МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

Факультет Магістерської підготовки
Кафедра інформаційних технологій

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Технологія створення ігрового двигуна мовою Java»

Виконав студент 2 курсу групи МІС-18
спеціальності 122 Комп'ютерні науки

Каленчук Іван Сергійович

Керівник д.х.н., проф.
Кругляк Юрій Олексійович

Консультант

Рецензент д.т.н., проф.
Мещеряков Володимир Іванович

Одеса 2019

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

БД	– База Даних
ОС	– англ. Operating system, OS
DAW	– Digital Audio Workstation
EJLF	– Executable-Jar-Library-File
FLS	– Fruity Loops Studio
inLan	– англ. Initialization Language
iOS	– iPhone Operation System
Jar-файл	– англ. Java ARchive
JSON	– JavaScript Object Notation
MIDI	– Musical Instrument Digital Interface
NPC	– Non-Player Character
OpenGL	– Open Graphics Library
PC	– Personal Computer
RPG	– Role-Playing game
SDK	– Software development kit
SvSAteunane	– сінір. Serpye Varante Scarynte
UML	– Unified Modeling Language
WAV	– Waveform
WXGA	– Wide Extended Graphics Array
XML	– англ. Extensible Markup Language

Терміни

Геймпад – (англ. Gamepad, джойпад – англ. Joypad) – тип ігрового маніпулятора. Являє собою пульт, який утримується двома руками, для контролю його органів управління використовуються великі пальці рук

Геймплей – компонент ігри, який відповідає за інтерактивну взаємодію гри та гравця

Інді-гра (англ. Indie game, від англ. Independent video game – «незалежна комп'ютерна гра») – комп'ютерна гра, створена окремим розробником або невеликим колективом без фінансової підтримки видавця комп'ютерних ігор

Інтерпретатор – програма, що виконує інтерпретацію порядковий аналіз, обробку та виконання вихідного коду програми або запиту

Клас-ініціалізатор – клас ігрового двигуна SvSAteunane, що призначений для ініціалізації всіх ігрових об’єктів рівня

Кодогенерація – це процес генерації коду на основі певних даних

Мод (моддінг ігор, прог. жарг. «мод») – додаток до комп'ютерної гри

Нативна мова – система команд (мова) конкретної обчислювальної машини (машинний мову), який інтерпретується безпосередньо мікропроцесором або мікропрограмами даної обчислювальної машини

Нативний виконувальний файл – виконувальний файл даної ОС

Партитура (італ. Partitura) в музиці – нотний запис багатоголосого музичного твору, призначеного для виконання ансамблем, хором або оркестром, в якій всі партії (голоси) одна над іншою дані в певному порядку

Піксель – найменший логічний елемент двовимірного цифрового зображення в растровій графіці, або фізичний елемент матриці дисплеїв, які формують зображення

Плагін – незалежно компільований програмний модуль, що динамічно підключається до основної програми та призначений для розширення

Платформер – жанр комп'ютерних ігор, в яких основною рисою ігрового процесу є стрибання по платформах

Рефлексія – (від лат. Reflexio – звернення назад) – це механізм дослідження даних про програму під час її виконання. Рефлексія дозволяє досліджувати інформацію про полів, методів та конструкторів класів

C++ – компільована, статично типізована мова програмування загального призначення

Секвенсор (англ. Sequence – «послідовність») – апаратний пристрій або прикладна програма для запису, редагування та відтворення «послідовності MIDI-даних»

Спрайт – графічний об’єкт у комп’ютерній графіці

Стелс (англ. Stealth – «невидимка, прихованість») – жанр комп'ютерних ігор, в яких гравець повинен уникати виявлення ігрового персонажа противниками або приховано усувати їх, не привертаючи до себе уваги

Текстура – зображення, що відтворює візуальні властивості будь-яких поверхонь або об'єктів

Шутер (англ. Shooter – «стрілець») – жанр комп'ютерних ігор, де геймплей зосереджений на стрілянні зі зброї

ArgoUML – засіб UML моделювання. Є відкритим програмним забезпеченням та розповсюджується під ліцензією EPL

BitsLib – бібліотека команди SvS, що призначена для ініціалізації маскових констант

C – типізована мова програмування

Hitbox – метод виявлення взаємодій з ігровим об'єктом складної форми через перевірку простіших за формою області навколо об'єкта

HotSpot – це основна віртуальна машина Java (JVM) як для клієнтських, так і для серверних комп'ютерів, що випускається корпорацією «Oracle»

InteliSense – технологія автодоповнення Microsoft, найбільш відома в Microsoft Visual Studio. Дописує назву функції при введенні початкових літер. Крім прямого призначення, IntelliSense використовується для доступу до документації та для усунення неоднозначності в іменах змінних, функцій і методів, використовуючи рефлексію

IntelliJ IDEA – інтегроване середовище розробки програмного забезпечення для багатьох мов програмування, зокрема Java, JavaScript, Python, розроблена компанією JetBrains

JavaFX – платформа на основі Java для створення додатків з насиченим графічним інтерфейсом

LibGdx – фреймворк для створення ігор та програм, написаний на Java з використанням C та C++ (для швидшої роботи)

OracleDB – об'єктно-реляційна система управління базами даних

PlayStation (яп. プレイステーション *Пурэйсутэ : сён*, офіційне скорочення PS) – ігрова приставка п'ятого покоління, розроблена компанією Sony Computer Entertainment

Roguelike – жанр комп'ютерних ігор, зокрема, піджанр комп'ютерних рольових ігор

Shrife – бібліотека, що розроблена командою SvS для набору алгоритмів шифрування даних, файлів тощо

Singleton – породжує шаблон проектування, що гарантує, що у програмі буде єдиний екземпляр деякого класу, та надає глобальну точку доступу до цього екземпляру

StarUML – це проект з відкритим кодом для розробки швидкої, гнучкої, розширюваної, функціональної та вільно доступної платформи UML

StringBuffer – клас, створений на Java для оптимізації роботи з рядком, що часто змінюється

SvSList – бібліотека, що представляє собою заміну LinkedList

Visual Paradigm – це інструмент UML CASE, що підтримує UML 2, SysML та позначення моделювання бізнес-процесів (BPMN) з групами управління об'єктами

Visual Studio – лінія продуктів компанії Microsoft, що включають інтегроване середовище розробки програмного забезпечення та ряд інших інструментальних засобів

Xbox – ігрова приставка

ВСТУП

Ігровий продукт – (або комп’ютерна гра) комп’ютерна програма, що використовується для організації ігрового процесу (геймплею), зв’язку з партнерами по грі, або ж сама виступає у ролі партнера.

На сьогодні, замість терміну комп’ютерна гра може використовуватися термін відеогра, тобто ці терміни можуть бути взаємозамінними. Відеогра, на даний момент, являє собою один з найскладніших, найпотрібніших, масштабніших проектів, що включає в себе багато найрізноманітніших направлень, зокрема програмування графічного / фізико-математичного ядра, графічно-модельний / музикально-звуковий контент, програмне проектування, ідейна розробка, Web-design та інше.

Комп’ютерні ігри являють собою одною з драматичних форм, а їх інтерактивність – це питання рівня участі, але не форми. Саме через це, як і інші форми, відеогра має п’ять ключових елементів: стиль, фабула, герой, декорації та тема. Усі гарні ігри повинні володіти деяким розважальним потенціалом, та в більшості випадків їх він базується на класичних законах драми.

На перший погляд може здатися, що відеогра є негативним комп’ютерним контентом, але це не зовсім так. Сюжетні ігри, де увага дитини зосереджена на повісті, покращують властивість до концентрації та виробляють звичку до читання. Логічні ігри покращують мислення та пам’ять у літніх людей. Також є фактом, що бельгійські лікарі допомагають пацієнтам з агорафобією та акрофобією подолати страхи за допомогою шолома віртуальної реальності.

Не мало комп’ютерних ігор мають здатність розширятись за допомогою модулів, так званих модів, зокрема на нативній мові програмування (тобто надають вихідний код), що являють собою деяку мотивацію вчити мови програмування. Є доволі не мало відеоігор, що вважаються розслаблюючими, тобто їх контент, динаміка, та, що мабуть найголовніше – музичне супроводження, так підібрані, що людина після тяжкого дня, приїхавши, напри-

клад, з роботи, може завдяки такій грі відкинути всі непотрібні в даний час думки та просто відпочити. Такі ігри в деякому розумінні захищають психіку людини, як би це не дивно звучало.

Гра – це віртуальна реальність, яка дає гравцеві дивовижну свободу дій та такі можливості, яких у них немає та ніколи не буде в реальному житті. Тут можна провести паралелі з різними філософіями: наприклад, реально тільки те, про що мислить людина і, якщо він мислить про гру (грає) і занурений у віртуальну реальність, значить, вона насправді реальна та немає ніякої різниці між реальним світом й вигаданим. Більш того, емоції, які гравець відчуває під час гри, часто набагато вище гостріше, ніж в реальному світі. Чи помічали ви, що віртуальному ігровому світі час тече зовсім інакше? Сідаючи на п'ять хвилин, ймовірно, ви проживаєте ціле життя. І навпаки – здавалося б ви грали лише п'ять хвилин, а пройшло вже кілька годин. І весь цей час людина рухається до поставленої мети, будь то проходження квесту, рішення головоломки, отримання максимальних очок, перемогу над противником. І це дуже важливо.

Тож, кожна гра – це симулятор отримання щастя. Припускаю, що більше за всіх грають люди, у яких рух на щастя в реальному житті чимось загальмовано, а також ті, у кого немає чіткої мети в житті. Перший випадок – наприклад, довгоочікуваний реліз все ніяк не може відбутися, розробка тягнеться й тягнеться. Мета все ніяк не досягається і постійно відкладається, час йде повільно. Хтось поставить собі альтернативні, тобто додаткові цілі, а хтось піде з головою в якусь гру. Другий випадок – це коли людині незрозуміло чим зайнятися в житті. Чому грають, в основному, діти і підлітки? Тому, що вони ще тільки визначаються з тим, що ж саме зробить їх щасливими, вибирають свій шлях. Чи багато ви знаєте процвітаючих бізнесменів або підприємців, які грають в ігри? Сумніваюся. У них просто немає на це часу, тому що їхні цілі знаходяться в реальному світі і приносять цілком реальне щастя при їх досягненні.

Відеоігри також можна ще класифікувати на двовимірні та тривимірні. І хоч може здатись, що двовимірна гра гірша, але далеко не завжди тривимірна гра може передати те відчуття, що присутнє у грі на вимір меншої. Двовимірна гра безперечно буде менш важкою у плані комп'ютерної продуктивності, але на сьогодні, враховуючи рівень сучасного, так би мовити, «заліза» можна на це не звертати увагу. Але навіть так, двовимірні ігри не зникли як чергова ступінь розвитку, вони й по сьогодні активно створюються, адже двовимірна гра разом з мінімалістичною графікою уособлює собою класику ігрової індустрії, з цього все почалось, та на мою думку такий вид гри навряд чи зникне, та завжди буде актуальним.

Даний ігровий продукт є комп'ютерною двовимірною інді-грою у жанрі двовимірної пісочниці з елементами survival horror та rougelike .

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Характеристика об’єкта розробки

Як було сказано, відеогра являє собою досить великий проект, що поєднує в собі багато різних напрямків. Та хоча продукт, як і інші, є односпрямованим, це ПО має набагато більше шансів зацікавити людей різноманітнішого світогляду. Саме це є причиною вибору теми дипломного проекту з усіх можливих програмних продуктів саме комп’ютерну гру.

Для того, щоб гра могла доволі довго цікавити людину вона має бути або довгою у плані проходження, або ж якось сама могла створювати різні рівні. Для даного проекту було вигідніше обрати другий варіант, так, оскільки проект розроблюється лише двома людьми, і за такі терміни не було просто часу на створення рівнів. Плюсом є ще те, що гра, так би мовити, сама себе обновлює, адже гарно спроектована система генерації сама постійно створює новий рівень. Завдяки цьому гра має додатковий жанр roguelike.

1.2 Опис предметної області

Хоча для створення інді-проекту існує велика кількість безкоштовних у вільному доступі різноманітних бібліотек, ігрових двигунів та інше, для даного проекту було збудовано новий власний двигун SvSAteunane та власні бібліотеки для раціоналізації та його функціонування.

Також для захисту jar-файлу від перегляду вихідного коду була розроблена система захисту EJLF. Був розроблений сервер, БД та система реєстрації, що зобов’язує гравця обов’язково зареєструватися. Спеціально для проекту було створено у програмі FLS музикальний контент. Також було створено унікальний геймплей. Ще було створено ряд міні-програм, що допомагали обробляти арт, наприклад, створити атлас спрайтів, створити кадрі анімації з поворотом зображення або ж з візуальним переходом, та інше.

1.3 Аналіз існуючих аналогів

Існує не мала кількість ігрових двигунів. Кожен з них має свої особливості. Для наочності було розглянуто три цілком відомі ігрові двигуни.

Першим аналогом було розглянуто ігровий двигун GameMaker (рис. 1.1). Він написаний на Delphi. Доступний для ОС Windows, 7-а версія програми також існувала в версії для Mac. Система розрахована в основному на створення двомірних (2D) ігор будь-яких жанрів. Також підійде для створення різних презентацій і подібного. Починаючи з 6-ї версії з'явилася обмежена можливість працювати з 3D.

Створення гри в Game Maker не вимагає попереднього знайомства з будь-яким з мов програмування. Інтерфейс об'єднує в собі редактори спрайтів, об'єктів, кімнат, скриптів, а також тайм-лайнів (послідовностей дій з прив'язкою за часом), шляхів (маршрутів) руху та констант. Гра в Game Maker будується як набір ігрових об'єктів. Поняття об'єкта в основному відповідає поняттю класу в об'єктно-орієнтованому програмуванні, об'єкти можуть успадковуватися один від одного. Примірники об'єктів можуть бути розміщені в ігровому просторі за допомогою редактора кімнат, або ж створені динамічно [1].

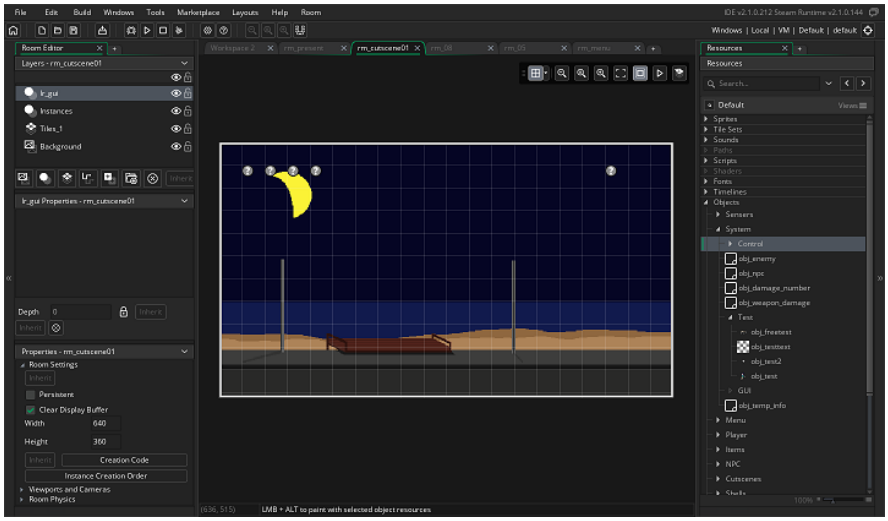

Рисунок 1.1 – Робоче середовище GameMaker

Переваги:

- графічний редактор;
- малі по об’єму ігри доволі швидко працюють;
- актуальність.

Недоліки:

- несумісність з Windows 10;
- для своїх задач він дуже тяжкий у плані ресурсів при розробці;
- не зовсім зручний інтерфейс.

Наступним двигуном виступає Unity. Мультиплатформенне середовище розробки комп'ютерних ігор (рис. 1.2). Дозволяє створювати додатки, що працюють під більш ніж 20 різними операційними системами, що включають персональні комп'ютери, ігрові консолі, мобільні пристрої, інтернет-додатки та інші [2].

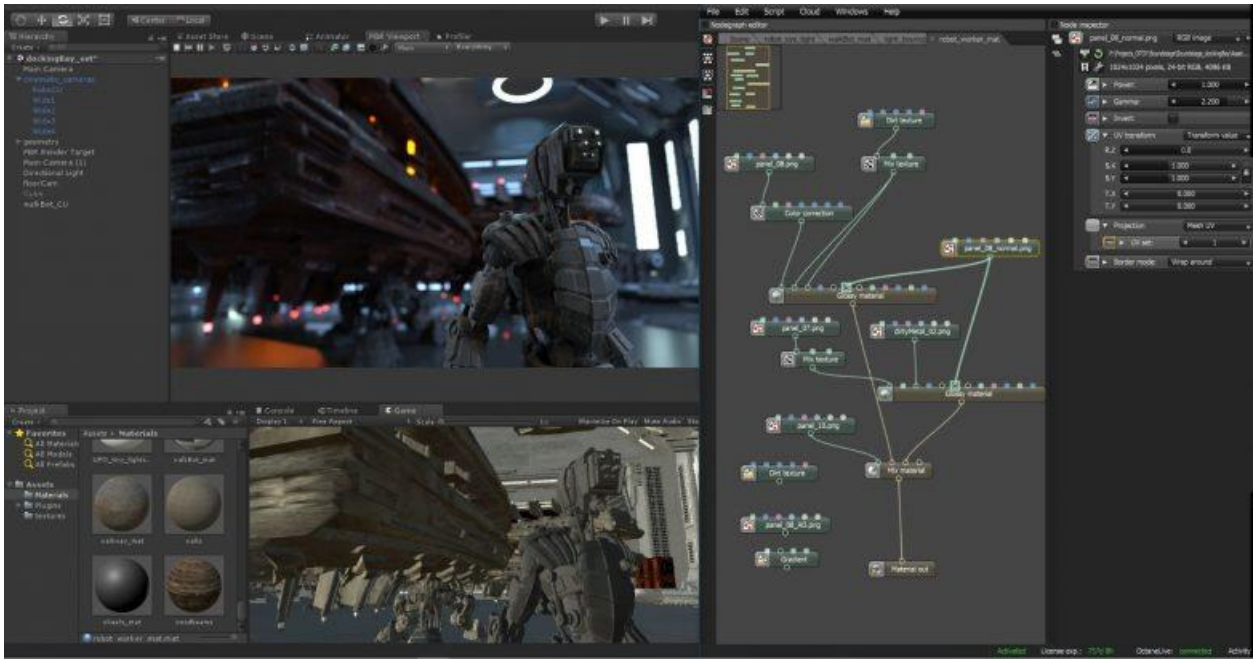

Рисунок 1.2 – Робоче середовище Unity

Переваги:

- зручний інтерфейс;
- великий об’єм плагінів, що значно прискорює розробку гри;
- підтримка великого набору API, платформ та технологій.

Недоліки:

- повільність великих ігор;
- ігри потребують багато ресурсів;
- навіть малі ігри займають багато місця на диску.

Останнім аналогом виступає UnrealEngine (рис. 1.3). Хоча двигун спочатку був призначений для розробки шутерів від першої особи, його наступні версії успішно застосовувалися в іграх самих різних жанрів, в тому числі стелс-іграх, файтингах та масових багатокористувацьких рольових онлайн-іграх [3].

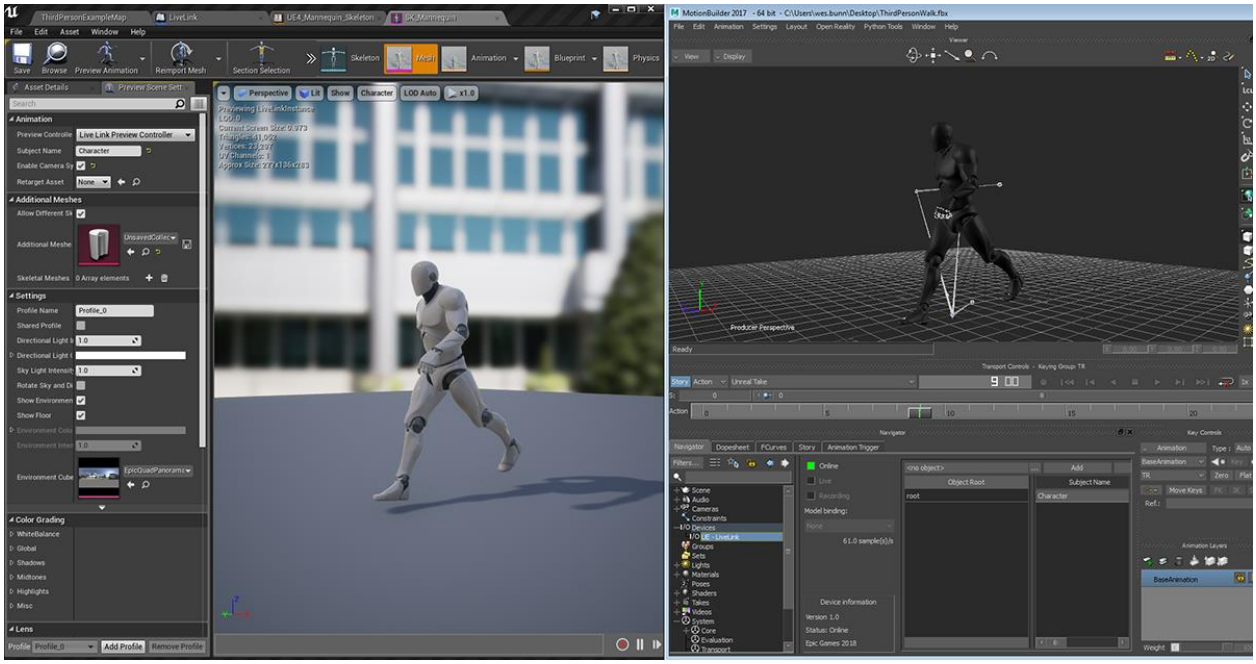

Рисунок 1.3 – Робоче середовище UnealEngine4

Переваги:

- ідеальний для написання фундаментальних речей;
- зручне візуальне програмування;
- безкоштовний.

Недоліки:

- занадто не повороткий для створення ігрових механік;
- не підходить для мобільних додатків;
- складність реалізації малих ігор.

Після проведеного аналізу подібних інформаційних систем була поставлена задача: спроектувати та розробити відеогру, що буде унікально захищена, потребувати реєстрацію гравця. Також наявність генерації рівнів та фізичного ядра. Повинно бути створено ексклюзивний музичний / графічний арт. Основна причина створення власного двигуна це нераціональність сучасного підходу до архітектори проекту, що впливає на продуктивність. До того ж, сучасні двигуни націлені на простоту інтерфейсу, що не дає такої гнучкості та раціоналізації як програмний код. Звідси й проблеми з продуктивністю. Також власний двигун є демонстрацією навичок програмної розробки.

1.4 Аналіз засобів розробки

Для реалізації відеогри на PC та Андроїд були використані мови C та Java, інтерфейс LibGdx для роботи з графічним процесором через OpenGL, та JavaFX для реалізації вікна реєстрації.

IntelliJ IDEA середовище розробки, що було обрано для розробки дипломного проекту. Переваг у цієї програми доволі багато, але важливо те, що саме це середовище, незважаючи на те, що воно здатне підтримати багато мов програмування, зосереджено саме під розробку на мові Java. Дуже вигід-

ним є доволі гнучке налаштування стилізації програмного тексту. Інші середовища не мають настільки зручного налаштування.

C – здатна до компіляції статично типізована мова програмування загального призначення, що була розроблена в 1969-1973 роках співробітником Bell Labs Денніс Рітчі як розвиток мови Бі. Спочатку була розроблена для реалізації операційної системи UNIX, але згодом була перенесена на безліч інших платформ.

Мова Сі унікальна з тієї точки зору, що саме вона стала першою мовою високого рівня, всерйоз витіснивша асемблер в розробці системного програмного забезпечення. Вона залишається мовою, що реалізована на максимальній кількості апаратних платформ, і одна з найпопулярніших мов програмування, особливо в світі вільного програмного забезпечення [4].

Java – сильно типізована об’єктно-орієнтована мова програмування, розроблена компанією Sun Microsystems (в подальшому вона була придбана компанією Oracle). Програми Java зазвичай транслюються в спеціальний байт-код, тому вони можуть працювати на будь-якій комп'ютерній архітектурі, за допомогою віртуальної Java-машини [4].

Байт-код виконується віртуальною машиною Java (JVM) – програмою, що обробляє байтовий код та передає інструкції обладнанню як інтерпретатор. Перевагою подібного способу виконання програм є повна незалежність байт-коду від операційної системи та устаткування, що дозволяє виконувати Java-програми на будь-якому пристрої, для якого існує відповідна віртуальна машина. Іншою важливою особливістю технології Java є гнучка система безпеки, в рамках якої виконання програми повністю контролюється віртуальною машиною.

Будь-які операції, які перевищують встановлені повноваження програми (наприклад, спроба несанкціонованого доступу до даних або з'єднання з іншим комп'ютером), викликають негайне переривання. Програми, написані на Java, мають репутацію більш повільних та займають більше оперативної пам'яті, ніж написані на мові C. Проте, швидкість виконання програм, напи-

саних на мові Java, була істотно поліпшена з випуском в 1997-1998 роках так званого JIT-компілятора в версії 1.1 на додаток до інших особливостей мови для підтримки кращого аналізу коду (такі, як внутрішні класи, клас String-Buffer, спрощені логічні обчислення і т. д.). Крім того, була проведена оптимізація віртуальної машини Java – з 2000 року для цього використовується віртуальна машина HotSpot. Станом на лютий 2012 року, код Java 7 приблизно в 1.8 рази повільніше коду, написаного на мові Сі [5].

LibGdx – фреймворк для створення відеоігор та іншого роду програм, написаний на Java з використанням C/C++ (для більш швидкої роботи). Він дозволяє писати кросплатформенні ігри та програми використовуючи один код (рис. 1.4).

LibGDX дозволяє розробнику створювати, тестувати та налагоджувати код на власному комп'ютері, і потім переносити його на інші ОС. Для цього фреймворк використовує окремі модулі для збірки додатку під кожну платформу, а також незалежний модуль, який містить основний код програми. Гнучкість та збільшення можливостей досягається за рахунок розширень. Можна підключити фізичний двигун для роботи з об’єктами та фізикою реального світу, додати підтримку шрифтів або працювати з 3D об’єктами [6].

Рисунок 1.4 – LibGdx

SvSAteunane – ігровий двигун, що був розроблений з використанням бібліотеки LibGdx для створення ігрових продуктів, використовуючих дво-

вимірну графіку (рис. 1.5). Для роботи самого двигуна було розроблено бібліотеки, зокрема SvSList, Imager, BitsLib, Shrife та інші [7].

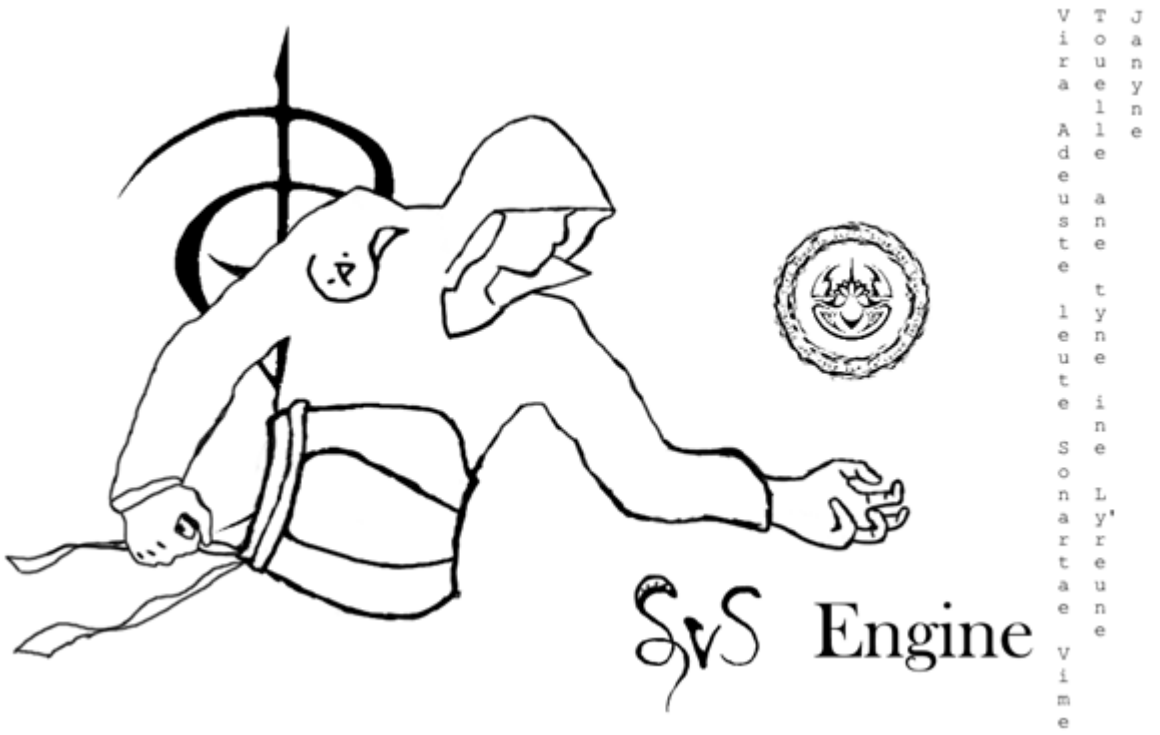

Рисунок 1.5 – SvSAteunane

FLS – цифрова звукова робоча станція (DAW) для написання музики (рис. 1.6). Музика створюється шляхом запису та зведення аудіо або MIDI-матеріалу. Готова композиція може бути записана в форматі WAV, MP3 або OGG. FL Studio є доріжковим (патерн) секвенсором, де створення музики відбувається в Piano Roll, Step Sequencer та потім здійснюється складання всієї композиції з окремих частин у вікні Playlist. Є великий набір вже готових інструментів та безліч ефектів, що можуть бути задіяні в режимі реального часу.

Головна складова проекту композиції є генератор (канал). Генератор синтезує або відтворює звук. Генераторів в проекті композиції може бути необмежена кількість. Кожен генератор володіє своїми налаштуваннями та унікальним звуком, що імітує будь-який інструмент. Для генераторів програму-

ються нотні партитури, що записуються в Piano Roll (рис. 1.7). Партитури в FL Studio мають нескінченну довжину.

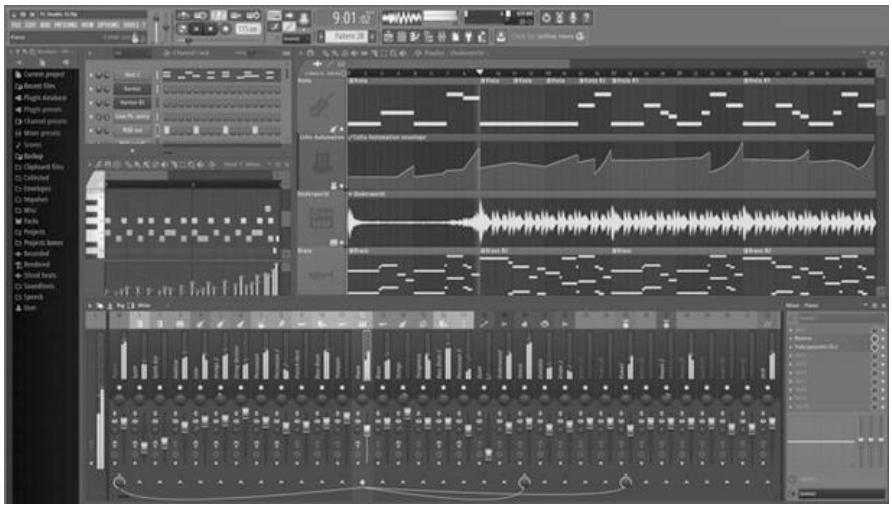

Рисунок 1.6 – Робоче середовище «FL Studio 12»

Шматочки партитур (патерни) складаються в послідовності (розташовуються в потрібному порядку в списку відтворення) у вікні Playlist. Звук кожного генератора може бути оброблений за допомогою безлічі ефектів, що підключаються [8].

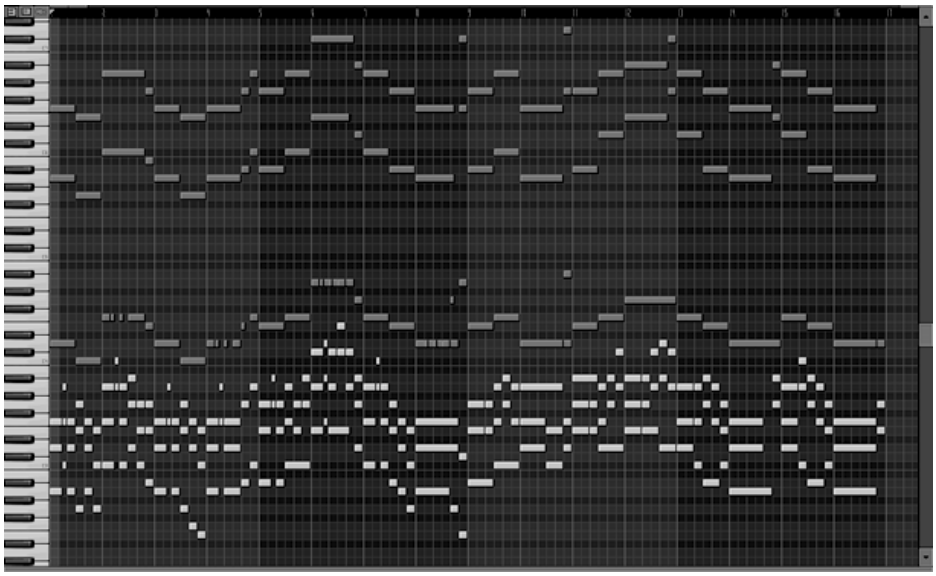

Рисунок 1.7 – Середовище «Piano Roll»

2 ТЕХНОЛОГІЯ СТВОРЕННЯ ПРОГРАМНОГО ПРОДУКТУ

2.1 Основні елементи системи

UML (англ. Unified Modeling Language – уніфікована мова моделювання) – мова графічного опису для об'єктного моделювання в області розробки програмного забезпечення, моделювання бізнес-процесів, системного проектування та відображення організаційних структур. UML є мовою широкого профілю – це відкритий стандарт, який використовує графічні позначення для створення абстрактної моделі системи, так званої UML-моделі. UML була створена для визначення, візуалізації, проектування та документування, в основному, програмних систем. UML не є мовою програмування, але на підставі UML-моделей можлива генерація коду. В UML використовуються наступні види діаграм (для виключення неоднозначності приведені також позначення англійською мовою):

- діаграма класів (class diagram / static structure diagram);
- діаграма компонентів (component diagram);
- діаграма композитної / зіставної групи (composite structure diagram);
- діаграма розгортання (deployment diagram);
- діаграма пакетів (package diagram);
- діаграма об’єктів (object diagram);
- діаграма діяльності (activity diagram);
- діаграма станів (state machine diagram);
- діаграма варіантів використання (use case diagram);
- діаграма комунікації (communication diagram);
- діаграма послідовності (sequence diagram).

До переваг UML відносять:

- UML об'єктно-орієнтований, в результаті чого методи опису результатів аналізу та проектування семантично близькі до методів програмування на сучасних об'єктно-орієнтованих мовах;

- UML дозволяє описати систему практично з усіх можливих точок зору та різні аспекти поведінки системи;
- діаграми UML порівняно прості для читання після досить швидкого ознайомлення з його синтаксисом;
- UML розширює та дозволяє вводити власні текстові та графічні стереотипи, що сприяє його застосування не тільки в сфері програмної інженерії;
- UML набув широкого поширення й динамічно розвивається.

2.2 Діаграма діяльності

Діаграма діяльності або діаграма активності (англ. Activity Diagram) – UML-діаграма, на якій показані дії, стану яких описано на діаграмі станів. Під діяльністю (англ. Activity) розуміється специфікація виконуваної поведінки у вигляді координованого послідовного та паралельного виконання підлеглих елеме0нтів – вкладених видів діяльності та окремих дій (англ. Action), з'єднаних між собою потоками, що йдуть від виходів одного вузла до входів іншого. Діаграма виглядає найбільш простою, оскільки нагадує звичну всім блок-схему. Насправді ж діаграма активності – це щось більше, ніж блок-схема, хоча цілі у них схожі – обидві вони відображають певний алгоритм. Нотація UML пропонує п'ять видів системи:

- вид системи з точки зору прецедентів;
- вид з точки зору проектування;
- вид з точки зору процесів;
- вид з точки зору розгортання;
- вид з точки зору реалізації.

При цьому кожен з перерахованих способів подання системи може містити послідовності дій, які можуть бути описані за допомогою алгоритмів. Ось тут, так би мовити, і виходять на сцену діаграми діяльностей. Взагалі кажучи, будь-який елемент моделі, що має динамічну поведінку, може бути

доповнений діаграмою діяльності – саме для уточнення цієї самої динаміки. Гарно підібраний по контексту приклад це можливість застосування діаграм активності для опису бізнес-процесів, що існують в компанії (нотації Grapes-BM, BPML / BPMN та інші). Ось вже і є безпосередньо динаміка [9].

Саме на діаграмі діяльності представлені переходи потоку управління від однієї діяльності до іншої. Це, по суті, різновид діаграми станів, де всі або більша частина станів є деякими діяльностями, а всі або більшість переходів спрацьовують при завершенні певної діяльності та дозволяють перейти до виконання наступної. Діаграма діяльності може бути приєднана до будь-якого елементу моделі, що має динамічну поведінку. До речі, логічніше говорити не «діаграма діяльності», а «діаграма діяльностей» – у множині [10].

Діаграми діяльності використовуються при моделюванні бізнес-процесів, технологічних процесів, послідовних та паралельних обчислень. Діаграми діяльності складаються з обмеженої кількості фігур, з'єднаних стрілками.

Найбільш близьким та точним аналогом діаграм діяльності є математично строгі дракон-схеми візуальної алгоритмічної мови ДРАКОН. Більш віддаленим аналогом діаграм діяльності є схеми алгоритмів по ГОСТ 19.701-90 [9, 11].

Для дипломного проекту було розроблено діаграму, що зображена на рис. 2.1. Вона відображає процес перевірки валідації реєстрації. Якщо файл валідації не був знайдений, або ж ключі не співпадають між собою, або не співпадають з ідентифікатором пристрою, то відбудеться запит на реєстрацію. Цей підхід дає захист від змінення даних файлу, його видалення, та навіть, якщо гру буде переміщено на інший пристрій, то вона не запрацює.

2.3 Діаграма класів

Діаграма класів (англ. Static Structure diagram) – діаграма, що демонструє класи системи, їх атрибути, методи і взаємозв'язку між ними. Являється

однією з основних UML-діаграм. Це набір статичних, декларативних елементів моделі. Діаграми класів можуть застосовуватися й при прямому проектуванні, тобто в процесі розробки нової системи, та при зворотному проектуванні – описі існуючих та використовуваних систем. Інформація з діаграми класів безпосередньо відображається в вихідному коді програми – в більшості існуючих інструментів UML-моделювання можлива кодогенерація для певної мови програмування (зазвичай Java або C++). Таким чином, діаграма класів – кінцевий результат проектування та відправна точка процесу розробки [10].

Діаграма класів займає центральне місце в проектуванні об'єктно-орієнтованої системи. Нотація класів використовується на різних етапах проектування та будується з різним ступенем деталізації. Мова UML застосовується не тільки для проектування, але і з метою документування, а також створення ескізів проекту.

У будь-якому об'єктно-орієнтованому процесі проектування діаграма класів є результатом, тому що вона є моделлю, найбільш близькою до реалізації, тобто коду. Існують інструменти, що здатні перетворити діаграму класів в код – такий процес називається кодогенерацією та підтримується безліччю IDE та засобів проектування.

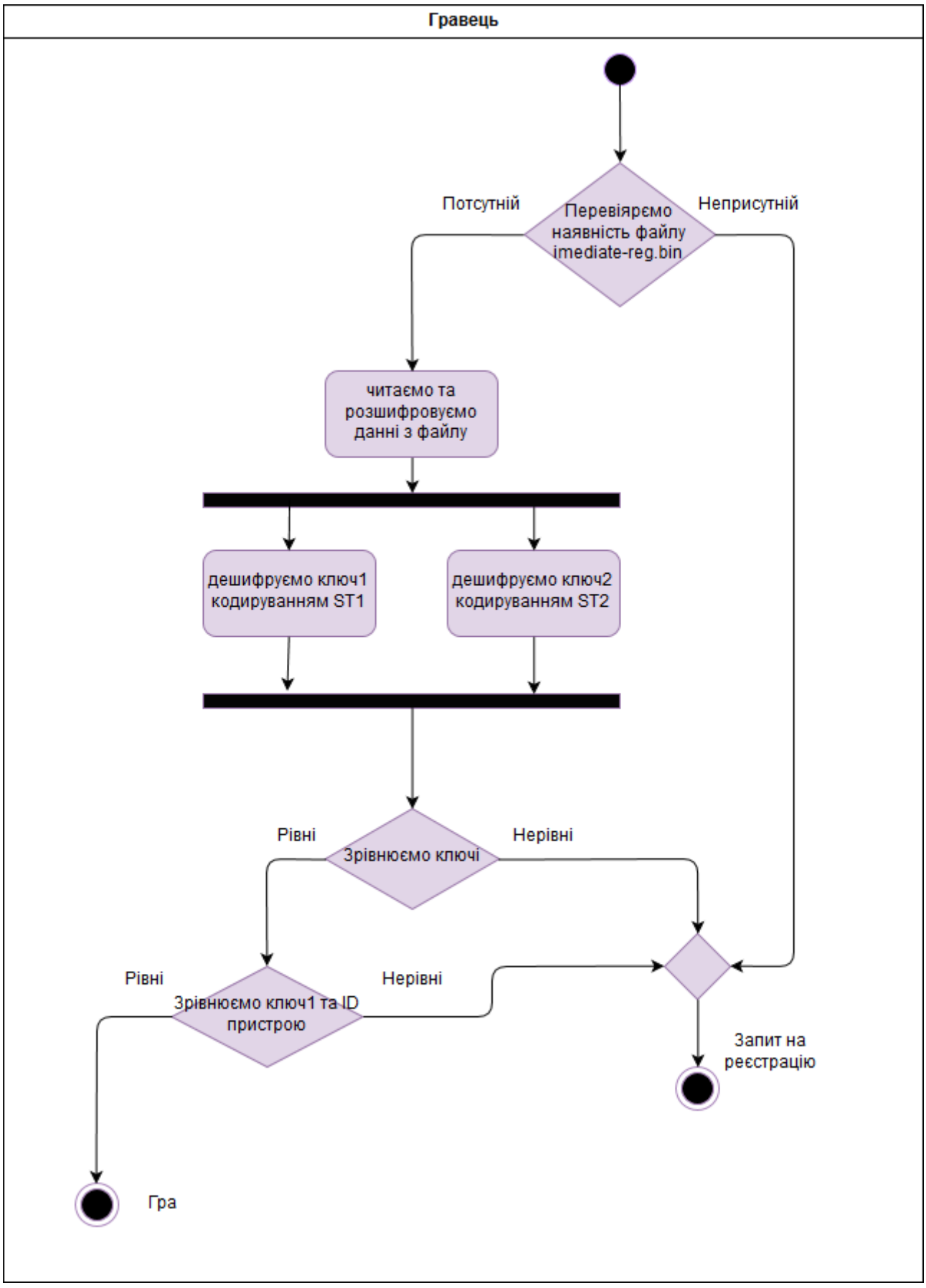

Рисунок 2.1 – UML діаграма активності процесу «Валідації реєстрації»

Наприклад, кодогенерацію виконує Visual Paradigm (доступно у вигляді плагінів для безлічі IDE), нові версії Microsoft Visual Studio, такі засобів UML-моделювання як StarUML, ArgoUML і ін. Щоб побудувати по діаграмі хороший код, вона повинна бути досить детальною. Для дипломного проекту діаграма класів будувалася за допомогою IntelliJ IDEA.

В Java типи, зазначені з великої літери, називаються примітивними (наприклад, double або boolean). Значення атрибутів такого типу безпосередньо зберігається в об'єкті. Типи, зазначені з великої літери, називаються посилальними (наприклад, String або ConnectDB). В атрибуті такого типу зберігається посилання на об'єкт, створений на базі відповідного класу.

В Java методи, що не повертають результату, позначаються за допомогою модифікатора void. Для інших методів обов'язково повинен вказуватися тип повертається результату. Виняток становлять конструктори, для яких тип результату не вказується, а повертається значення є посиланням на об'єкт, який був створений при виклику конструктора. Сучасні case-засоби при розробці класів, як правило, працюють в режимі синхронізації діаграм та вихідних текстів. Тобто, якщо змінюється діаграма класів, то це призводить до автоматичного коригування тексту програми та навпаки [12].

Існує два види:

- статичний вид діаграми розглядає логічні взаємозв'язки класів між собою;
- аналітичний вид діаграми розглядає загальний вигляд і взаємозв'язку класів, що входять в систему.

Існують різні точки зору на побудову діаграм класів в залежності від цілей їх застосування:

- концептуальна точка зору – діаграма класів описує модель предметної області, в ній присутні тільки класи прикладних об'єктів;
- точка зору специфікації – діаграма класів застосовується при проектуванні інформаційних систем;

– точка зору реалізації – діаграма класів містить класи, використовувані безпосередньо в програмному коді (при використанні об'єктно-орієнтованих мов програмування).

Основні властивості діаграми:

– інкапсуляція захищає внутрішній устрій об'єкта та реалізується шляхом обмеження доступу до атрибутів та операцій класу з інших частин програми;
– узагальнення дозволяє повторно використовувати вже існуючі рішення, створюючи нові класи шляхом успадкування від наявних класів;
– поліморфізм дозволяє працювати з групою різнорідних об'єктів однаковим чином, не замислюючись про відмінності в реалізації;
– інкапсуляція, успадкування та поліморфізм – три кити, на яких тримається ООП;
– у будь-якій системі між об'єктами існують відносини різних типів;
– відношення залежності означає, що реалізація одного класу залежить від специфікації операцій іншого класу;
– асоціація висловлює ставлення між декількома рівноправними об'єктами та може мати напрямок, ролі та кратність, а також зображуватися у вигляді класу асоціації;
– композиція та агрегація використовуються, якщо між об'єктами існують відносини типу «частина-ціле», причому композиція передбачає, що частини не можуть існувати окремо від цілого [13].

Для проектування відеогри було створену діаграму класів (додаток Г), що відображає необхідний контент класів для реалізації гри.

3 ПРОЕКТНІ ТА ТЕХНІЧНІ РІШЕННЯ

При плануванні та розробці було вирішено створити спочатку ігровий двигун, на базі якого вже розроблювати відеогру. Для розробки було застосовано мову Java та бібліотеку LibGdx. Також, для захисту jar-файлу від перегляду вихідного коду проекту була використана мова C. А для захисту від переносу гри на різні пристрої було розроблено реєстрацію.

Для реалізації своєї частини проекту мені як програмному розробнику була потрібна база спрайтів та атласів, тобто весь арт, що буде надалі реалізовано у ігровому продукті. Далі продемонстровано декілька з них: на рис. 3.1 зображено атлас, що містить набір кадрів для анімації кнопки; на рис. 3.2 зображено набір кадрів анімації обертання ключа; рис. 3.3 зображує атлас спрайтів декорацій.

Рисунок 3.1 – Атлас спрайтів кнопки «New game»

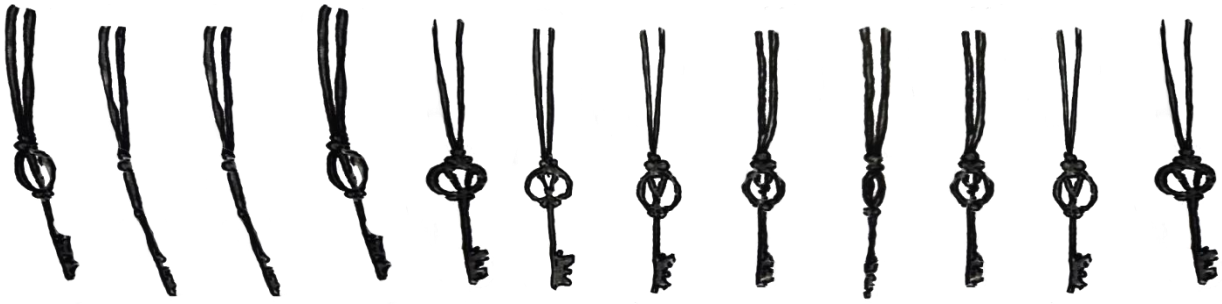

Рисунок 3.2 – Атлас спрайтів обертання ключа

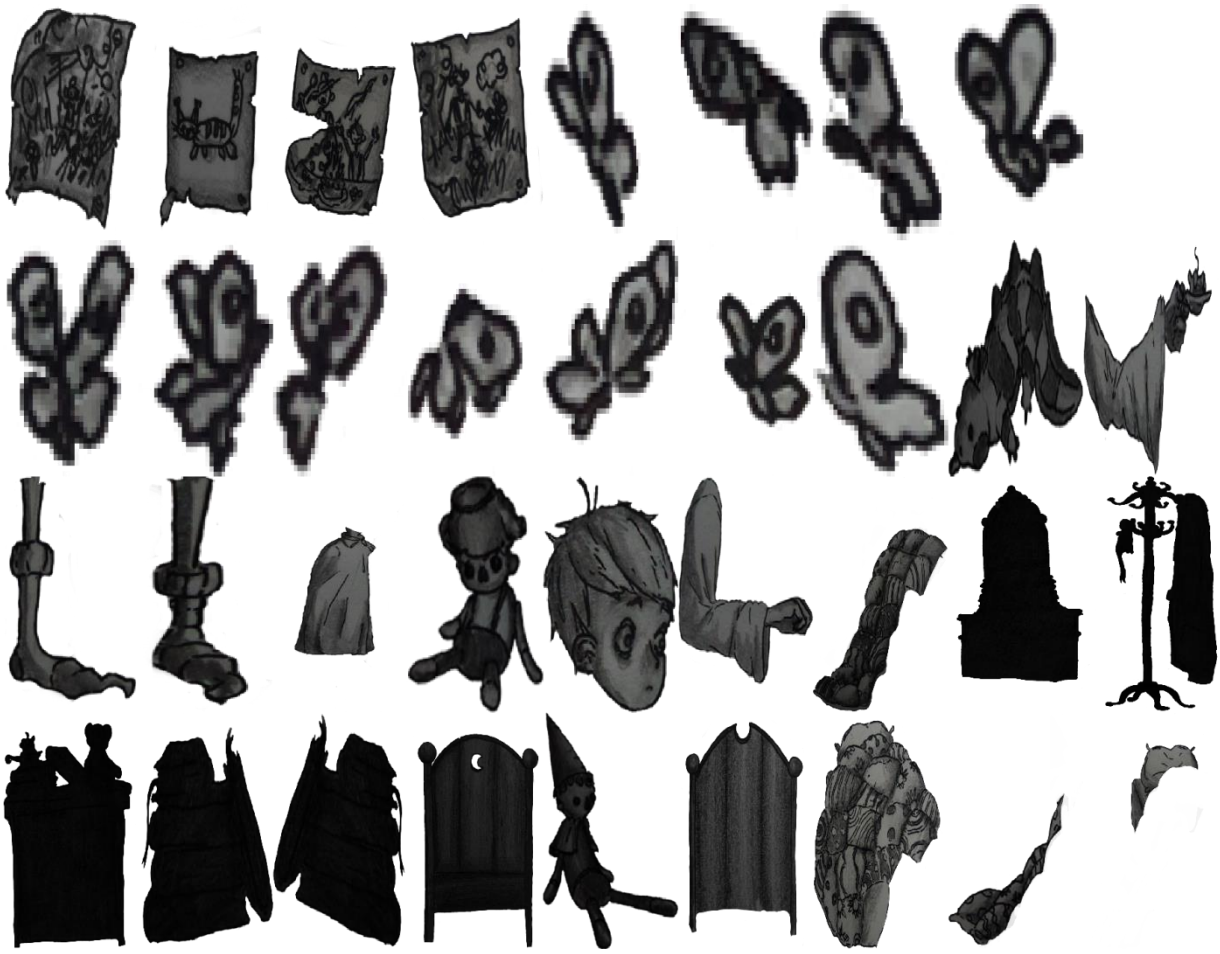

Рисунок 3.3 – Атлас спрайтів декорацій

3.1 Захист jar-файлу від перегляду

Так як Java байт-код являється беззахисним, довелося його приховувати. Для цього було використано можливість мови Java – можливість динамічно підключати зкомпільовані Java-класи. Для початку було написано бібліотеку на мові C, що приймала масив байтів у шифрованому вигляді, та повертала у розшифрованому. Також був розроблений нативний виконувальний файл під ОС Windows – exe-файл, що призначений для запуску jar-файлу та розшифруванню шифрованих класів. Захист не являється на 100%, але знавці мови Java, хто не знають мови Асемблер, та не знають як дизасемблерувати, обійти цей захист не просто зможуть. Алгоритм запуску проекту зображений у додатку Б.

3.2 Реєстрація продукту

Для реєстрації нам потрібно було сервер та база даних. Сервер також написаний на мові Java, а база даних на мові OracleDB. Взагалі, реєстрація було розроблена для зберігання налаштувань та прогресу гравця у грі. Надалі, коли проект розростеться, на цій основі буде реалізовано систему оплати, яка буде враховувати випадки, коли гравець видаляє куплену гру, а потім безкоштовно встановлює її знову. Алгоритм реєстрації зображений у додатку В.

Суть реєстрації у тому, щоб захистити продукт від переносу на інші пристрої. Тобто, якщо гравець купив гру та вирішив нею поділитися зі своїм другом, передавши гру на його комп’ютер, наприклад, то вона там не запрацює, бо не співпаде ID пристрою. Також, якщо гравець видалив гру, а потім заново вирішив її встановити, він не зобов’язаний наново платити за неї. Коли гравець запустить гру, вона перевірить наявність його реєстраційних даних в базі даних, та, якщо дані присутні, вона підтвердить реєстрацію безкоштовно.

Реєстрація ще потрібна для можливості зберігання ігрового прогресу на сервері. Це дуже вигідно економить місце на віртуальному просторі гравця. Також це забезпечує цілісність даних.

3.3 SceneEditor

Так як розташуванням ігрових об’єктів, декорацій та анімацій повинен займатися саме митець, а не програміст, потрібно було розробити зручне середовище для можливості будувати карти. Для зручного та гнучкого налаштування рівня було розроблено графічний редактор для побудови карт SceneEditor.

Редактор включає в себе можливість додавати зображення та анімації, відділяти їх шарами. Також базовий набір митця. Сюди входять такі інструменти та ефекти як:

- кисть;
- олівець;
- піпетка;
- лінія;
- крива;
- керування рівнями кольорів;
- відтінки;
- інверсія кольорів.

Редактор дозволяє зручно розташовувати об’єкти та декорації на полотні. Також є можливість відігравати анімації, що являє собою частковий рендер мапи.

Ще одна дуже важлива особливість редактора в тому, що він має інструменти по роботі з межами об’єктів та визначення їх колізій. Тобто митец власноруч може встановити межі для об’єктів як вважає потрібним.

Останньою особливістю, але не останньою по важливості являє собою створення атласу рівня. Для генерації самої мапи спрайтів використовується засіб LibGdx під назвою TexturePacker, але метадані атласу зберігаються у форматі, визначеним двигуном під назвою SimpleDataSerializer.

Рисунок 3.4 – Редактор побудови ігрової сцени

3.4 Розробка відеогри

3.4.1 Налаштування двигуна та розробка меню

Зазвичай розробка гри починається з головного меню, а вже потім реалізовується сам ігровий процес. Але перш ніж почати розробку меню, потрібно було підготовити, так би мовити, основу, тобто налаштувати параметри вікна та ввімкнути потоки, що надалі будуть займатись графічним контекстом та поточною обробкою. Цим займається початковий клас Lcynte. Далі було створено на базі класу InputProcessorAbstract клас InputPrcs, що призначений для відловлювання натискань клавіш клавіатури та миші. Та, нарешті, останнім класом, що відіграє роль підготовки був IniterGO (більш детально зображено у додатку М). Його головною роллю було ініціалізація всіх об’єктів гри, заповнення ігрового списку, перехід між режимами та вивільнення ресурсів, що вже не є потрібними. Також цей клас розподіляє одну те-

кстуру (в даний момент це атлас спрайтів) на регіони та підганяє розміри, що відповідають розміру екрана.

Головне меню складається з фону, напіврухомого хлопчика, кнопки та іскор, що світяться. Фон це лише одне статичне зображення, тобто це об'єкт класу OneSpriteSGO, що був описаний як елемент пакету sgo. Для створення іскор було створено клас на основі DynamicGO. Для руху іскор було задіяно екземпляри класу ForceS_F. Кожна іскра летить зверху та зліва. Коли іскра достигає правого або нижнього краю ігрового екрану, вона переініціалізується, тобто змінює свої координати та форму. Ось приклад коду цієї дії:

```
if (RANDOM.nextBoolean())
{
//--:     переміщення сніжини зліва від ігрового екрану
     x = -RANDOM.nextInt(sizeMax);
     y = RANDOM.nextInt(SCREEN_HEIGHT);
}
else {
//--:     переміщення сніжини зверху від ігрового екрану
     y = getRelHI(-RANDOM.nextInt(sizeMax));
     x = RANDOM.nextInt(SCREEN_WIDTH);
}
```

А ось алгоритм, що відображає переініціалізацію іскри, що вийшла за межі екрану. Тут визначається нова швидкість, розмір та спрайт з атласу спрайтів:

```
((ForceS_F)force).setEdge(RANDOM.nextFloat()*(forceMax -
forceMin) + forceMin);
sizeX = sizeY = RANDOM.nextInt(sizeMax - sizeMin) +
sizeMin;
texture.setRegion(30*RANDOM.nextInt(10), 0, 30, 30);
```

Наступним кроком було створення кнопок. Це було легко, адже у двигуні вже заздалегідь було створено відповідний клас. Єдине, що треба відзна-

чити, то це бібліотеку Imager, що значно прискорила роботу по створенню кадрів анімації та атласів з спрайтів. Саме завдяки цій бібліотеці було створено обертання на колірний перехід під час анімації.

Й останнім штрихом головного меню було створення напіврухомого хлопчика. Для нього, як і для іскор, було розроблено клас, що є child-класом DynamicGO та також без ForceS_F не обійшлося. Задача хлопчика складається у тому, щоб реагувати на рух комп'ютерної миші, а саме переміщувати руку хлопчика у положення, котре буде найближчим до курсору миші. З початку було запропоновано реалізувати такий рух анімаційно, але незабаром було прийнято реалізовувати програмно і на це є декілька причин. По-перше, так виходить універсально й доволі просто можна підлаштувати, якщо це потрібно, тоді як для кадрів це не так просто й швидко зробити. По-друге система руху на основі сил вже була реалізована на рівні двигуна. По-третє, в анімаційному підході прийшлося б створити просто величезну кількість кадрів, при чому на любий випадок, тобто від будь-якої першої кнопки нарисувати анімацію польоту до будь-якої іншої. Хоча в дуже старих відеоіграх так робилося, з причини малого місця для обробки даних.

Окремо варто згадати про музику, що грає у головному меню та звуки сфокусування та натискання кнопок. Вони були створені у програмі FLS.

Таким чином перший крок по створенню відеогри пройдений (рис. 3.5).

Рисунок 3.5 – Ігрове меню гри «No Sleep»

3.4.2 Генерація рівня

За допомогою редактора SceneEditor створюються лише кімнати, але саме генератор з’єднує їх між собою. Генератор має можливість генерувати рівень на основі всіх кімнат або ж лише їх частини. Також генератор враховує стилізацію кімнат, тобто рівень не може складатись з кімнат різної стилізації. Кімнати є п’яти типів:

- з одним входом або ж кутова;
- з двома входами типу лінія;
- з двома входами типу куток;
- з трьома входами;
- з чотирма входами.

Генератор враховує усі п’ять типів кімнат та розташовує їх так, щоб двері сусідніх кімнат співпадали. Також генератор займається створенням

динамічних об’єктів, наприклад, ключі або ж ліхтарик, а також ворогів та інших інтерактивних NPC.

Спочатку, генератор визначає кількість кімнат на рівні. Діапазон залежить від складності гри, а також від самого рівня. Далі, генератор визначає яку загальну площу будуть займати кімнати. Відштовхуючись від площі, генератор визначає типізацію кімнат, а також їх розміри та положення.

Наступним кроком виступає генерація персонажів. Персонажі розташовуються в залежності від їх фізичних потреб. Наприклад, гладкий персонаж буде розташований поблизу кухні, а сильний – поряд зі спортзалом.

Останнім кроком генератора це розміщення інтерактивних об’єктів, а також квесів й об’єктів, пов’язаних з ними. Так, наприклад, для ключа може згенеруватись сундук, шафа та навіть кімната.

3.4.3 Взаємодія об’єктів та поведінка NPC

У грі йде підтримка подієво-орієнтоване програмування, за рахунок якого об’єкти можуть взаємодіяти між собою. У грі існує декілька типів взаємодії об’єктів, а саме:

- розмова;
- атака;
- колізування;
- поглинання.

Розмова – це спосіб отримання інформації у нейтрально налаштованих NPC. Під час взаємодії відкривається щось по типу чату у котрому можна вибирати варіанти запитань та відповідей. Кожен NPC має свою налаштованість по відношенню до головного героя. В залежності від запитань та відповідей, а також квестів залежить зміна настрою персонажа. Таким чином розмова може перетворювати NPC на ворогів або ж навпаки – на добре налаштованих персонажів.

Атаку застосовують лише вороже налаштовані персонажі. Ця взаємодія приносить шкоду головному герою, та може навіть вбити його.

Колізування – це не перетинання об’єктів. Цю особливість реалізовує сам двигун. Головний герой та інші персонажі колізують зі статичними об’єктами, наприклад, постіль та й один з одним. Межі колізії встановлюються редактором сцени.

Й остання взаємодія це поглинання. Поглинання дозволяє підбирати об’єкти й використовувати. Це може бути, наприклад, ключі, їжа тощо.

У кожного типу персонажа є свої особливості у плані поведінки. Добре налаштовані NPC зазвичай стоять на одному місті, зрідка відвідуючи місця, наприклад, вбиральню. Псевдо випадково згенеровані вороги ж починають блукати по рівню від кімнати до кімнати. Як тільки ворог помітить головного героя, він починає його переслідувати. Переслідування – це окремий режим NPC, при якому він починає шукати найкоротший шлях до головного героя. Такий шлях обраховується за допомогою алгоритму Дейкстри (рис. 3.6). Настрій персонажів також залежить від часу доби – чим темніше, тим вороже вони налаштовані, чим світліше – тим добріше.

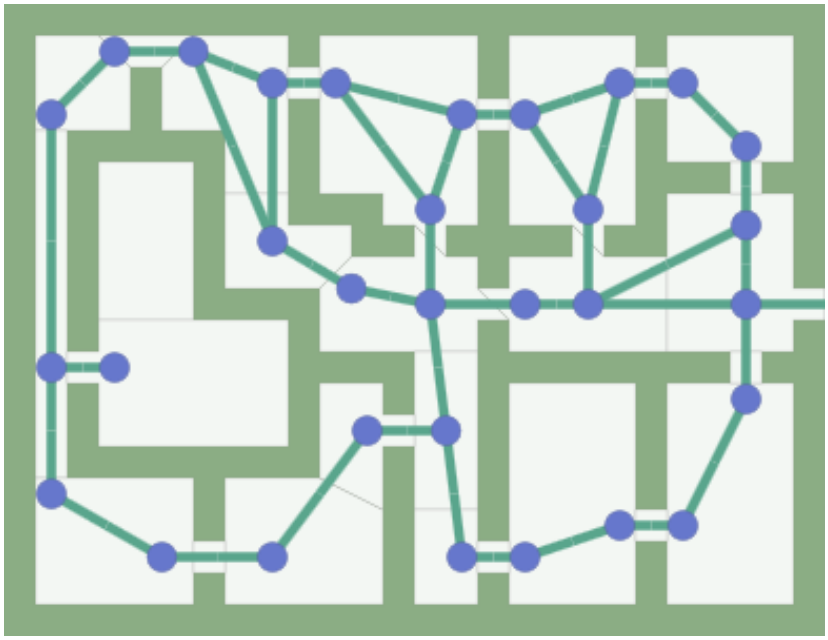

Рисунок 3.6 – Схема пошуку шляху за допомогою алгоритму Дейкстри

4 АНАЛІЗ ІГРОВОГО ДВИГУНА SVS ATEUNANE

4.1 Розробка ігрового двигуна SvSAteunane

Для того, щоб приступити для розробки ігрового продукту потрібен був ігровий двигун, так як він займається тим, що оброблює важкі низькорівневі процеси та надає інтерфейс для написання гри.

Що ж, все розпочинається з пакету gamecore (рис. 4.1), та до його складу входять:

- InputProcessorAbstract;
- MouseCollisable;
- Blockable;
- GameObject;
- GameList;
- animation;
- force;
- handler;
- sgo;
- dgo.

Клас InputProcessorAbstract є абстрактним, він представляє собою абстракцію для перехвату ввідних пристроїв гравця (комп'ютерна миша, клавіатура, дотик до сенсору смартфону, геймпад тощо). Інтерфейс MouseCollisable призначений для ідентифікації класу на те, що він має можливість реагувати на курсор миші, якщо вона в його області.

Інтерфейс Blockable дозволяє блокувати реакцію на ввідний пристрій, якщо це потрібно. Клас GameObject являється початком для всіх ігрових об'єктів гри. Він абстрактний, тому його екземпляр створити не можна, та він надає інтерфейс для створення більш типізованих інших ігрових об'єктів на його основі. Найважливішими абстрактними методами є draw – що призначений для малювання об'єкта на вікні програми, та метод free – що призначений для вивільнення непотрібних ресурсів. У ньому також є поля, що ви-

значають розмір текстури. Справа в тому, що в залежності від розміру графічної робочої області, тобто розмірів вікна потрібно змінювати розміри й текстури рисованих об’єктів, інакше буде виглядати незакінченим й рубаним.

Клас GameList являється нічим іншим як інтерфейсом над роботою зі звичайними списками бібліотеки SvSList. Справа в тому, що для раціональної роботи потрібно декілька дубльованих списків, що мають один й той же контент, але різний порядок розташування.

4.2 Пакет animation

Пакет animation надає класи, що працюють з анімацією ігрових об’єктів (рис. 4.2). Класи, що входять в цей пакет:

- AnimationAbstract;
- Animation;
- AnimationCycle;
- AnimationHandle;
- AnimationHandler.

Він надає інтерфейс для класів-оброблювачів анімацій та є ядром їх обробки – тобто паралельний потік, який постійно робить обхід списку анімацій, та запускає методи обробки.

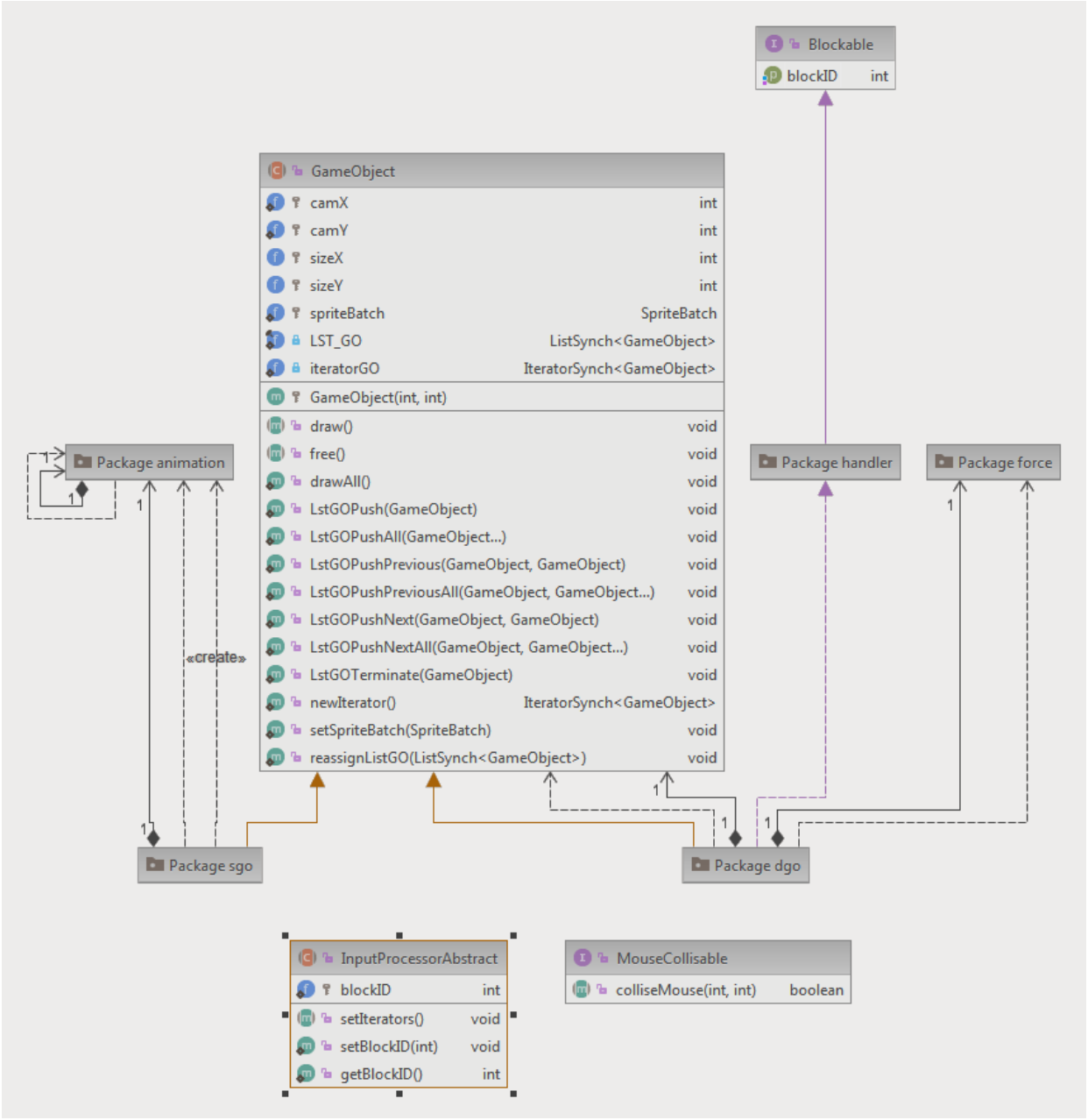

Рисунок 4.1 – Діаграма класів пакету «gamecore»

Для анімації об’єкту є у крайньому випадку два підходи – векторний рух або ж набір кадрів. Для даного проекту було обрано саме набір кадрів, адже таким чином можна передати більш чітко перетворення. Правда, такий підхід більш затратний по часу, адже малювання кожного кадру забирає не мало часу. Для даного двигуна було створено класи, що відповідають усім потребам для реалізації тої чи іншої анімації.

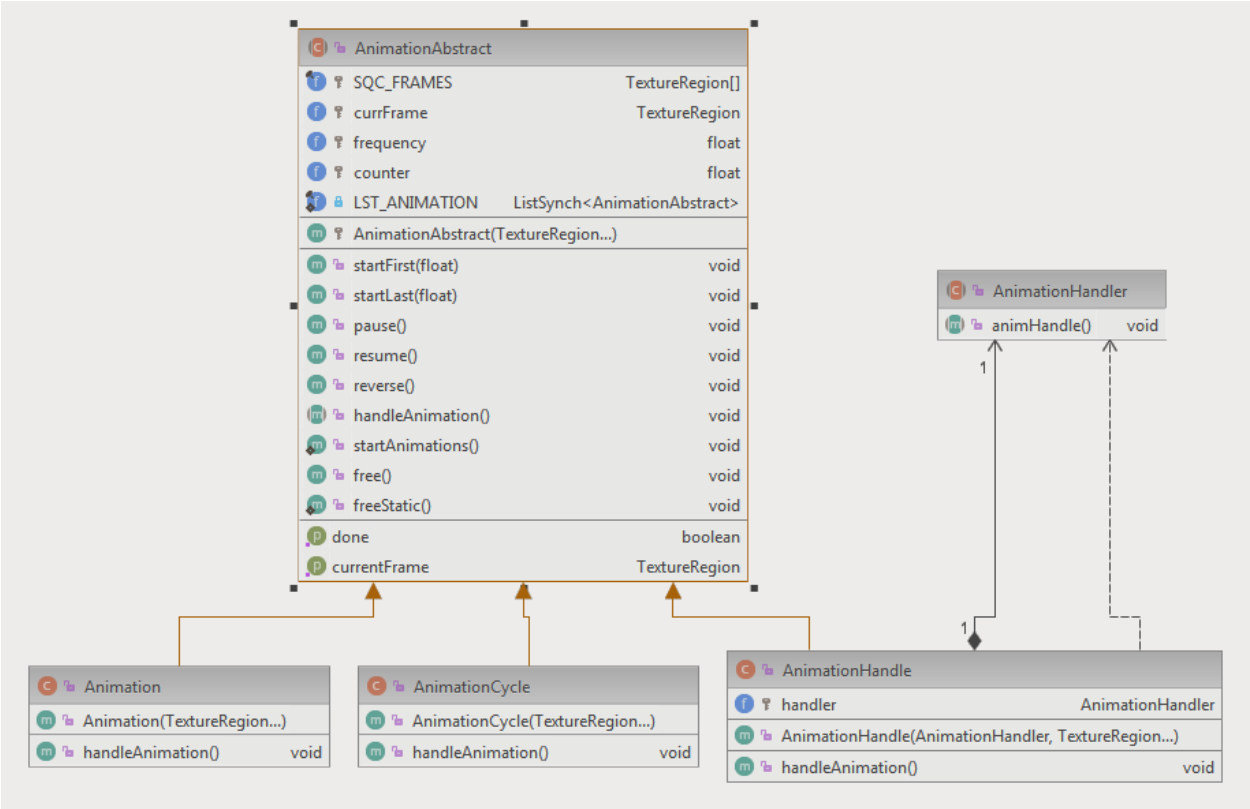

Рисунок 4.2 – Діаграма класів пакету «animation»

Загальний цикл, що постійно крутиться з затримкою 1 мс, реалізовано наступним чином:

```
do
{
//==:     Цикл, що обходить кожну анімацію списку
     for
     (
          AnimationAbstract anim = iterator.next();
          anim != null;
          anim = iterator.next()
     )
     Animation.handle();
}
```

Кожна гра має свою динамічність. Для створення того чи іншого рівня динамічності використовується затримка виконання потоку. Зазвичай роб-

лять 60 кадрів за секунду, також зустрічаються на 30 кадрів за секунду. Наступна частина коду зображує цей процес:

```
try
{
     Thread.sleep(1);
}
catch (InterruptedException exc)
{
     exc.printStackTrace();
}
```

Клас Animation є найпершим та найпростішим з анімаційних класів. Його основна суть у тому, щоб грати анімацію за початку до кінця або навпаки – з кінця до початку, лише один раз. Потім можна знову програвати анімацію. Розглянемо код обробки для цього класу детальніше. Синхронізований метод обробки анімації має наступний вигляд:

```
public synchronized void handleAnimation()
{
//--:     Умова – якщо анімація грає вперед
     if (frequency > 0)
     {
     //--: Умова – якщо анімація вже завершена
          if (isDone)
          //--:     то вийти з методу
               return;}}
```

Далі, якщо лічильник більший або рівний кількості кадрів анімації, буде виконуватись наступна частина:

```
if (counter >= SQC_FRAMES.length)
{
//--:     встановити значення лічильника на останній кадр анімації
```

```
            counter = SQC_FRAMES.length - 0.0001f;
        //--:     встановити, що анімація вже завершена
            isDone = true;
        //--:     вийти з методу
            return;
        }
```

Також у коді передбачається випадки, якщо анімація грає у протилежну сторону.

Клас AnimationCycle створений для програвання анімацій постійно без додатковий умов зупинки. Даний клас здебільше схожий на клас Animation, але він по завершенню анімації виконує якусь дію. Ця дія є реалізацією абстрактного класу AnimationHandler, що має лише один метод animHandle. Це вигідно, адже це позбавляє додаткових перевірок, коли потрібно відловити завершення анімації.

Пакет force призначений для обробки сил. Його складають:

- ForceInterface;
- ForceInterfaceEx;
- Force_F;
- ForceS_F;
- ForceR_F;
- Controller_F.

Вся ієрархія починається з інтерфейсу ForceInterface, що надає основний інтерфейс для розроблення класу сил. На його основі будується клас Force_F, що реалізує основні маніпуляції з силами, такі як додавання, віднімання, задати значення нуль тощо. Наступним по ієрархії йде інтерфейс ForceInterfaceEx, що є child-інтерфейсом класу ForceInterface, та надає більш розширений інтерфейс для сил.

Клас ForceS_F вже є реалізатором ForceInterfaceEx. Він може не просто додавати силу, а також реалізовую інерційну систему. Тобто цей клас надає можливість мати прискорення та інерцію, що значно покращує, наприклад, рух якого-небудь об’єкта. Завдяки цьому рух став більш плавний. Також тут

є обмеження максимального значення сили. Безпосередньо у цьому класі обмеження реалізовано окремо для кожної сили. Клас ForceR_F є практично ідентичною копією попереднього класу, але обмеження тут відбувається синхронізовано по кожній силі. На відміну від попереднього класу це більш точніше обмеження, але є більш важкішим для обробки.

Закриваючим класом даного пакету є клас Controller_F, що слугує для управління яким-небудь об’єктом. Він вже налаштований на певні клавіші та реакцію на них (переміщення вліво / вправо / вперед / назад). Саме цей клас буде подалі використовуватися в класі, що представляє собою ігровий об’єкт гравця у грі.

4.3 Пакети handler, sgo та dgo

Пакет handler надає набір інтерфейсів для різної обробки. В його склад входять інтерфейси:

- MouseMotionHandler;
- MouseTouchHandler;
- KeyHandler;
- RuntimeHanlder.

Задачі інтерфейсів дуже прості – MouseMotionHandler призначений для встановлення дії на переміщення курсора миші, MouseTouchHandler – для встановлення дії на натискання та відпускання миші, KeyHandler – для обробки дій на натискання або відпускання будь-якої клавіші.

RuntimeHanlder призначений для постійної обробки об’єкта під час гри. Для виклику цього методу виділений спеціальний потік обробки. Цей метод в більшості випадків використовується для обробки колізій та переміщення ігрових об’єктів.

Пакет sgo є абревіатурою Static-Game-Object, тобто статичний ігровий об’єкт. Цей пакет являється набором класів, що призначені для статичних нерухомих, або частково рухомих об’єктів (додаток Ж). В цей пакет входять:

- StaticGO;
- MediateScreen;
- background;
- button;
- other.

Відкриває ієрархію клас StaticGO, child-клас класу GameObject. Він також є абстрактним та надає основний інтерфейс для статичних об’єктів. Від нього успадковується клас MediateScreen. Цей клас є Singleton, при чому лінивий, тобто його ресурси не будуть задіяні до першого звертання. Основним завданням класу є перехід з одного ігрового режиму в інший з програванням анімації для більш гарної візуалізації. Здебільшого цей клас використовується у класі-ініціалізаторі. Також тут передбачено інтерфейс по вивільненню основних ресурсів режиму.

На даний момент пакет background налічує лише один клас і це OneSpriteSGO. Його задача надати інтерфейс для класу, що має лише один спрайт, та що не має ні анімації та жодного додаткового спрайту.

Пакет button містить у собі класи, що представляють собою статичні ігрові об’єкти типу кнопки.

Пакет складають:

- Button;
- ButtonRound;
- ButtonHandler.

Клас Button являє собою прямокутну кнопку, та має набір методів для її керування. Крім спадкування класу StaticGO, кнопка додатково реалізує інтерфейси MouseMotionHandler – дає змогу програвати анімацію фокусування кнопки та навпаки, MouseTouchHandler – оброблює натискання миші, та MouseCollisable – забезпечує знаходження колізії кнопки з курсором миші. Розглянемо приклад методу, реалізованого від інтерфейсу MouseMotionHandler.

Для початку відзначимо, що метод ділиться на дві основні частини – якщо курсор миші колізує з кнопкою, та якщо ні. У випадку коли кнопка колізує, інакше кажучи курсор миші у межах кнопки, то виконується наступна частина коду: перевіряємо, чи курсор миші у межах кнопки:

```
if (colliseMouse(x, y))
```

далі перевірка чи є кнопка сфокусованою, якщо так просто вихід з методу, має наступний вигляд:

```
if (isFocused)
     return true;
```

Інакше кнопка була несфокусована, тому потрібно її сфокусувати, на ряду з цим програти мелодію:

```
soundFocused.play();
//--:     та запустити анімацію
animation.reverse();
//--:     встановити фокус кнопки
isFocused = true;
//--:     вийти з методу
return true;
```

Інакше, у випадку, коли курсор миші не колізує з кнопкою в силу вступає наступний код: перевіряється чи є кнопка несфокусованою, якщо так:

```
if (!isFocused)
//--:     вихід з методу
     return false;
//==:     Інакше, якщо вона є сфокусованою потрібно програти анімацію
animation.reverse();
//--:     зняти фокусування з кнопки
```

```
isFocused = false;
```

Клас ButtonRound є майже таким, як попередній клас, але являє собою кнопку у виді круга, відповідно й колізія з курсором миші буде працювати інакше. ButtonHandler – це клас-обробка, що призначений для обробки дії на натискання миші на кнопці.

Пакет other на даний час містить лише клас Cursor, що є безпосередньо курсором миші у грі. Справа у тому, що нативний курсор бібліотеки LibGdx не може встановити курсор миші напівпрозорим. Встановлений спрайт стане повністю прозорим у місцях де пікселі напівпрозорі. Для обходу цієї проблемі було розроблено власний клас.

Пакет dgo це абревіатура, що розшифровується як Dynamic-GameObject, тобто динамічний ігровий об’єкт. Цей пакет надає набір класів, що являються основою для всіх динамічних об’єктів, тобто тих, що постійно або майже постійно у русі.

Першим класом, що представляє пакет є DynamicGO – абстрактний клас, що є child-класом GameObject, та зразу реалізує інтерфейс RuntimeHandler, наголошуючи при цьому що об’єкт реально динамічний та потрібний у постійній обробці. Також цей клас має поле ForceInterface, вказуючи, що об’єкт належить до системи сил, та його рух буде від цього залежати.

Наступним класом, що представляє даний пакет являється CollisableDGO – child-клас DynamicGO. Цей клас також абстракний, та надає інтерфейс ігрового об’єкта, що сприятливий до колізії до будь-якого з об’єктів, що є child-класами класу CollisableDGO. Цей клас містить додатково hitbox для більш гарної колізії – оскільки всі текстури є прямокутними, а спрайти далеко не завжди такими, то колізія по прямокутнику була б не зовсім правильна. У сучасних іграх великих масштабів взагалі використовується полігональна система колізії, але для даного проекту це не вигідним.

Ось фрагмент коду, що реалізує обхід всіх здатних до колізії об’єктів:

```
for (GameObject go = iterator.next(); go != null; go =
iterator.next())
//--:     якщо об’єкт є колізуючим
     if (go instanceof CollisableDGO)
     //--:     та якщо він дійсно колізує з даним об’єктом
          if (colliseGO((CollisableDGO)go))
               reactOnCollision((CollisableDGO)go);
```

4.4 InLan

Спочатку для створення об’єктів та їх зручне налаштування була розроблена мова для ініціалізації InLan. Вона з’явилася в перших версіях двигуна. Основними перевагами є зручне створення об’єктів без нагромадження багато коду. Ця мова вирішує проблему створення конструкторів з багатьма параметрами, або ж багатьох set-методів, що виконаються лише раз – зразу ж після створення об’єкту. Також, бувають випадки, коли у двох різних об’єктів є поле на третій об’єкт, і це створює проблему, адже ці об’єкти знаходяться в різних місцях, та для вирішення цієї проблемі потрібно було костилювати та порушувати логіку структури проекту. Ще одна гарна якість у тому, що незалежно від об’єму ініціалізації, об’єм коду завжди має константний розмір – це розмір бібліотеки InLan. Можливість використання бібліотеки забезпечує рефлексія мови програмування Java. Також мова використовує бібліотеку CharLine, що реалізовує собою рядок. У кожного класу, що працює з рядком (String, StringBuffer, StringBuilder, StringTokenizer та інші) мають свої мінуси, та не мають тих методів, що потрібні. Мова InLan здалеку нагадує XML та JSON, але насправді вона відрізняється. Звісно ж, можна було б не вигадувати нічого, та писати код ініціалізації на вище описаних аналогах, але так код був би нагромаджений, й було б доволі складно працювати. Тому була створена нова мова ініціалізації.

Тут є чотири основні елементи:

- var;
- method;
- class;
- object;
- varray.

Var розроблений для звичайних змінних примітивних типів (int, char, Boolean та інші), method – для зберігання методу, щоб потім при необхідності можна було його викликати, class використовується для задавання статичних полів, object – для динамічних полів, й останній елемент це varray. Він призначений для масивів. Можна було б реалізувати масив на базі object, але так виходить більш простіше у роботі. Ось приклад ініціалізації за допомогою мови InLan: тут зображено як проводиться ініціалізація звуку для кнопки, коли вона фокусується. Так як звук для кожної кнопки однаковий, то це поле буде статичним, а отже відкриваємо елемент class. Далі вказуємо шлях до класу та назву поля:

```
<<class
     com.pack.Button;
     #soundFocused;
```

Але так просто встановити поле не вдасться, тому що воно встановлюється за допомогою методу newSound з інтерфейсу Audio. Об’єкт, що реалізує цей інтерфейс являється стичним полем класу Gdx, тому для встановлення звуку створюється метод на основі об’єкта, що є полем класу Gdx:

```
<method> newSound;
     <<object
          ;
          #audio; com.pack.Gdx;
          ();
     >>;
```

```
(com.pack.FileHandler);
```

Але тут також не так просто – вхідний параметр для методу newSound є об’єктом класу FileHandler. Його можна отримати викликом методу internal з об’єкта Files. Цей об’єкт також є полем класу Gdx. Як у попередньому випадку створюється метод на основі об’єкта Files, що є полем класу Gdx:

```
<method> internal;
<<object
     ;
     #files; com.pack.Gdx;
>>;
(string);
@@.("res/sound/button/focused.mp3");
```

4.5 JavaModuleAct

Але мови InLan було недостатньо для більш гнучкої роботи. Здебільшого, це стосується логіки. InLan, по своїй суті, не має логіки – це конфігурація. Звісно, можна було б розширяти межі мови, але її суть була в іншому і це було не зовсім правильним рішенням. Тому було прийнято інше рішення. Java має можливість не тільки вивантажувати об’єкти з пам’яті, а й класи. Отже, був розроблений модуль для роботи з динамічними класами, що відробивши, мали можливість вивільняти оперативний простір.

Даний приклад коду демонструє роботу класу InputReader, що потрібен для читання даних зі зовнішнього модуля.

```
public synchronized String readLine(long timeout){
          if (isReading)
               try{
                    wait(timeout);
               }
               catch (InterruptedException exc) {
                    exc.printStackTrace();
```

```
            }
        String line = buffer;
        buffer = null;
        isReading = true;
        return line;
    }
```

3.6 AutoDisposableResources

Цей модуль відповідає за автоматичне вивільнення ресурсів з пам’яті. При створенні менеджера ресурсів, вказується час на життя ресурсу. Якщо ще при завантаженому ресурсі, до нього відбувається звертання, його час на життя подовжується. Як тільки час на життя закінчується, ресурс вивільняється. Якщо ресурсу в пам’яті немає, система автоматично завантажує ресурс повторно. На рис 4.3 Зображено принцип роботи ресурсів.

Він доволі зручний – допомагає програмісту не контролювати завантаження та вивантаження ресурсів. Це особливо помітно, коли на проект великий та працює за багатьма різними ресурсами. В Java не всі ресурси вивантажуються за допомогою сміттяра, особливо класи Libgdx, які потребують ручного виклику метода disposable.

Наступний код показує, яким чином проводиться вивільнення ресурсу.

```
if (resources.getCount() > 0 && res[0].lastAccess < TIME){
    res[0].save();
    resources.remove(res[0]);
    final int COUNT = resources.getCount();
    for (int i = 0; i < COUNT; ++i){
    AResource r = res[i];
    if (r.lastAccess < TIME){
        r.save();
        resources.remove(r);
    }
    else
        break;
    System.gc()
```

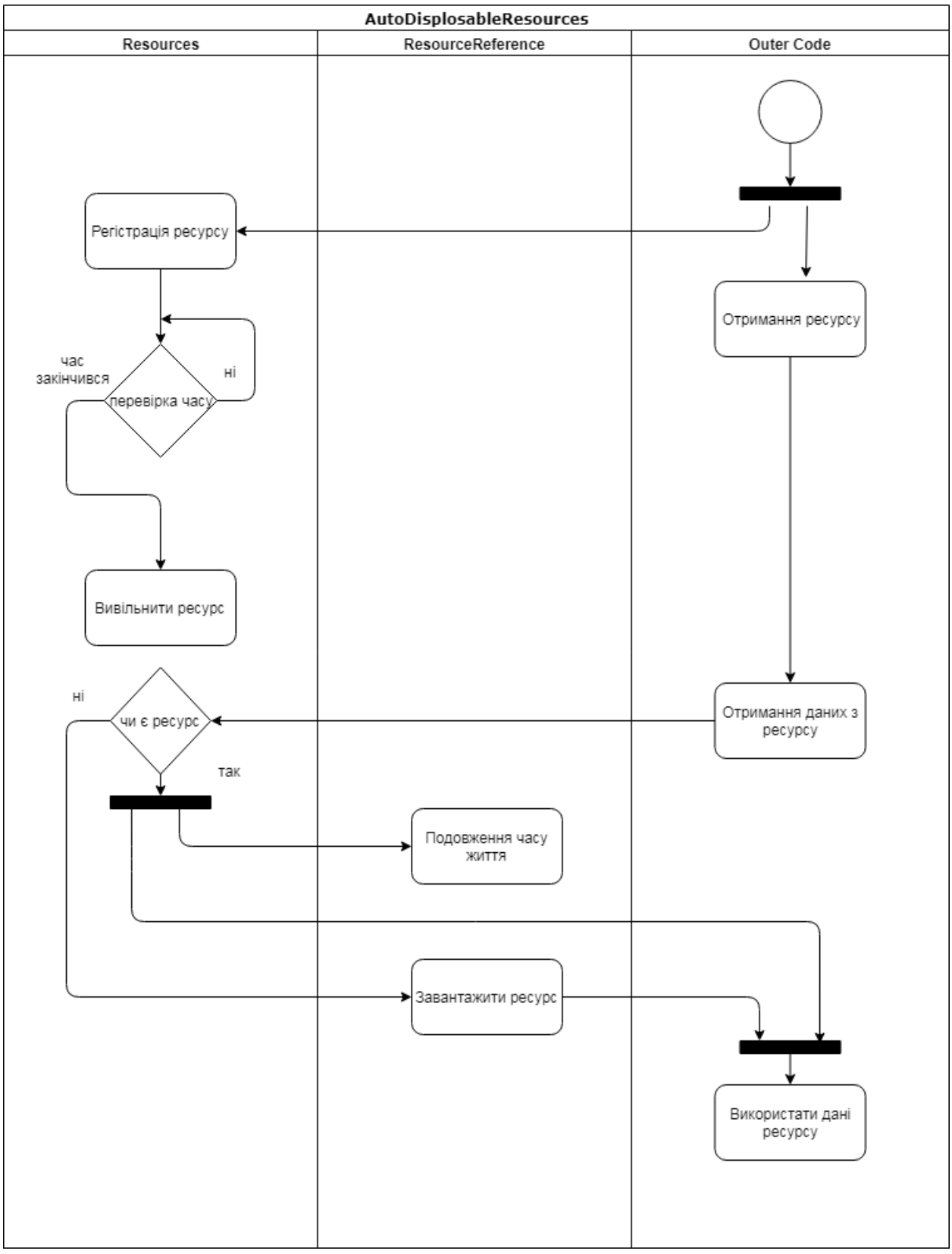

Рисунок 4.3 – Діаграма активності роботи ресурсів

4.7 Пакет thread

У більш великих проектах потрібно, щоб був вірний розподіл ресурсів. Пакет thread надає гнучку та зручну роботу с потоками. У житті ми дуже часто зустрічаємося з організацією пулу. Наприклад, коли йдете в їдальню, ви зустрічаєтеся з пулом підносів. Підноси організовані в пул; клієнтів може бути набагато менше, ніж таць, й навпаки. Коли таць багато, вони лежать без діла, коли таць мало, клієнти чекають, поки вони звільняться. Число таць, тобто розмір пулу, заздалегідь визначається так, щоб в більшості випадків клієнти не чекали таць. Однак трапляються години пік, коли клієнтів дуже багато. Просто нереально виділити окремий піднос кожного клієнта, та й не потрібно це. Клієнт все одно буде стояти в черзі до каси, так що витрати на підноси не принесуть реальних вигод. Це, звичайно, дуже далека і недосконала аналогія, але вона показує, що в природі та житті пул чого-небудь дуже часто використовується як найефективніше схема обслуговування запитів.

Розглянемо механізм роботи пулу потоків. Є головний потік додатку, що прослуховує запити на задачі. Пул потоків створюється заздалегідь або під час вступу першого запиту. Мінімальний розмір пулу звичайно вибирається рівним 1, однак це не принципово. Під час отримання запиту головний потік вибирає потік з пулу і передає йому запит. Якщо кількість потоків в пулі досягло максимуму, запит поміщається в чергу. Якщо кількість потоків менше максимального, і всі вони зайняті обробкою, створюється новий потік, який отримує клієнтський пакет на обробку. Якщо кількість потоків одно максимальному і всі потоки займаються обробкою, тобто активні, пакет ставиться в чергу і чекає звільнення одного з потоків. Алгоритми додавання потоків в пул і визначення оптимального розміру пулу сильно залежать від розв'язуваної задачі. Принцип роботи пулу зображений на рис 4.4.

Потоків у полу повинно бути у два рази більше, ніж число процесорів. Це дуже загальна рекомендація, і для деяких завдань вона не підходить. За великим рахунком є тільки два критерії, за якими можна визначати, потрібно

створювати новий потік чи ні. Ці критерії – завантаженість процесора і число пакетів задач. Якщо число пакетів перевищує певну кількість, і завантаженість процесора невисока, є сенс створити новий потік. Якщо пакетів мало, або процесор зайнятий більш ніж на 90 відсотків, додатковий потік створювати не слід. Видаляти потік з пулу потрібно, якщо він давно не обробляв клієнтські запити. При видаленні потоку потрібно стежити, щоб закінчилися всі асинхронні операції введення / виводу, початі цим потоком.

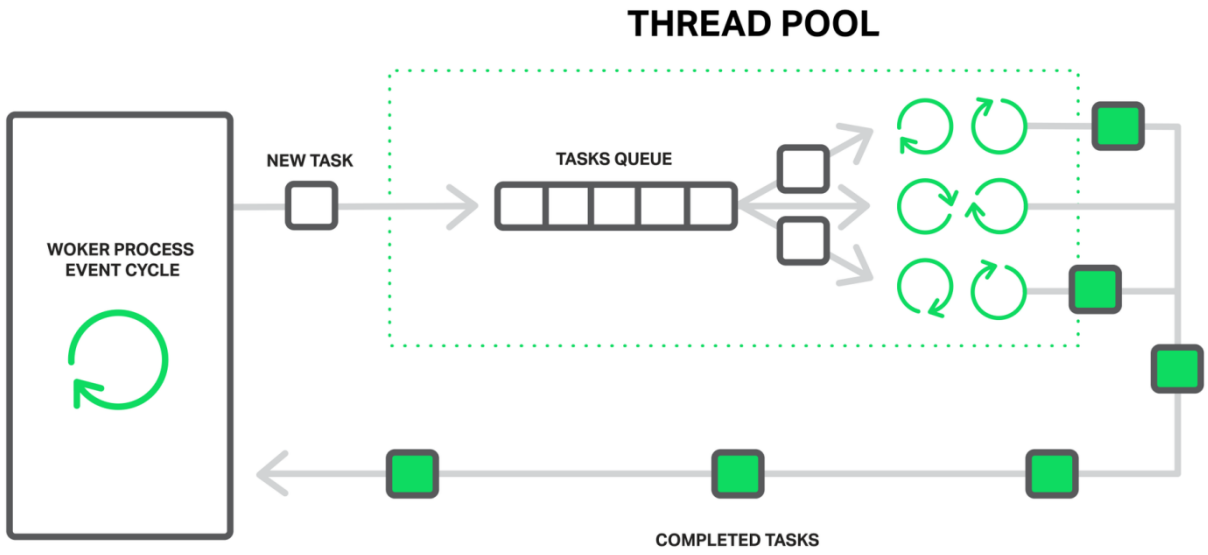

Рисунок 4.4 – Діаграма роботи пулу потоків

Також у пакет входить клас FlowHandler, що націлений на обробку задач у поточному потоці, або ж у паралельних, при цьому всередині кожного з них можуть створюватись нові задачі. По закінченню всіх задач викликається замикаюча задача. Це особливо зручно, коли потрібно виконати ряд задач, котрі напряму одна з одною не зв’язані, але їх результати будуть впливати на основний результат. Наприклад, таким же чином влаштовується сортування масивів, шляхом розбиванням на частини. Далі приведений код, ща відображає принцип роботи цього класу.

```
     LList<Runnable> unthreaded = this.unthreaded;
     do{
          Runnable item = null;
          synchronized (this){
               LVertex<Runnable> vertex = unthreaded.getFirst();
               if (vertex == null)
                    try{
                         if (threaded.getCount() == 0 &&
unthreaded.getCount() == 0){
                              if (action != null)

                                   action.onAction(getScope());

                         isActive = false;

                         return;
                    }

               wait();
               continue;
          }
          catch (InterruptedException exc) {
               exc.printStackTrace();
          }
          else{
               item = vertex.getItem();
               unthreaded.remove(vertex);
          }
     }
     item.run();
}while (true);
```

Останнім елементом пакету є підтримка асинхронних потоків. Асинхронність дозволяє винести окремі завдання з основного потоку в спеціальні асинхронні методи або блоки коду. Особливо це актуально в графічних програмах, де тривалі завдання можу блокувати інтерфейс користувача. І щоб цього не сталося, потрібно задіяти асинхронність. Також асинхронність несе вигоди в веб-додатках при обробці запитів від користувачів, при зверненні до

баз даних або мережевих ресурсів. При великих запитах до бази даних асинхронний метод просто засне на час, поки не отримає дані від БД, а основний потік зможе продовжити свою роботу. У синхронному ж додатку, якби код отримання даних знаходився в основному потоці, цей потік просто б блокувався на час отримання даних.

Ключовими для роботи з асинхронними викликами є два ключових слова: async та await, мета яких – спростити написання асинхронного коду. Вони використовуються разом для створення асинхронного методу.

Асинхронний метод володіє наступними ознаками:

- у заголовку методу використовується модифікатор async;
- Метод містить одне або кілька виразів await.

Асинхронний метод, як і звичайний, може використовувати будь-яку кількість параметрів або не використовувати їх взагалі.

Також варто відзначити, що слово async, яке зазначається в ухвалі методу, що не робить автоматично метод асинхронним. Воно лише вказує, що даний метод може містити одне або кілька виразів await.

Справа в тому, що Java сама по собі такої підтримки не має. Для цього довелося створювати прекомпілятор на Java, для того, щоб перевести код, написаний з використанням await в зрозумілий для компілятора Java код.

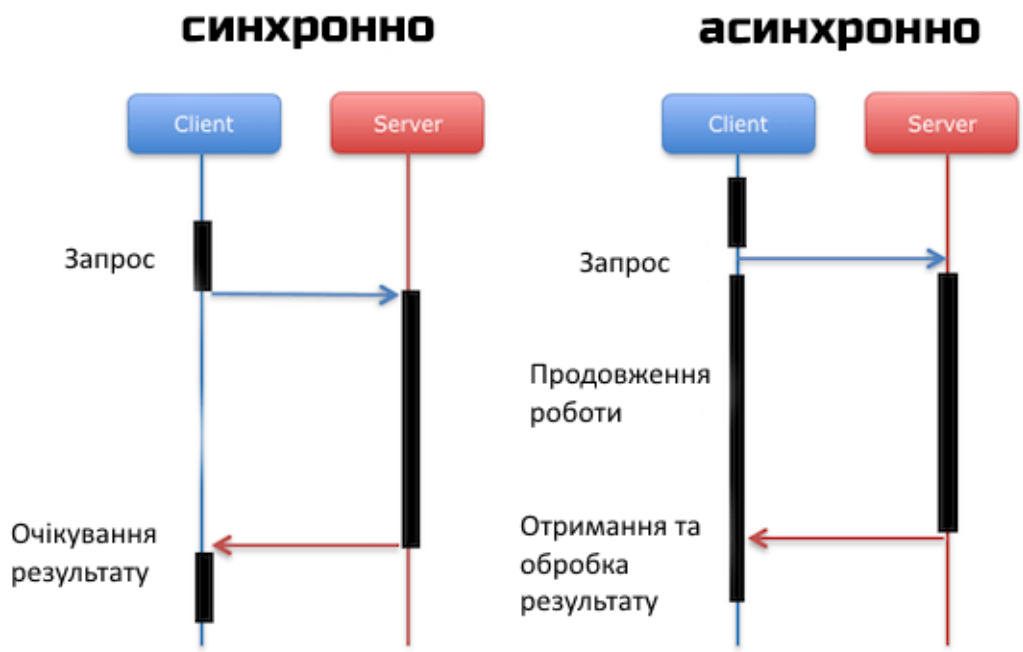

Рисунок 4.5 – Зрівняння синхронних та асинхронних потоків

4.8 ScopeFactory

Особливістю ScopeFactory являється можливість отримання посилання на об’єкти через Scope, та не потрібно клонувати посилання у кожному об’єкті, котрий з ним працює. По-перше, це вирішує питання, зв’язане з нагромадженням коду – тобто 20 класів, що мають посилання один на одного. До того ж, не всі з цих об’єктів оброблюються за робочий потік. По-друге, відбувається економія heap-пам’яті за рахунок стекової.

Додатковим елементом є Dependency Injection. Процес надання зовнішньої залежності програмного компоненту. Є специфічною формою «інверсії управління», коли вона застосовується до управління залежностями. У повній відповідності з принципом єдиною обов'язки об'єкт віддає піклуватися про побудову необхідних йому залежностей зовнішньому, спеціально призначеному для цього спільного механізму.

Взагалі-то, це може використовуватися як шаблон проектування. Незважаючи на те, що ScopeFactory проектувався для рішення проблем з

пам’яттю, він не порушує архітектуру проекту, та навіть покращує її. Якщо заздалегідь передбачити використання ScopeFactory, то можна в багато разів спростити структуру проекту. Цей модуль вже самостійно буде керувати створенням об’єктів, а також не потрібно загромаджувати програмний код, що суперечить поняттю архітектури проекту.

До речі, C# має подібний модуль, але він налаштований на роботу з Web-серверами, тобто новий Scope автоматично створюється при виклику одного з методів контролера, та новий створити не можна. До того ж, їх Dependency Injection не дозволяє створювати об’єкти з вхідними параметрами й при створенні Scope всі зарезервовані класи створюють свої екземпляри. Відповідний модуль ігрового двигуна SvS Ateunane цю проблему вирішує.

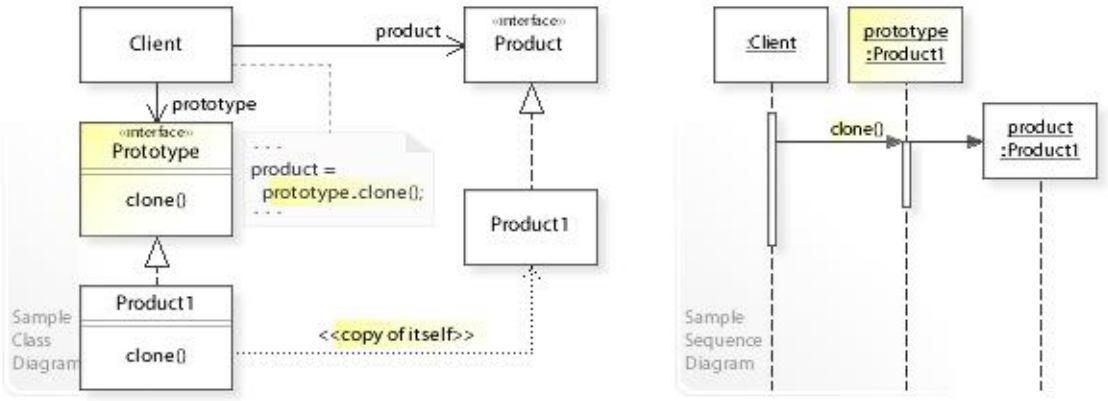

Рисунок 4.6 – Принцип роботи Dependency Injection

Код, що приведений нижче, зображує принцип роботи модулю ScopeFactory.

```
    try{
        ThreadScopeBlock block = new
ThreadScopeBlock(Thread.currentThread(), this);
        CURRENT_SCOPES.push(block);
        V value;
        if (params != null)
            value = (V) constructor.newInstance(params);
        else
```

```
                value = (V) constructor.newInstance
                (getParams(constructor));
            SCOPES.put(value.hashCode(), this);
            values.push(value);
            CURRENT_SCOPES.remove(block);
            return value;
        }
        catch (InvocationTargetException | InstantiationException |
IllegalAccessException exc){
            exc.printStackTrace();
        }
```

4.9 SimpleDataSerializer

Структура даних – це структурна програмна одиниця, що дозволяє зберігати і обробляти безліч однотипних і / або логічно пов'язаних даних в обчислювальній техніці. Для додавання, пошуку, зміни і видалення даних структура даних надає певний набір функцій, з яких складається інтерфейс.

При розробці програмного забезпечення, тобто гри, складність реалізації та якість роботи програм істотно залежать від правильного вибору структур даних. Це розуміння дало початок формальним методам розробки та мов програмування, в яких саме структури даних, а не алгоритми, є наріжним архітектури програмного засобу. Велика частина таких мов володіє певним типом модульності, що дозволяє структурам даних безпечно перевикористовувати в різних додатках. Об'єктно-орієнтовані мови, такі як Java, C # і C ++, є прикладами такого підходу.

Фундаментальними будівельними блоками для більшої частини структур даних є масиви, записи, розмічені об'єднання та посилання. Наприклад, двозв’язний список може бути побудований за допомогою записів та посилань, де кожен запис (вузол) буде зберігати дані та посилання на «лівий» та «правий» вузли.

Структура даних – по суті це контейнер, який зберігає дані в певному макеті. Цей «макет» дозволяє структурі даних бути ефективною в деяких

операціях і неефективною в інших. Структури даних діляться на два типи – лінійні та нелінійні. Лінійні, елементи утворюють послідовність або лінійний список, обхід вузлів лине. Приклади: Масиви. Пов'язаний список, стеки і черги. Нелінійні ж, якщо обхід вузлів нелінійний, а дані не послідовні. Приклад: граф та дерева.

Для ігрового двигуна SvS Ateunane була розроблена структура даних SimpleDataSerializer. У цьому форматі зберігаються усі конфігурації та дані об’єктів, а також стан усіх ігрових об’єктів у даний момент часу – це дає можливість зберігати прогрес гри з найбільш максимальною кількістю параметрів. Вона являється лінійною структурою даних.

У структурі немає вкладеностей, усі елементи йдуть послідовно один за одним, розділяючись роздільником. Якщо самі дані мають цей роздільник, то він екранується другим спеціальним символом. Цей символ також екранується, якщо попадається всередині даних. Зокрема існує ще два контрольні символи – для порожнього рядка та для елементу, що приймає значенню null.

SimpleDataSerializer являється унікальною та має ряд переваг та недоліків порівняно з цілком відомим структурами даних як Json та XML. До переваг входять економія об’єму пам’яті на носії та час на серіалізацією та дисеріалізацією. Так як парсери звичайних форматів використовують рефлексію. По-перше, рефлексія не усюди присутня, а там, де присутня, може мати не повну її підтримку. По-друге, рефлексія – інструмент тяжкий для обробки. Вона працює повільно. Недоліком є гірша людиночитабельність даних, що значно ускладнює ручне редагування даних. Ще одним недоліком, що структура даних є статичною, тобто, коли парсер працює, він заздалегідь повинен знати у що парсити. Це стосується даних, не маючих фіксованого розміру, тобто масиви, списки тощо. Клас, який виступає у ролі даних, повинен реалізувати інтерфейс ISerializable, котрий потребує визначення методів збору даних для та їх встановлення. Подальші версії формату даних підтримують типізацію класів, що дозволяє куди простіше проводити завантаження конфігурацій. Ідея наступна: програміст реалізовує інтерфейс ISerializable. Всередині

цих методів він власноруч встановлює поля, котрі будуть серіалізовуватись, вид, в якому вони буду представлені, та їх порядок.

Далі приведений код, що зображує роботу з SimpleDataSerializer. Перший шматок коду демонструє клас, що наслідує інтерфейс ISerializable, та реалізацію методу по збору параметрів.

```
protected static class SerialTest implements ISerializable{
     public String name;
     private String[] arr;
     @Override public void grabSerializeValues(LList list){
          list.push(name);
          final int COUNT = arr.length;
          list.push(COUNT);
          for (int i = 0; i < COUNT; ++i)
               list.push(arr[i]);
     }
```

Наступний шматок коду зображує метод по встановленню даних зі структури.

```
@Override
public void fromSerializeValues(TypedGoerOver values){
     name = values.nextString();
     final int COUNT = values.nextInt();
     String[] arr = new String[COUNT];
     for (int i = 0; i < COUNT; ++i)
          arr[i] = values.nextString();
     this.arr = arr;
}}
```

Останній шматок демонструє безпосередньо роботу по серіалізації да дисерілізації.

```
TypedSerializer serializer = new TypedSerializer();
String serialized = serializer.serialize(
```

```
          new SerialTest(123, "?|#@This is string", 456.001f,
"1", "2=", "3_"), new SerialTest(78, "?#@|Cu-cu|@#?", 0.999f,
"?", "#", "@"));
     System.out.println(serialized);
     LList<ISerializable> list =
serializer.deserialize(serialized);
     for (
          LVertex<ISerializable> vertex = list.getFirst();
          vertex != null;
          vertex = vertex.next())
               System.out.println(vertex.getItem().toString());
```

4.10 PupetSystem

Візуалізація – один з найбільш важливих розділів в комп'ютерній графіці, та на практиці він тісно пов'язаний з іншими. Зазвичай програмні пакети тривимірного моделювання та анімації включають в себе також і функцію рендеринга. Існують окремі програмні продукти, що виконують рендеринг. Одним із таких продуктів являється модуль двигуна PupetSystem. Реалізація механізму рендеринга завжди ґрунтується на фізичній моделі. Продукція, що обчислення відносяться до тієї чи іншої фізичної або абстрактної моделі. Основні ідеї прості для розуміння, але складні для застосування. Як правило, кінцеве елегантне рішення або алгоритм більш складні та містять в собі комбінацію різних технік.

Векторна графіка – графіка, у якій зображення представлено у вигляді геометричних форм, що дає максимальну точність побудованого зображення. Такий формат картинки легко редагується, масштабується, повертається, деформується, і дозволяє імітувати тривимірність. З недоліків вектора можна відзначити відсутність реалістичності і неможливість використання ефектів. Векторна графіка підходить для малювання креслень і схем, використовується для масштабованих шрифтів, ділової графіки, для елементів брендбуку (логотипи, декоративні візерунки і т.п.), застосовується для створення мультфільмів і різних роликів, а також у пресі (забезпечує високу якість зобра-

ження). В даному випадку, цей вид графіки ідеально підходить до лялькової системи.

Лялькова система або ж PupetSystem – це глобальний модуль двигуна SvS Ateunane, ща націлений на обробку двовимірних костей. Як і в 3D моделюванні, для спрайтового об’єкта можна застосувати кістки, що будуть керувати рухом цього об’єкта. На кості може впливати фізична сила, тобто кості можуть рухатись під впливом сил. Для роботи з кістками для двигуна був розроблений міні редактор, який дозволяє зчіплювати спрайти та прив’язувати кістки, формуючи при цьому рухомий ігровий об’єкт. Перевагами цієї системи являється гнучкість по відношенню до фізики, тобто відтепер фізика може впливати на рух окремих елементів ігрового об’єкта. Також, тепер не потрібний величезний атлас анімацій великий багатофункціональних об’єктів. Відтепер в атласі присутні лише базові деталі ігрового об’єкта, а анімації опрацьовуються безпосередньо самим двигуном.

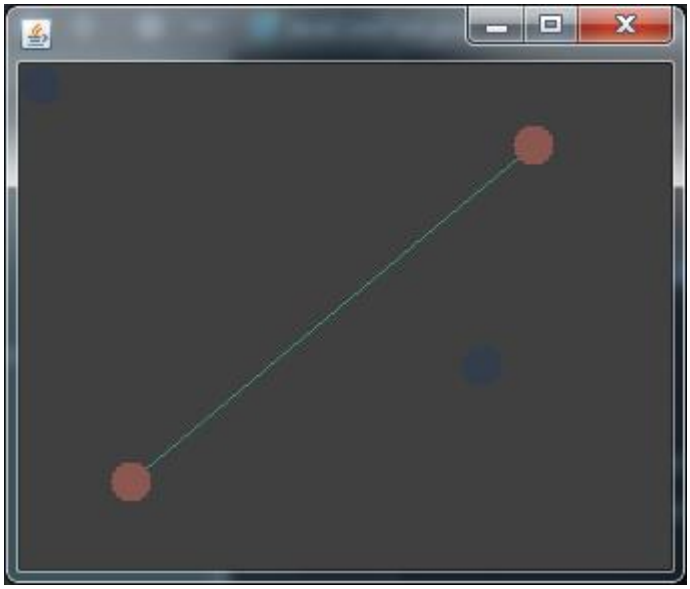

Рисунок 4.7 – Елементарна кістка

Рис 4.7 Зображає елементарну кістку, що має безпосередньо одну кістку та два суглоби. Рухаючи один суглоб, інший повинен підлаштовуватись під рух першого, зберігаючи при цьому оригінальну відстань між ними.

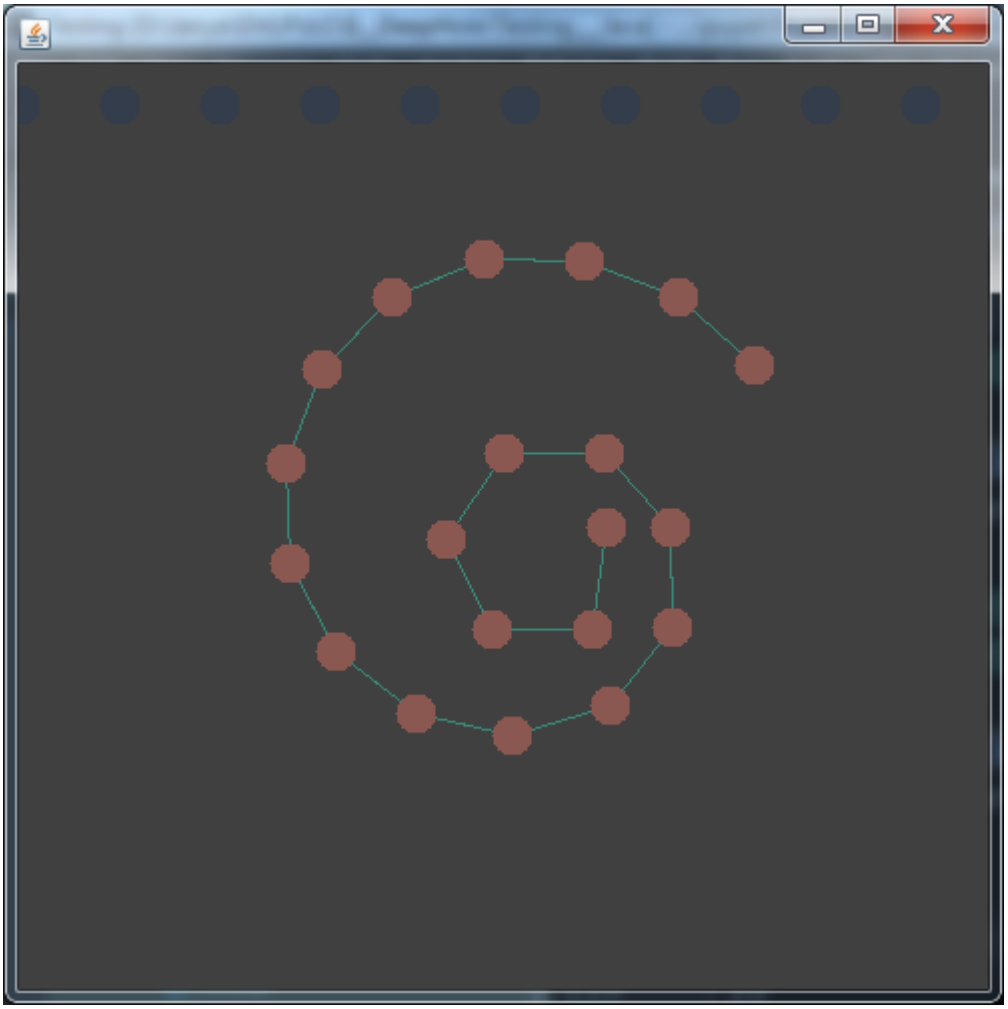

Рисунок 4.8 – Ланцюгова модель ляльки

На рисунку 4.8 зображено більш складну так звану ланцюгову модель. В цій моделі всі суглоби з’єднані в ланцюг один за одним. У даній моделі всі суглоби мають однакову відстань між сусідніми суглобами. На рисунку 4.9 зображена морфована версія цього ланцюга, тобто деякі суглоби змістились та таким чином видно як вся структура зі суглобів видозмінилась.

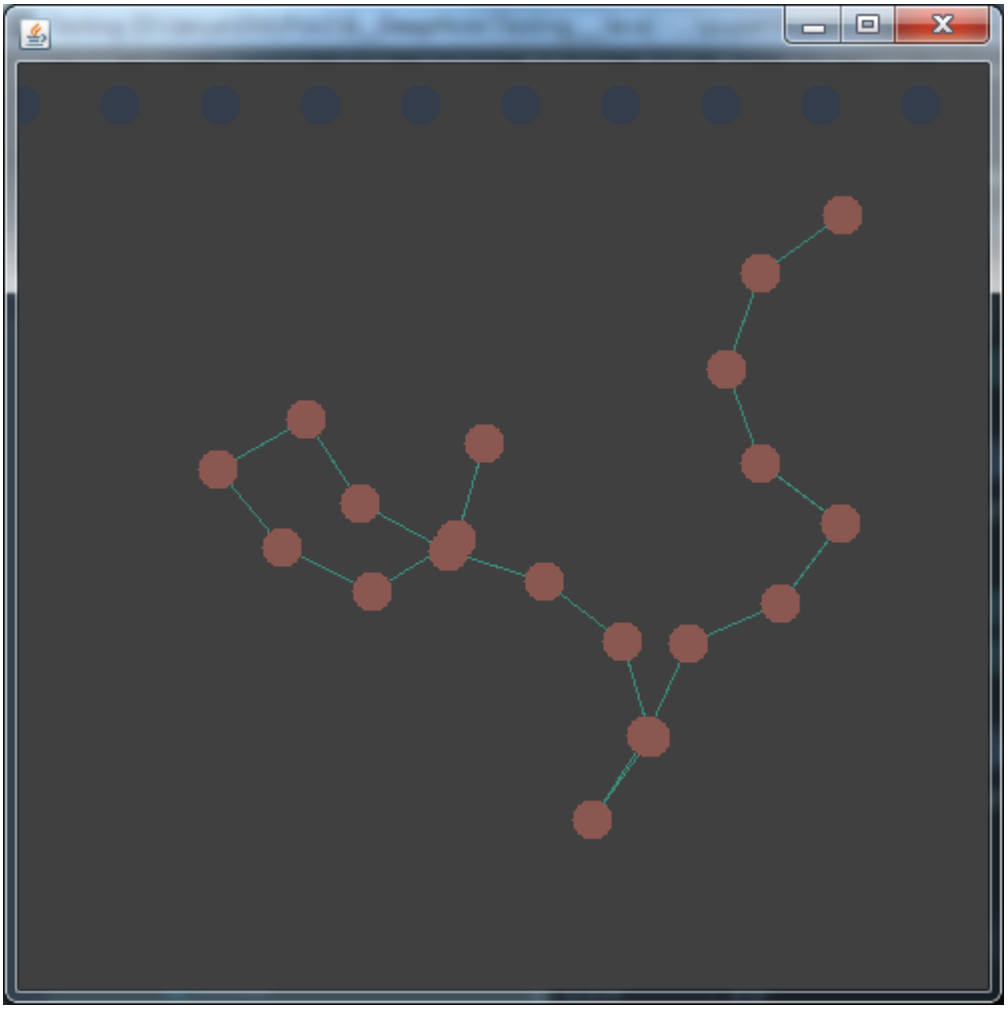

Рисунок 4.9 – Морфована версія ланцюгової моделі ляльки

Однією з найбільш складних моделей являється людиноподібна. У ній один суглоб може з’єднувати декілька костей, що значно ускладнює логіку роботи коректування положення суглобу. На рисунку 4.10 зображена скелетна модель людиноподібної істоти, яка складається з 10 костей та 11 суглобів.

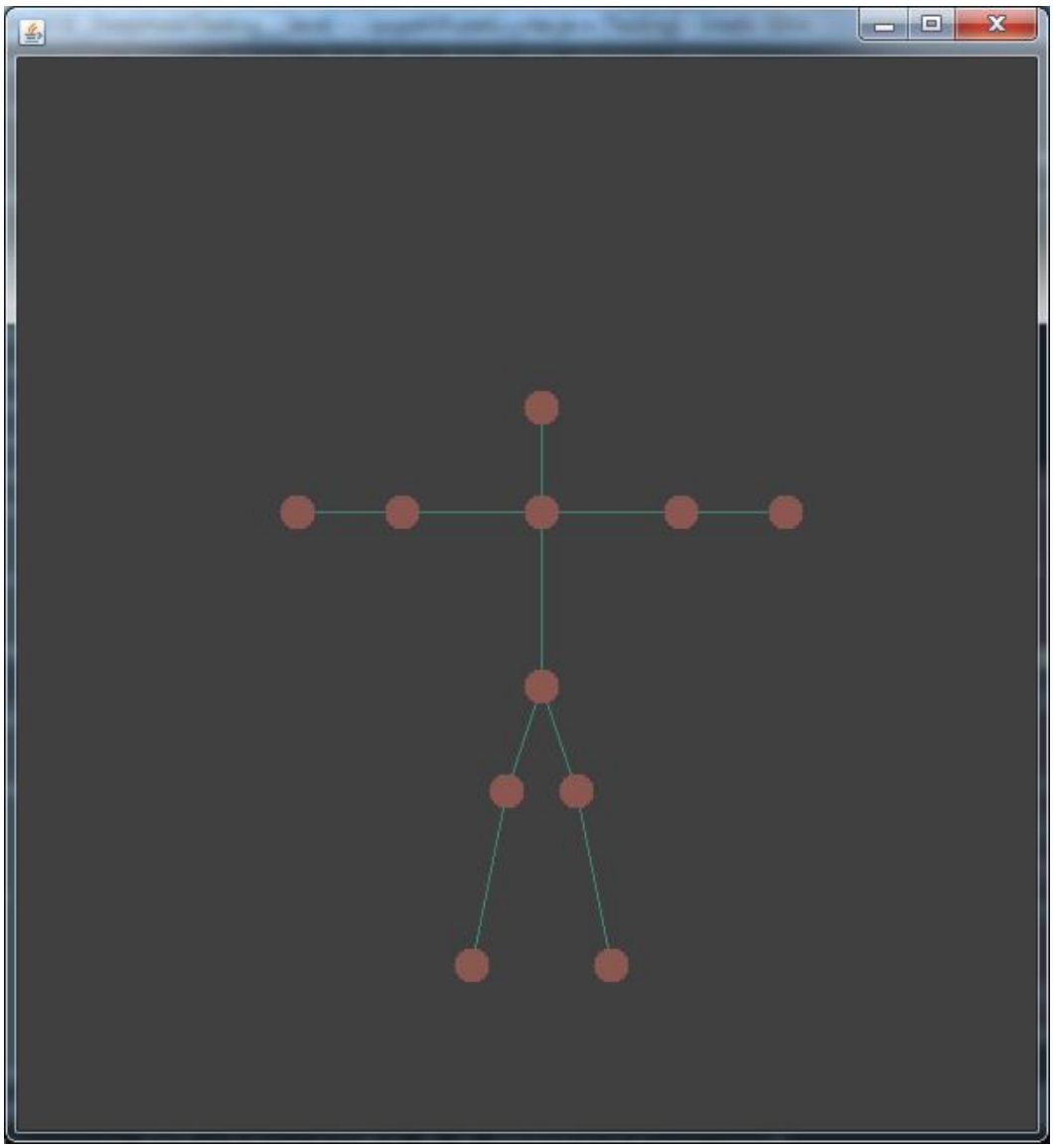

Рисунок 4.10 – Людиноподібна модель ляльки

Рисунок 4.11 зображає як попередня модель спотворюється при русі двох опорних суглобів. Усі кістки рекурсивно обраховуються та корегують своє положення.

На рисунку 4.12 зображено атлас структурних елементів ігрового об’єкта, тобто його спрайтів. Атлас згенеровано за допомогою SceneEditor. Подалі, ці спрайти будуть прив’язані до лялькової системи.

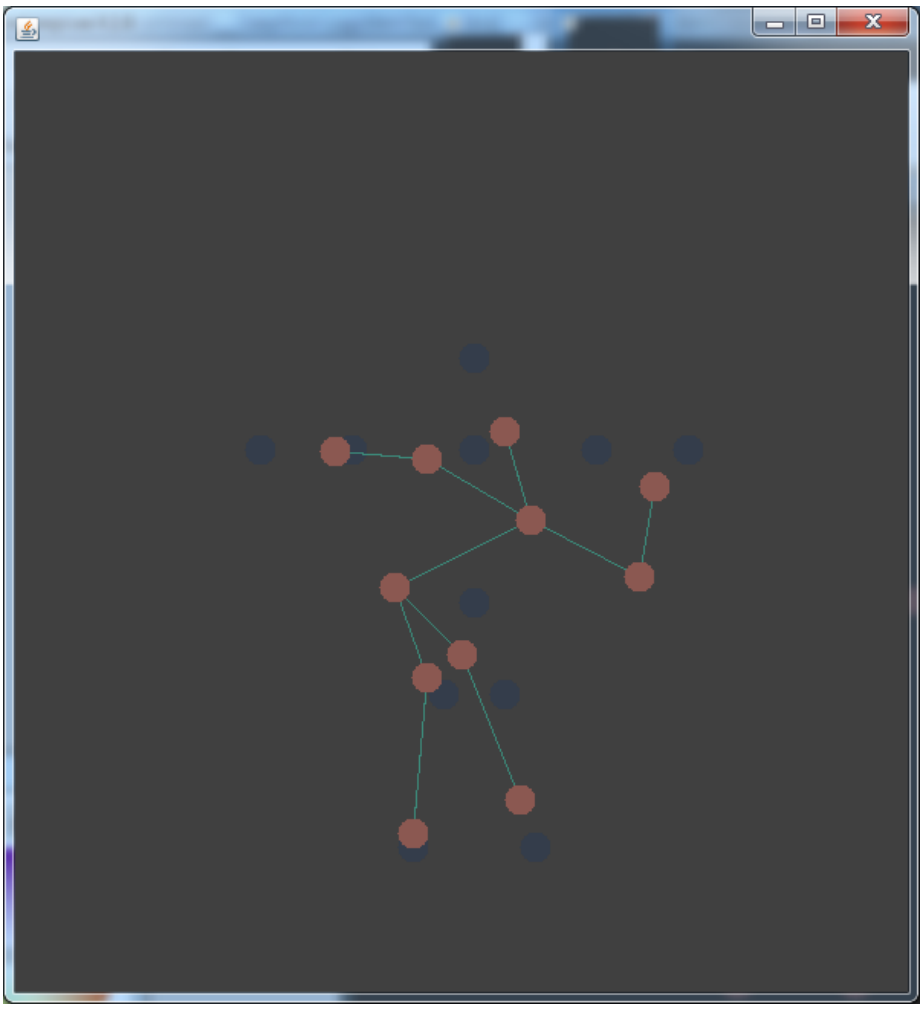

Рисунок 4.11 – Видозмінена версія людиноподібної моделі ляльки

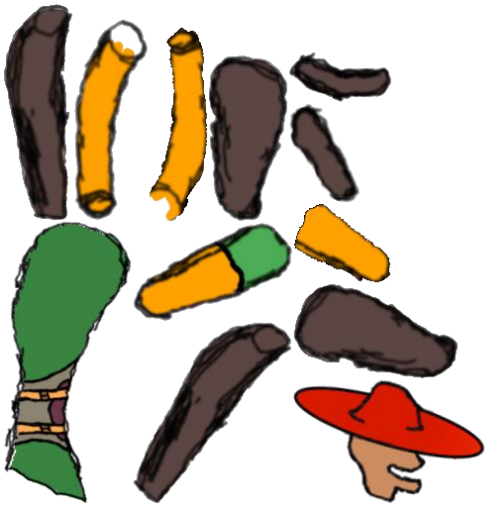

Рисунок 4.12 – Атлас спрайтів ігрового об’єкта

На рисунку 4.13 зображено повноцінний ігровий об’єкт, тобто на спрайти ігрового об’єкта накладені кістки. Відтепер ця структура може рухатися, видозмінюватись під час анімацій та бути під впливом різних фізичних сил.

Рисунок 4.13 – Повноцінний ігровий об’єкт

ВИСНОВКИ

Зараз у світі одним з найскладніших, найрозповсюджених та найприбутковіших проектів є саме комп'ютерна гра. Ігри розроблюють на всіх платформах, яких тільки це можливо. Такий вид ПО проникає всюди, й попит на нього не падає, та навряд чи буде падати.

Під час роботи було виконано:

– проведений повний аналіз предметної області;
– розглянуті аналоги, проаналізовано їх переваги й недоліки;
– розглянута мова UML, зокрема діаграма класів та діаграма діяльностей.

Результатом практичної частини дипломної роботи є:

– проектування алгоритмів реєстрації та захисту проекту за допомогою діаграм діяльностей;
– проектування структури проекту на основі діаграми класів;
– створення ігрового двигуна;
– реалізовано основні ігрові об'єкти, їх характеристики та взаємозв'язок;
– створений власний музичний арт.

В рамках подальшого розвитку даної дипломної роботи планується:

– зареєструвати та продавати гру на Google Play Market, AppStore та Steam Greenlight;
– доповнювати графічним та звуковим контентом задля урізноманітнення відеогри.

ПЕРЕЛІК ПОСИЛАНЬ

1. GameMaker Studio 2 [Електронний ресурс]. – Режим доступу: https://www.econdude.pw/2017/09/gamemaker-studio-2-stoit-li-togo-pljusy-i-minusy.html
2. Движок Unity – особенности [Електронний ресурс]. – Режим доступу: https://cubiq.ru/dvizhok-unity/
3. UNITY VS UNREAL ENGINE 4 [Електронний ресурс]. – Режим доступу: https://visschool.ru/blog/3d-viz-blog/unity-vs-unreal-engine-4-bitva-titanov-27/
4. Довідник з програмування [Електронний ресурс]. – Режим доступу: http://codingcraft.ru/glossary.php
5. Підтримка розробника на Oracle [Електронний ресурс]. – Режим доступу: https://www.oracle.com/
6. A Beginner's Guide To Making Your First Video Game [Електронний ресурс]. – Режим доступу: https://kotaku.com/
7. Засоби створення відеоігор [Електронний ресурс]. – Режим доступу: https://biblprog.org.ua/
8. Розробка програмного забезпечення [Електронний ресурс]. – Режим доступу: http://aphd.ua/
9. Моделювання на UML [Електронний ресурс]. – Режим доступу: http://book.uml3.ru/
10. Quizful [Електронний ресурс]. – Режим доступу: http://www.quizful.net/
11. Хабрахабр [Електронний ресурс]. – Режим доступу: https://habr.com
12. FLS Student Handbooks [Електронний ресурс]. – Режим доступу: http://fls-handbooks.org/student/tutorial-guides/first-year-tutorial-guide/
13. Введення в UML 2.0 [Електронний ресурс]. – Режим доступу: http://bourabai.kz/dbt/uml/index.htm

Д О Д А Т К И

ДОДАТОК А
ДІАГРАМИ КЛАСІВ ТА ПРОЦЕСІВ

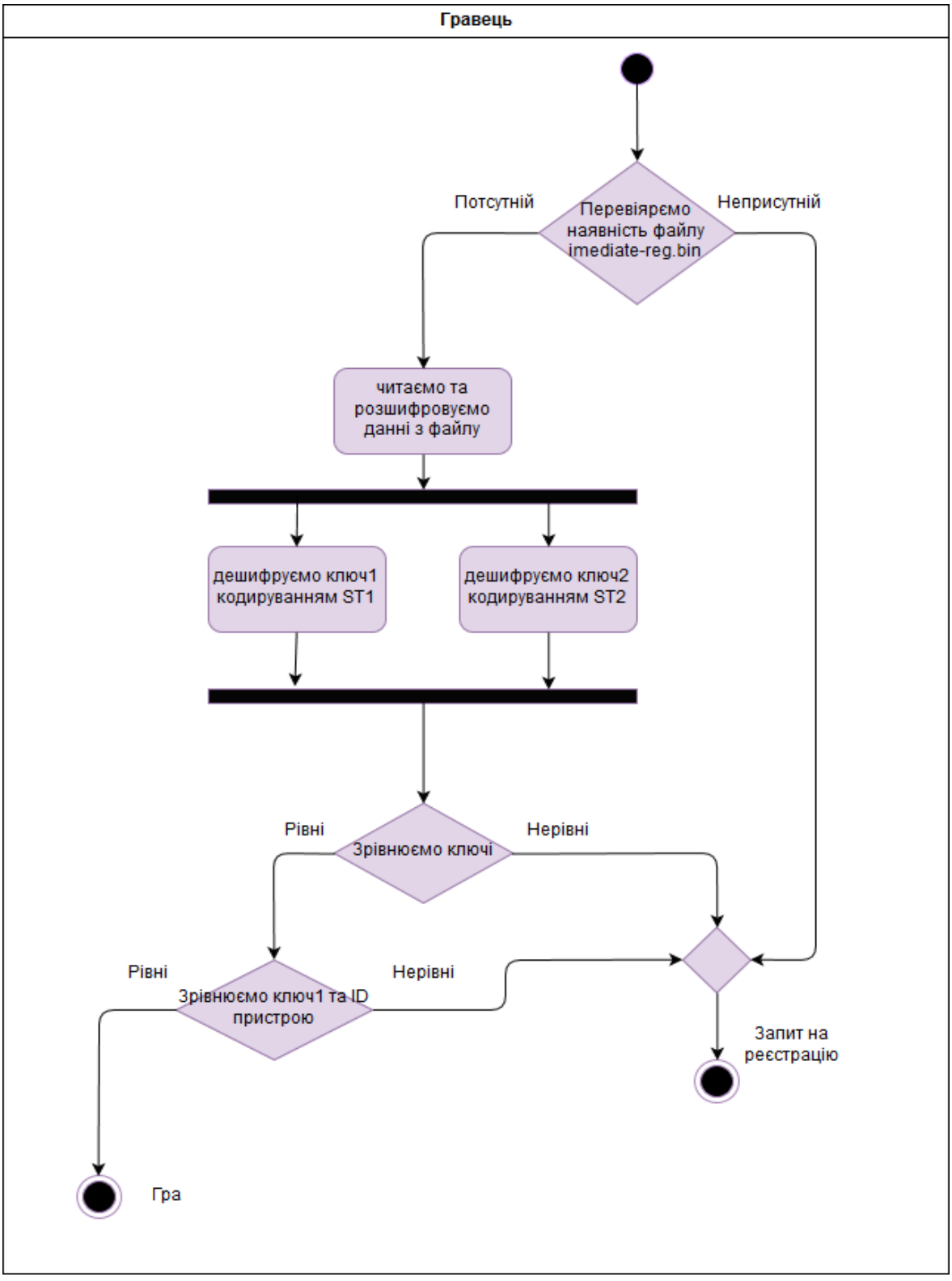

Рисунок А.1 – «Валідація реєстрації»

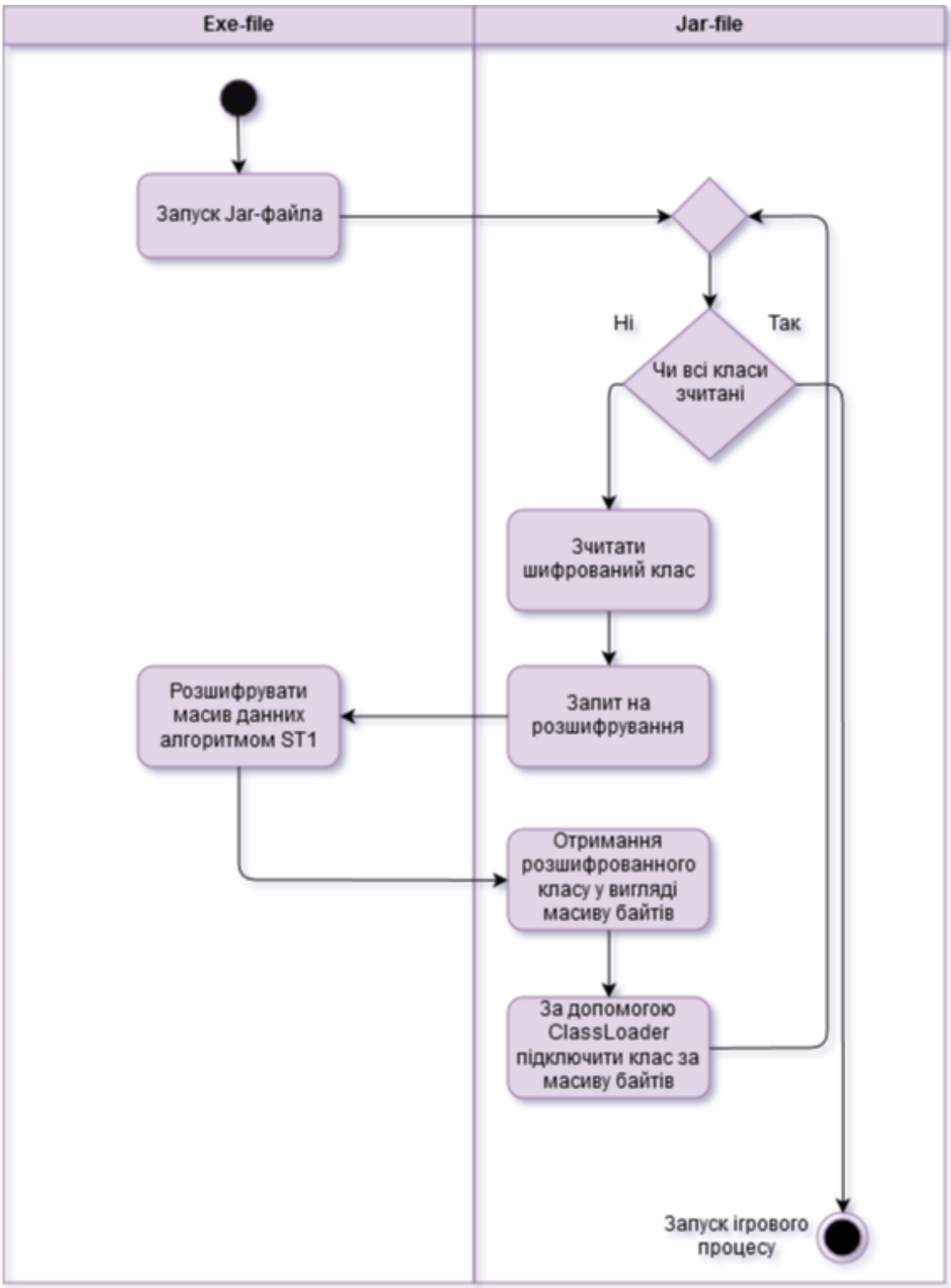

Рисунок А.2 – «Захист від перегляду вихідного коду»

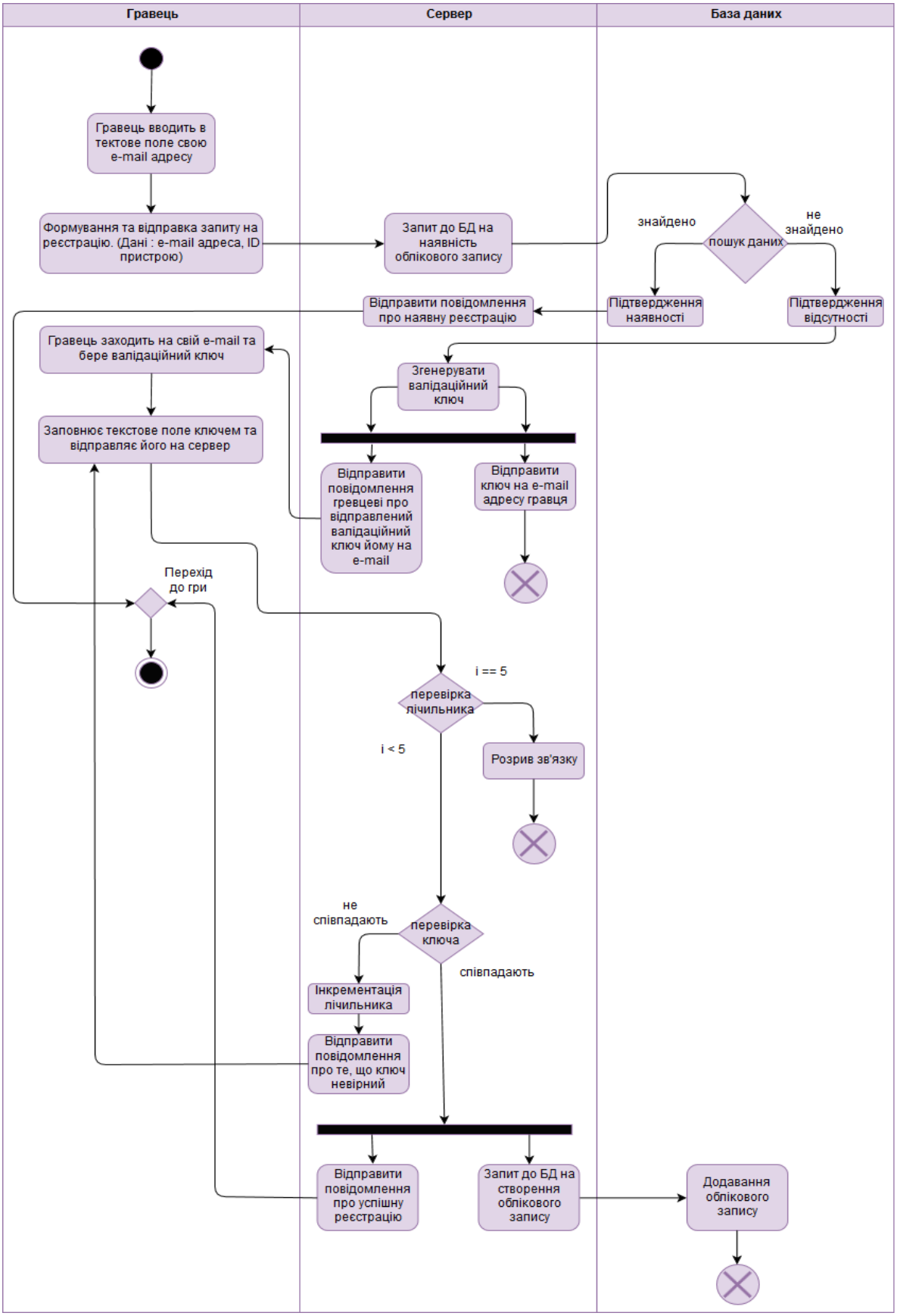

Рисунок А.3 – «Процес реєстрації»

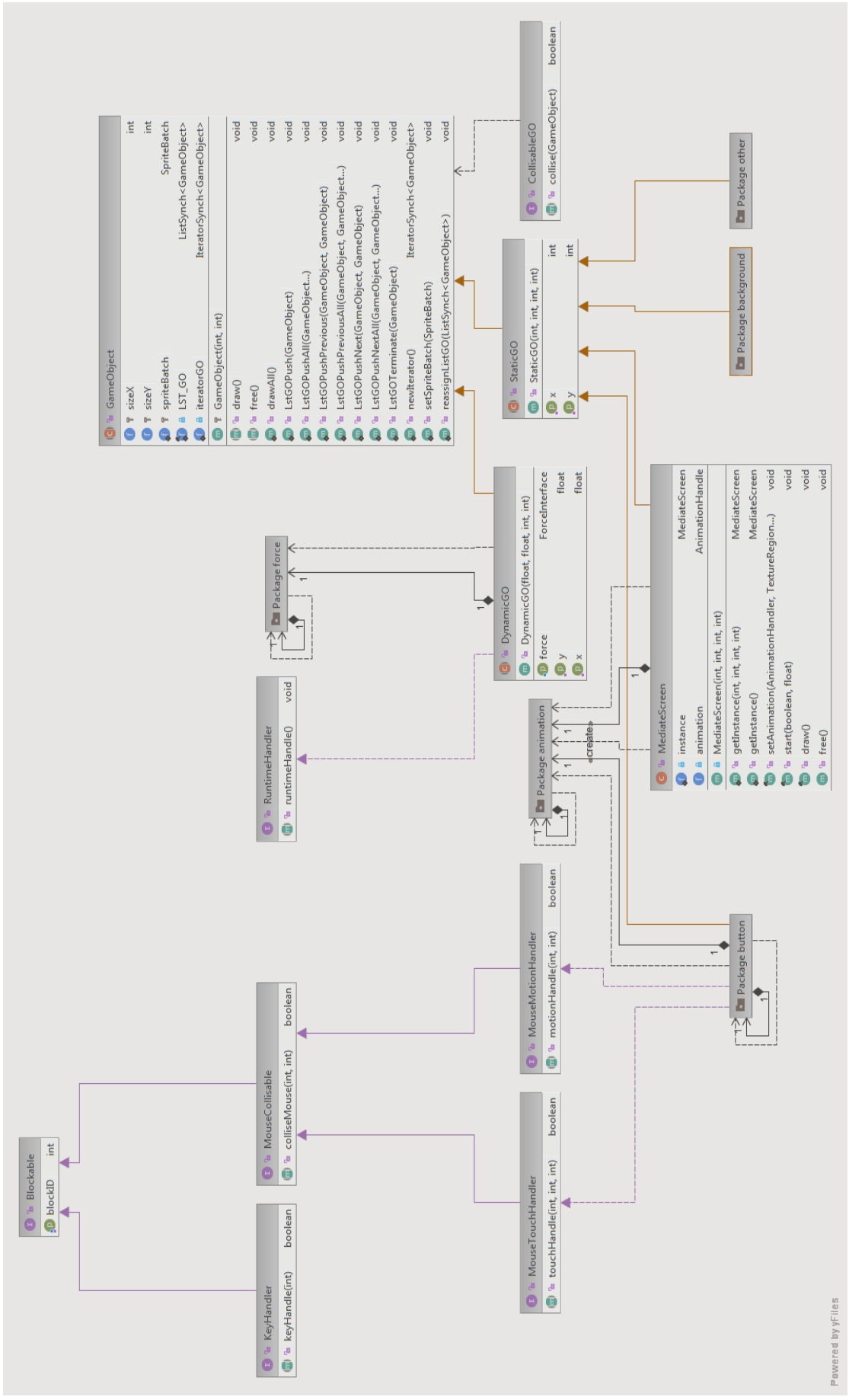

Рисунок А.4 – «Структура ігрового двигуна SvSAteunane»

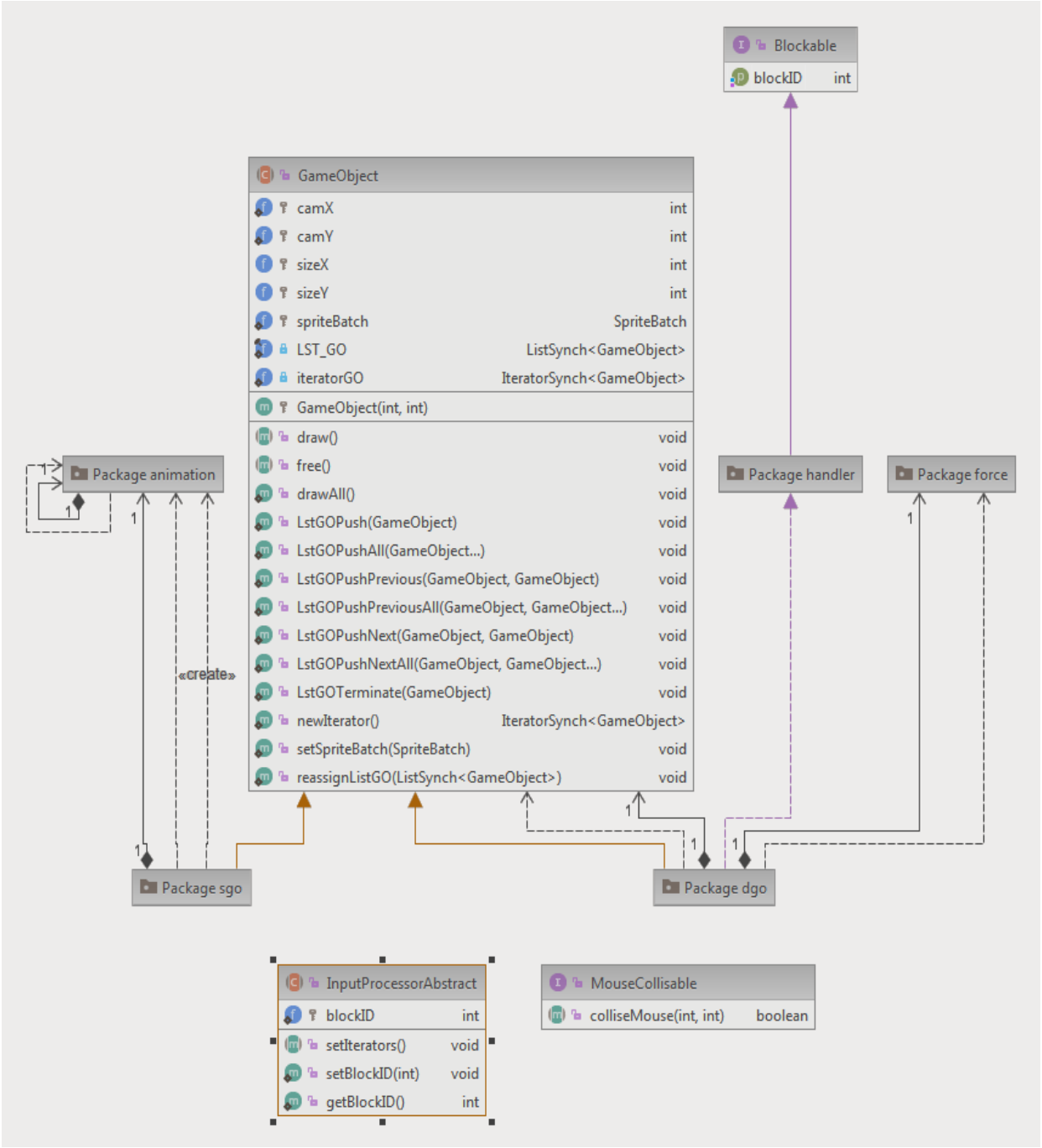

Рисунок А.5 – «Структура пакету ігрових об’єктів»

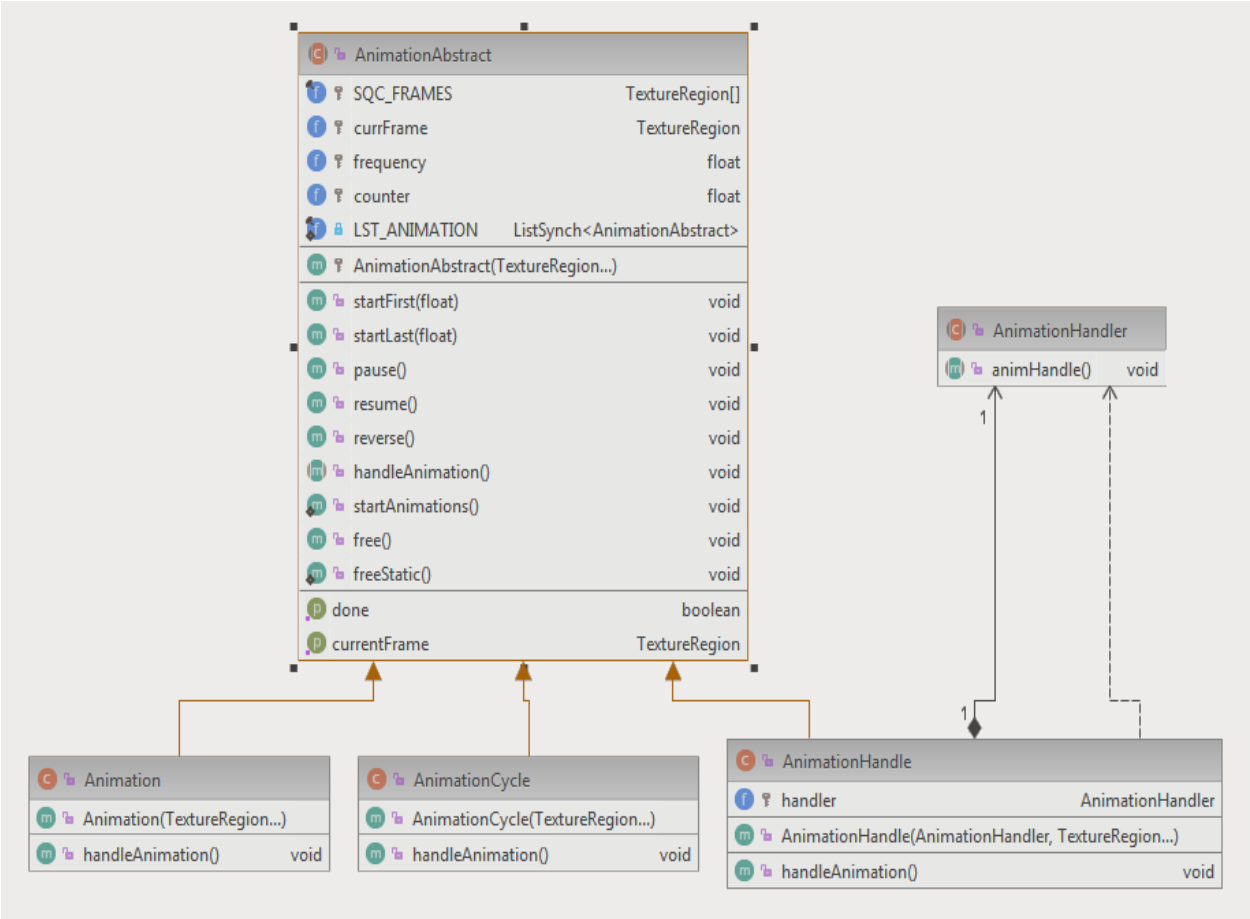

Рисунок А.6– «Структура пакету анімацій»

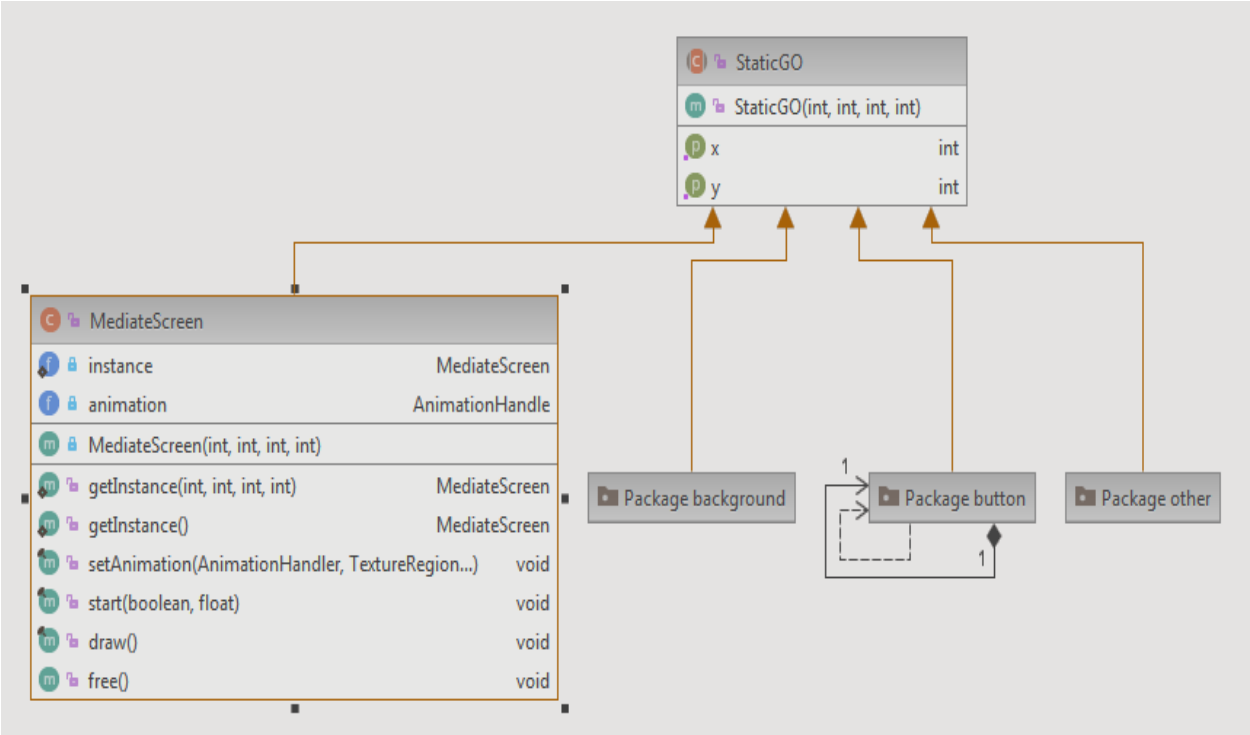

Рисунок А.7 – «Структура пакету статичних об’єктів»

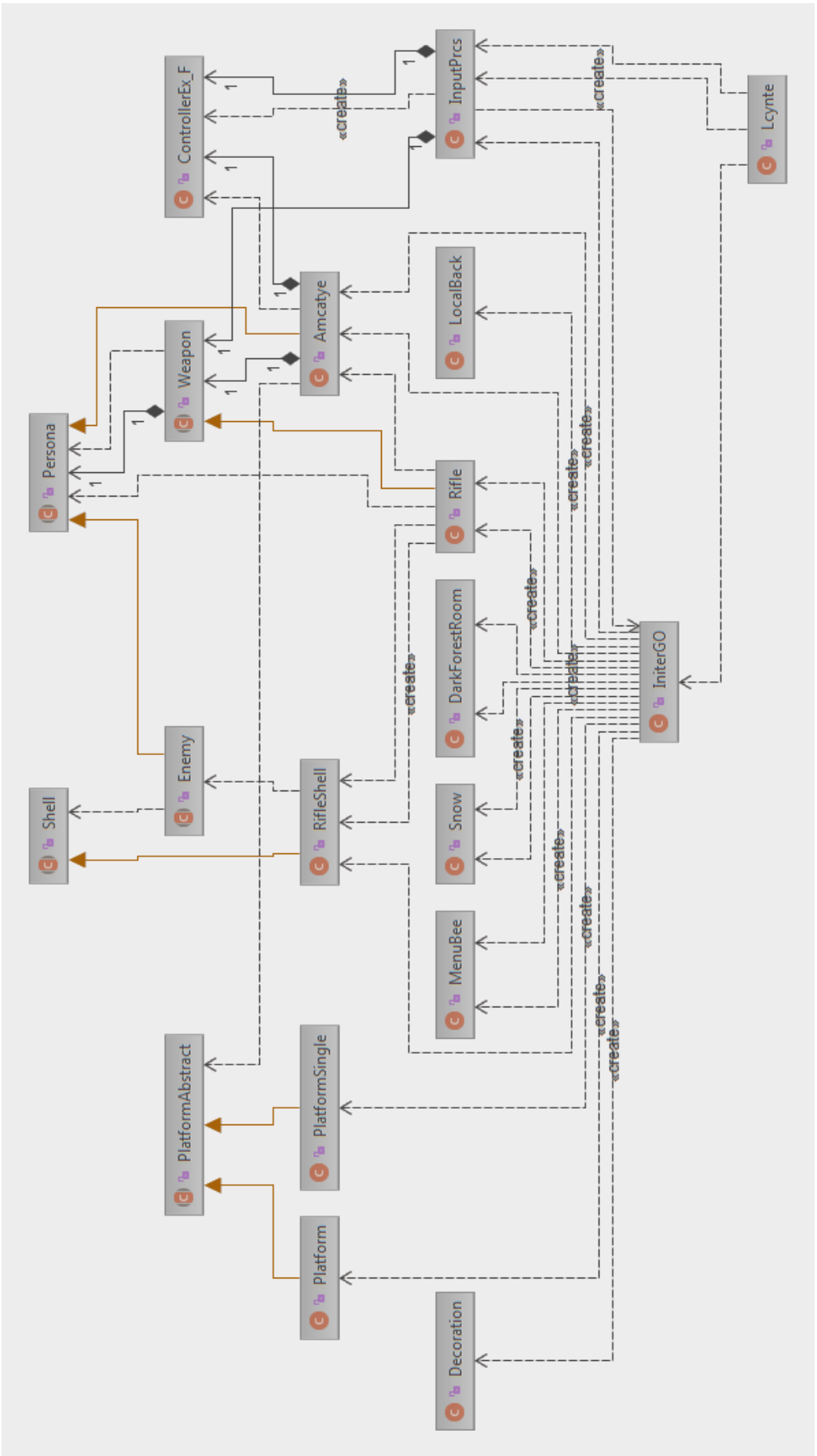

Рисунок А.8 – «Структура програмного ядра відеогри»

ДОДАТОК Б
ПРОГРАМНИЙ КОД МОДУЛЯ ІНІЦІАЛІЗАЦІЇ ІГРОВИХ ОБ'ЄКТІВ ГРИ ТА НАЛАШТУВАННЯ ОБРОБЛЮЮЧИХ ПРОЦЕСІВ

```
static final byte INIT_MAIN_MENU = 0;
static final byte INIT_GAME_PROC = 1;
static final byte INIT_PAUSE = 2;
static final byte INIT_NONE = 3;
static final byte MODE_MENU = 0;
static final byte MODE_GAME_PROC = 1;
static final byte MODE_PAUSE = 2;

static Cursor cursor;

public static final Random RANDOM = new Random();

static byte gameProcMusicCounter = 0;

static final float MEDIATE_SCREEN_ANIM_FREQ = 0.03f;
static void init()
{     AnimationHandler handler = new AnimationHandler()
      {
            @Override
            public void animHandle()
            {
                  Gdx.app.postRunnable(new Runnable()
                  {
                        @Override
                        public void run()
                        {
                              GAME_LIST.LstGOTerminateSynch(mediateScreen);

                              switch (mode)
                              {
                                    case INIT_GAME_PROC :
                                          initGameProcess();
                                    break;

                                    case INIT_MAIN_MENU :
```

```
                                        initMainMenu();
                                    break;

                                    case INIT_PAUSE :
                                        initPause();
                                    break;

                                    case INIT_NONE :
                                        InputPrcs.setBlockID(0);
                                    }
                                }
                            });
                        }
                };
        Cursor cursor = new Cursor
                (
                        "res/nullCursor.png",
                        new TextureRegion
                        (
                                new Texture(Gdx.files.internal("res/cursor_.png"))
                        ),
                        0, 0, 60, 60
                );
        IniterGO.cursor = cursor;
        InputPrcs.cursor = cursor;
        AnimationAbstract.startAnimations();

        initMainMenu();
}

private static void initGameProcess()
{
        camera.reset();
        gameMode = MODE_GAME_PROC;
        switch (gameProcMusicCounter)
        {
                case 0 :
                        music =
Gdx.audio.newMusic(Gdx.files.internal("res/sound/music/eminem_-
_takingMyBallsTimed.mp3"));
                break;
```

```
            default :
                  music =
Gdx.audio.newMusic(Gdx.files.internal("res/sound/music/eminem_-
_havingTheRelapseTimed.mp3"));
            gameProcMusicCounter = -1;
      }



      //==:  Room
            Texture textureDR1 = new Tex-
ture(Gdx.files.internal("res/room/darkForest/darkForest_L1.png"));
            Texture textureDR2 = new Tex-
ture(Gdx.files.internal("res/room/darkForest/darkForest_L2.png"));
            Texture textureDR3 = new Tex-
ture(Gdx.files.internal("res/room/darkForest/darkForest_L3.png"));
            Texture textureDR4 = new Tex-
ture(Gdx.files.internal("res/room/darkForest/darkForest_L4.png"));
            DarkForestRoom room = new DarkForestRoom
                  (
                        textureDR1, textureDR2, textureDR3, textureDR4,
                        0, 0, getRelW(1366), getRelH(768)
                  );
            gameList.push(room);

            Texture SNWTexture = new Tex-
ture(Gdx.files.internal("res/image/_atlasSnow2.png"));
            Snow.setAtlas(SNWTexture);
            Snow.sizeMin = 8;
            Snow.sizeMax = 30;
            Snow.forceMax = 0.7f;
            Snow.forceMin = 0.1f;
            Snow.arideStepMax = rads(15f);
            Snow.arideStepMin = rads(1f);
            final int LEN_SNW_U = 25;
            for (int i = 0; i < LEN_SNW_U; ++i)
                  gameList.push(new Snow());

      //==:  Platform
            generatePlatforms(gameList, PltfTexture, LBTexture, DecorTexture);
```

```
	//==:  Amcatye
		final float MOVE = 0.01f;
		Amcatye amcatye = new Amcatye
			(
				getRelW(170 >> 1), getRelH(230 >> 1),
				getRelH(119), InputPrcs.controller
			);
		amcatye.cursor = cursor;
		Texture AMRTexture = new Tex-
ture("res/animation/right/_atlasAdeptMove1.png");
		amcatye.setAnimMoveRight
			(
				new AnimationCycle
				(
					getRegions(AMRTexture, 170, 230,12)
				),
				MOVE
			);
		amcatye.stayRight = getRegions(AMRTexture, 170, 230, 170*12, 1)[0];
		gameList.push(amcatye);

		InputPrcs.weapon = rifle;
		gameList.push(rifle);

		RifleShell.texture = new TextureRegion(new Tex-
ture("res/image/rifleShell.png"));

		gameList.push(cursor);

		final MediateScreen MEDIATE_SCREEN = MediateScreen.getInstance();
		mode = INIT_NONE;
```