. Для реєстрації нативної функції необхідно викликати функцію FEngineScriptVM::DefineNativeFunction, яка приймає такі аргументи: назва функції, яка буде доступна у віртуальній машині, та вказівник на цю С++ функцію. Нативна функція повинна задовольняти такі вимоги:

* 1. Функція повинна приймати 2 аргументи: масив значень та кількість аргументів
  2. Функція повинна повертати значення.

Такий підхід дозволяє реалізувати функції, які можуть отримати гнучке число аргументів від 0 і до 255 (функція має обмежений розмір стекового фрейму). І в результаті виконання функція може повернути максимум одне значення. Повернення кількох аргументів не підтримується скриптовою мовою.

Якщо існує функція, яка нічого не повертає в результаті виконання у віртуальну машину, то в такому випадку необхідно повернути Nil поле. Такий підхід використаний у віртуальній машині мови програмування Lua.

За необхідністю нативну функцію можна переписати так, щоб вона могла читати стековий фрейм функції та модифікувати його для можливсті прямого контролю над даними, проте такий підхід може збільшити кількість помилок програміста.

Для того, щоб реалізувати потоки чи події, необхідно додати механізм прив’язування віртуальних функцій та додаткових аргументів, який надалі можна буде передати в систему обробки потоків та подій. Аналогом такого рішення в мові С++ є std::function та std::bind [6]. Для цього визначимо нативну функцію Bind, яка буде повертати об’єкт типу FEventBindValue під час виконання віртуальної машини, так як деякі об’єкти неможливо захопити на етапі компіляції на відміну від С++. На рис. 29 зображено реалізація методу Bind.



Рис. 29. Реалізація методу Bind

Є кілька способів реалізувати систему подій. Перший спосіб полягає в тому, що всі об’єкти у віртуальній машині мають ієрархію дерева. В такій архітектурі є головний об’єкт у віртуальній машині, який зберігає посилання на всі інші об’єкти. В такій архітектурі можна відправити подію через алгоритм пошуку в глибину до елементу, який підписаний на цю подію. Проте така реалізація має певні недоліки такі як:

1. Необхідність реалізувати прибиральник сміття так, щоб він міг видаляти об’єкти за вимогою та зануляти всі посилання на цей об’єкт.
2. Пошук певної події у великій ієрархії кодової бази може зайняти велику кількість часу що впливає на загальну продуктивність віртуальної машини.

Другий спосіб полягає в тому, що всі об’єкти у віртуальній машині існують в глобальній пам’яті, стеку, функції, замиканні і мають свій термін життя. Відповідно програміст може підписати певну функцію на подію та відписати її за необхідністю. Надалі віртуальна машина викликає ці події за необхідністю з С++ коду. Мінус цього підходу в тому, що система подій буде тримати посилання на функцію (або метод класу) і тому цей об’єкт не буде видалений прибиральником сміття. Тому на програміста полягає відповідальність у ручному видаленні елементів, які підписуються на події так як скриптова мова не підтримує деструктори в класах, що унеможливлює реалізувати ідіому RAII.

Проаналізувавши переваги та недоліки цих способів, можна дійти до висновку, що саме другий спосіб є більш оптимальним, так як відсутні затримки пошуку об’єкту та необхідність розширювати прибиральник сміття, проте цей спосіб накладає вимогу ручного керування життя функцій обробників подій.

Поток виконання може бути створеним лише у віртуальній машини та використовуючи скриптові функції. Потоки виконання недоступні для взаємодії з нативного коду на відміну від подій. Метод класу автоматично передає this в цю функцію що дозволяє зберігати дані класу. Поток виконання може бути зупиненим видалення цього потоку з масиву потоків.

Для реалізації потоків та подій необхідно створити систему, яка реєструє події та викликає їх за необхідністю. Клас системи відповідає за збереження всіх функцій обробників, контролюванням їхнього життя та інформацією про події, які будуть оброблені під час наступного оновлення стану гри. Для збереження цих даних використовуються таблиці та масиви. На рис. 30 зображено клас системи подій та потоків.

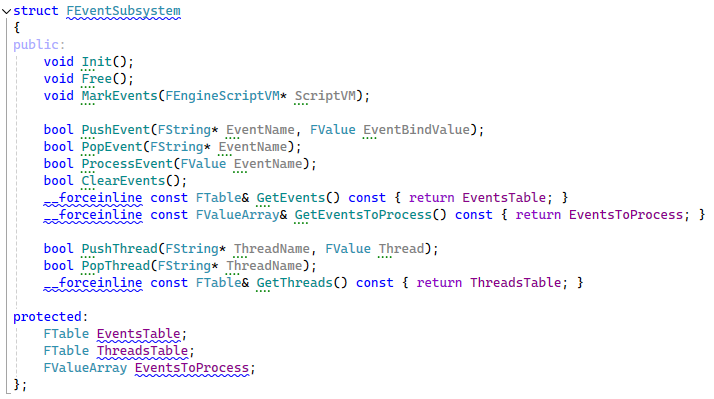


Рис. 30. Клас системи подій та потоків

Обробник подій та потоків знаходиться в функції оновлення віртуальної машини. Після виконання скриптової функції Tick, віртуальна машина проходить по масиву подій, шукає функції, які підписались на зареєстровані події та виконує їх по черзі. Потоки виконання виконуються лише тоді, коли співпадає умова за часом. Поток буде виконуватися до моменту, до поки не буде видалений з масиву потоків. На рис. 31 зображено функція оновлення віртуальної машини.



Рис. 31. Основна функція оновлення віртуальної машини

Наведемо приклад коду, який використовує ці підходи написання ігрового коду. В цьому прикладі створено два потоки виконання, які з періодичністю одна та три секунди виконують відповідні функції. Перший поток виконання викликає виконання події “SimpleEvent” коли другий виводить текст в консоль.

fun SimpleEvent(a, b)

{

print "Event Printed";

print a;

print b;

}

fun Main()

{

AddEvent("SimpleEvent", Bind(SimpleEvent, 1, 3.14));

CreateThread("FirstThreadName", Bind(CustomFirstThread), 3000);

CreateThread("SecondThreadName", Bind(CustomSecondThread), 1000);

}

fun CustomFirstThread()

{

ProcessEvent("SimpleEvent");

}

fun CustomSecondThread()

{

print "Call second thread";

}

[DEBUG] Calling thread SecondThreadName

Call second thread

[DEBUG] Calling thread SecondThreadName

Call second thread

[DEBUG] Calling thread SecondThreadName

Call second thread

[DEBUG] Calling thread FirstThreadName

[DEBUG] Calling event SimpleEvent

Event Printed 1 3.14

[DEBUG] Calling thread SecondThreadName

Call second thread

[DEBUG] Calling thread SecondThreadName

Call second thread

[DEBUG] Calling thread SecondThreadName

Call second thread

[DEBUG] Calling thread FirstThreadName

[DEBUG] Calling event SimpleEvent

Event Printed 1 3.14

**РОЗДІЛ 5. ІНТЕГРАЦІЯ**

**5.1 Інтеграція мови програмування з ігровим рушієм Unreal Engine 5**

Скриптова мова програмування була створена з метою простої інтеграції до рушія та запуску певного ігрового коду. Наведемо приклад інтеграції коду до ігрового рушія Unreal Engine 5. Загалом, цей рушій має свою скриптову мову програмування Blueprints, проте є найкращим варіантом для інтеграції коду так як має відкритий вихідний код та використовує мову програмування С++.

Враховуючи вище наведені обмеження до розробки, кодова база скриптової мови програмування має імплементацію необхідних класів для повноцінної роботи. Тому потрібно визначити підходи до інтеграції в рушій Unreal Engine 5. Під час інтеграції кодової бази в рушій необхідно врахувати такі особливості: взаємодія скриптової мови з кодом рушія, вивід тексту та помилок, робота з пам’яттю, підтримка «виконуваних зон».

Скриптова мова програмування створена з мінімізацією залежності від інших бібліотек, тому для перенесення коду в рушій, необхідно скопіювати файли реалізації скриптової мови до директорії проєкта гри та додати файли до конфігурації системи збірки проєкта (Visual Studio Project File, CMake, Premake, Sharpmake). У випадку рушія Unreal Engine створимо окрему динамічну бібліотеку, яка буде автоматично розпізнана програмою Unreal Build Tool та додана до конфігурації системи збірки. Для створення бібліотеки необхідно виконати такі кроки:

1. Створення папки з назвою скриптової мови в директорії {ProjectName}\Source\EngineScript та додавання файлів, що визначають цей модуль: EngineScript.h, EngineScript.cpp, EngineScript.Build.cs.
2. Копіювання всіх файлів скриптової мови програмування в цю директорію окрім Main.cpp бо рушій має свою точку початку виконання на відмінну від поточної реалізації.
3. Додання залежності проєкта гри від цього модуля. Зазвичай це можна зробити додавши інформацію про новий модуль проєкта в {ProjectName}\Source\{MainModule}\{MainModule}.Build.cs або в {ProjectName}\{ProjectName}.uproject.
4. Запустити генерацію файлів проєкта Visual Studio.
5. Перейменувати назву базового класу об’єкта FGCObject скриптової мови на інший з метою запобігання конфліктів з аналогічним класом рушія.
6. Додати визначений модулем рушія макрос ENGINESCRIPT\_API до структури віртуальної машини. Цей макрос підказує лінкувальнику С++, що саме цей клас та його функції треба відкрити для доступу з інших модулей (struct ENGINESCRIPT\_API FEngineScriptVM).

Наступним кроком є налаштування виводу тексту з метою логування роботи системи та помилок. Для виводу тексту в коді скриптової мови було створено дві функції, які були використані у всіх випадках де потрібно вивести текст: PrintOutputFormatted та PrintErrorFormatted. Тому для налаштування коректного виводу необхідно перенаправити вивід помилок до системи логування рушія. Для цього використаємо макрос UE\_LOG та створимо окрему категорію логування. Snprintf необхідно використати для форматування тексту, так як символи форматування, які були використані, не сумісні з тими, що підтримуються в макросі UE\_LOG. На рис. 32 зображено реалізацію цих функцій.

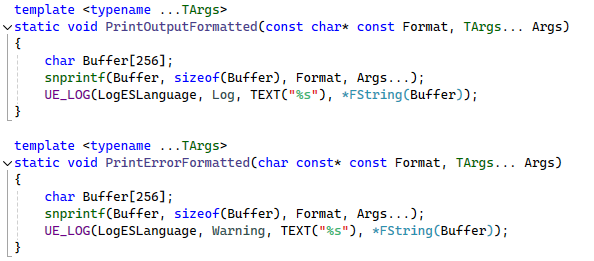


Рис. 32. Методи логування переписані для рушія Unreal Engine

Далі треба додати підтримку аллокаторів рушія Unreal Engine. Аналогічно з виводом тексту, створена функція, яка відповідає за весь процес видачі та видалення пам’яті. Ця функція використовує реалізований аллокатор блочного типу для роботи з пам’яттю. Одна з переваг рушія Unreal Engine в тому, що він має свою систему аллокації та деаллокації пам’яті, яку можна використати. Є кілька варіантів реалізації роботи з пам’яттю:

1. Перевизначити функції роботи з пам’яттю в аллокаторі скриптової мови на функції рушія Unreal Engine 5. Рушій має такі функції для стороннього коду: FMemory::Malloc, FMemory::Free.
2. Замінити код, який викликав функції аллокатора, на FMemory::Malloc, FMemory::Free. Реалізований клас аллокатора можна видалити так як він не буде використовуватися.

В результаті в цій інтеграції був вибраний перший спосіб, так як другий спосіб вимагає більш детального тестування і може нести ризики через те, що віртуальна машина скриптової мова має свої особливості роботи зі створеною пам’яттю.

Скриптова мова програмування спроєктована з урахуванням «виконуваних зон». Тобто кожна зона (ігровий світ) має певний набір файлів, які використовуються в цьому світі. Такий підхід дозволяє вручну керувати файлами, які необхідно завантажити у віртуальну машину для виконання, та ігнорувати ті, які не є потрібними для цього світу. Базова реалізація використовує текстовий файл, в якому прописані всі файли, які треба скомпілювати. Для збереження інформації про файли скриптової мови, які необхідно завантажити, треба створити клас, який наслідується від UDataAsset. На рис. 33 зображена імплементація цієї структури.

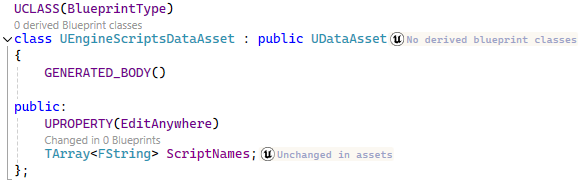


Рис. 33. Структура, яка визначає набір файлів скриптової мови.

Наступним кроком є ініціалізація та запуск віртуальної машини. Необхідно врахувати те, що віртуальна машина працює окремо від самого рушія, тому для неї необхідно створити окремий поток виконання, який не буде блокувати роботи основного ігрового потоку [18]. На рис. 34 зображений код, який запускає віртуальну машину в новому потоці рушія.

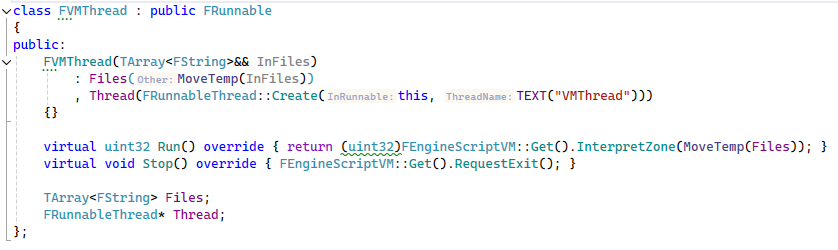


Рис. 34. Код створення нового потоку рушія.

Найкращим місцем для запуску віртуальної машини є клас, який наслідується від класу AGamemode. Цей клас відповідає за налаштування ігрових правил та створюється для конкретного ігрового світу. На рис. 35 зображений код, який завантажує виконувані файли та запускає новий поток виконання.



Рис. 35. Код ініціалізації віртуальної машини.

Отже, таким чином можна інтегрувати скриптову мову програмування у рушій Unreal Engine 5. Цього достатньо для повноцінної розробки гри, проте є певні аспекти, які необхідно виправити для коректної роботи віртуальної машини:

1. Синхронізація оновлення віртуальної машини з оновленням рушія. На даний момент, віртуальна машина оновлюється з періодичністю 60 раз на секунду але не синхронізована з часом оновлення самого рушія.
2. Додавання шляху до папки з скриптами в налаштування Unreal Build Tool з метою додавання її в фінальну збірку гри, так як рефлексія рушія не бачить залежності цих текстових файлів.

**5.2 Рекомендації щодо покращення скриптової мови програмування**

Реалізація має свої переваги та недоліки. Для подальшого використання необхідно перерахувати найбільші недоліки та окреслити можливі способи покращення цього коду для вирішення більш складних завдань, які стоять перед програмістами.

По-перше, необхідно реалізувати серіалізацію байт-коду, який видає компілятор. Поточна реалізація компілює код з кожним запуском клієнта гри, що є досить зручним для розробки коли програміст може швидко побачити зміни в коді, але цей підхід не є найкращим для фінальної збірки гри. Гравець не повинен очікувати компіляції коду кожен раз коли він відкриває гру чи завантажує новий рівень. Додаткові затримки під час запуску можуть погіршити ігровий досвід. Проте, за бажанням розробників, можна надати можливість гравцям модифікувати код та бачити їхні зміни в грі. Це спростить реалізацію та підтримку модів гри.

Для вирішення проблеми необхідно модифікувати інструментарій збірки гри, який зможе серіалізувати класи, які продукує компілятор, та зберегти їх в певні бінарні файли, які можна буде додати про релізної версії гри. Надалі клієнт гри повинен зчитати ці бінарні файли та десеріалізувати значення в щойно створені класи компілятора. Для коректної роботи серіалізації необхідно зберегти всю інформацію, яка знаходиться в чанках байт-коду, якими володіють функції та класи, глобальні імена функцій, змінних, класів та його нащадків. Таким чином необхідність додавання коду скриптової мови в проєкт та її компіляція під час запуску зникає.

По-друге, необхідно зробити оптимізації використання пам’яті класами віртуальної машини та компілятора. Для кращої роботи з пам’яттюта мінімізацією затримок під час виділення нової пам’ятів поточній реалізації використаний клас аллокатора. Проте реалізація цього аллокатору не є найкращою в порівнянні з іншими видами. Поточна імплементація базується на аллокаторах блочному типу. Цей тип побудований на ідеї резервування великого блоку пам’ятіта видачі пам’яті блоками з чітким розмірами кожного з них. Такий підхід має перевагу в тому, що розмір кожного блоку задовольняє всі необхідні потреби скриптової мови програмування, проте це створює такі недоліки:

1. Скриптова мова програмування має два типи об’єктів**:** класи віртуальної машини та масиви даних, кожні з яких мають свої потреби в пам’яті. Об’єктивимагають 128 байт на кожну копію класу, а масиви даних вимагають до 8Кб. Для вирішення цієї проблеми створені два аллокатори з різними розмірами блоків.
2. Друга проблема витікає з першої – немає ніяких гарантій, що певний об’єкт буде використовувати всю доступну пам’ятьцього блока. Наприклад, текстовий рядок “Hello World!” у віртуальній машині буде займати 12 байт, але блок для рядка займає 8Кб.
3. Третя проблема полягає в тому, що фіксовані блоки можуть мати велику кількість вільної пам’яті**,** яка не може бути використана бо зафіксована за певним блоком. А якщо пам’яті не вистачає, то необхідно збільшувати розміри блоків.

Враховуючи вище сказані проблеми необхідно реалізувати такий підхід до керування пам’яттю**,** який дозволить мінімізувати проблеми блочного аллокатора, але й не створювати високу фрагментацію даних.

Основна проблема фрагментації данихполягає в тому, що аллокатор може містити велику кількість вільної пам’ятіпроте вона буде розкидана по різним місцям і її може бути недостатньо для виділення послідовному блоку даних певного розміру під новий об’єкт. Тому для вирішення цих проблем необхідно реалізувати аллокатор, який буде відповідати таким вимогам:

1. Аллокатор повинен підтримувати виділення пам’яті блоками з різними розмірами.
2. Аллокатор повинен підтримувати можливість розширення пім’яті.
3. Аллокатор повинен мати певні способи боротьби з фрагментацією. Тобто, аллокатор повинен періодично шукати блоки, які можна додати в одне ціле та використовувати їх для нового виділення даних. Але він не може переносити дані з одного блоку в інший, бо це призведе до того, що віртуальна машина не буде знати про нову адресу об’єкту.

Для реалізації такого аллокатору можна використати такий підхід – клас зберігає зв’язний список блоків великого розміру, які він може видаляти або додавати залежно від того, як використовується пам’ять. В кожному блоці з фіксованим розміром аллокатор виділяє необхідну кількість байтів під об’єкт чи масив та мінімізує можливу фрагментацію.

По-третє, якщо розробники не планують робити підтримку модифікацій коду гравцем або спростити код, який попадає у фінальний білд, вони можуть створити певні макроси, які будуть на етапі препроцесінгу видаляти код сканера та компілятора.

По-четверте, розробники можуть додати підтримку геш-сум для рядків. Геш-суми дозволяють виконувати операції порівняння рядків швидше (за O(1)) ніж послідовне порівняння тексту (за O(n)). Також це дозволить мінімізувати кількість рядків, які знаходяться в пам’яті відкрито, бо деякі люди можуть досліджувати пам’ять з метою пошуку певних підказок нового контенту (дата-майнери), написання читів тощо.

**ВИСНОВКИ**

1. Опрацьовано інформаційні джерела з теми магістерської роботи: проаналізовано різноманітні скриптові мови програмування, досліджено особливості застосування формальної граматики під час компіляції токенів та вимоги до скриптових мов програмування.

2. Визначена специфіка скриптових мов програмування, які застосовуються в різних ігрових рушіях та методи взаємодії такої мови програмування з зовнішнім середовищем або нативним кодом, в якому виконується скриптова мова.

3. Опрацьовано ряд вимог, які потрібні для того, щоб скриптова мова програмування могла виконувати необхідні потреби, які ставить ігровий рушій та логіка самої гри.

4. Спроектовано та реалізовано необхідний код систем для базової реалізації скриптової мови програмування, а саме сканер символів, комплілятор та віртуальна машина, яка виконує результати компіляції коду.

5. Додано підтримку подій та потоків на рівні віртуальної машини, які є необхідними для написання асинхронного коду скриптів рушія.

6. Інтегровано код скриптової мови програмування до рушія Unreal Engine 5, що дозволяє визначити конкретні кроки, які необхідні для того, щоб перенести скриптову мову програмування до будь-якого іншого рушія.

7. Окреслено можливі способи покращення кодової бази скриптової мови програмування, а саме: додавання серіалізації для байт коду, що дозволить не компілювати скриптовий код під час запуску ігри, покращення роботи з пам’яттю, а саме заміна аллокатора на більш гнучку модифікацію, яка зможе мінімізувати фрагментацію пам’яті, видалення класів, які відповідають за компіляцію коду у фінальній версії гри, для того, щоб мінімізувати ймовірність зміни коду гравцями, та підтримку геш-сум для текстових рядків.

**СПИСОК ВИКОРИСТАНИХ ІНФОРМАЦІЙНИХ ДЖЕРЕЛ**

1. Порядок виконаня команд : веб-сайт. URL: <https://studfile.net/preview/3907471/page:35/> (дата звернення: 27.09.2024)
2. Closures : веб-сайт. URL: <https://www.lua.org/pil/6.1.html> (дата звернення: 27.09.2024)
3. A. Rigo and S. Pedroni. PyPy’s approach to virtual machine construction. In P. L. Tarr and W. R. Cook, editors, Companion to the 21th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, 2006. P. 944–953.
4. Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied. Reading, MA: Addison-Wesley, 2001.
5. Bilingual Problems: Studying the Security Risks Incurred by Native Extensions in Scripting Languages. Cristian-Alexandru Staicu, CISPA Helmholtz Center for Information Security; Sazzadur Rahaman, University of Arizona; Ágnes Kiss and Michael Backes, CISPA Helmholtz Center for Information Security ISBN - 978-1-939133-37-3
6. Bjarne Stroustrup. The C++ Programming Language (Fourth Edition). Reading, MA: Addison-Wesley, 2013.
7. C++ compilation steps : веб-сайт. URL: <https://www.linkedin.com/pulse/c-compilation-steps-amit-nadiger> (дата звернення: 27.09.2024)
8. Calling C++ Functions From Lua : веб-сайт. URL: <http://gamedevgeek.com/tutorials/calling-c-functions-from-lua/> (дата звернення: 27.09.2024)
9. David H. Eberly. Game Physics. San Francisco, CA: Morgan Kaufmann, 2003.
10. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., de Moura, L.: A metaprogramming framework for formal verification. Proc. ACM Program. Lang. 1(ICFP) (Sep 2017).
11. Fraser Brown, Andres Nötzli, and Dawson R. Engler. How to build static checking systems using orders of magnitude less code. In International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS), 2016.
12. Function Specifiers : веб-сайт. URL: <https://dev.epicgames.com/documentation/en-us/unreal-engine/ufunctions-in-unreal-engine> (дата звернення: 27.09.2024)
13. Jason Gregory - Game Engine Architecture 3rd Edition. ISBN - 978-1138035454
14. John C. Reynolds. Theories of Programming Languages. Cambridge University Press, Cambridge, England, 1998. Iv
15. List of game engines : веб-сайт. URL: <https://en.wikipedia.org/wiki/List_of_game_engines> (дата звернення: 27.09.2024)
16. Mark and Sweep garbage collection algorithm : веб-сайт. URL: <https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/> (дата звернення: 27.09.2024)
17. Minimal Example of Calling Lua Functions from C++ : веб-сайт. URL: <https://gouthamanbalaraman.com/blog/minimal-example-lua-function-cpp.html> (дата звернення: 27.09.2024)
18. Multithreading and Performance in Unreal : веб-сайт. URL: <https://forums.unrealengine.com/t/multithreading-and-performance-in-unreal/1216417> (дата звернення: 27.09.2024)
19. Roberto Ierusalimschy. Programming in Lua, Fourth Edition. Lua.org, 2016.
20. The Java Virtual Machine Specification. [Online]. Available: веб-сайт. URL: <https://docs.oracle.com/javase/specs/jvms/se8/html/index.html> (дата звернення: 27.09.2024)
21. Top Down Operator Precedence by Vaughan R. Pratt : веб-сайт. URL: <https://tdop.github.io/> (дата звернення: 27.09.2024)
22. Tracing garbage collection : веб-сайт. URL: <https://en.wikipedia.org/wiki/Tracing_garbage_collection> (дата звернення: 27.09.2024)
23. Unreal Engine 5 : веб-сайт. URL: <https://www.unrealengine.com/en-US/> (дата звернення: 27.09.2024)
24. Xavier Leroy. Java bytecode verification: algorithms and formalizations. Journal of Automated Reasoning, 2003, 30 (3-4), pp.235–269.
25. Циганков А. Виконання байт-коду в інтерпретованій мові програмування. *Вісник студентського наукового товариства* [електронне видання]: збірник наукових праць студентів, магістрантів і аспірантів / за заг. ред. О. В. Мельничука. Ніжин: НДУ ім. М. Гоголя, 2024. Вип. 30. с. 56–58.
26. Циганков А. Застосування байт-коду при розробці мови програмування. *Вісник студентського наукового товариства* [електронне видання]: збірник наукових праць студентів, магістрантів і аспірантів / за заг. ред. О. В. Мельничука. Ніжин: НДУ ім. М. Гоголя, 2023. Вип. 29. с. 41–46.
27. Циганков А. Застосування формальної граматики для написання інтерпретатора. *Вісник студентського наукового товариства* [електронне видання]: збірник наукових праць студентів, магістрантів і аспірантів / за заг. ред. О. В. Мельничука. Ніжин: НДУ ім. М. Гоголя, 2024. Вип. 31 (подано до друку).