МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет водного господарства та природокористування
Навчально-науковий інститут автоматики, кібернетики та обчислювальної техніки
Кафедра комп'ютерних технологій та економічної кібернетики

Допущено до захисту:

Завідувач кафедри

_____ д. е. н., проф. П. М. Грицюк

«_____» _____ 2021 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня «бакалавр»
за освітньо-професійною програмою «Інформаційні системи і технології»
спеціальності 126 «Інформаційні системи та технології»

на тему: «**Платформа для розробки 3D-додатків**»

Виконав:
здобувач вищої освіти 4 курсу, групи ІСТ-41
Живий Ярослав Віталійович

Керівник:
канд. техн. наук, доцент Гладка О. М.

Рецензент:
канд.ф.-м. наук., доцент Карпович І. М.

Рівне – 2021

Національний університет водного господарства та природокористування
ННІ автоматики, кібернетики та обчислювальної техніки
Кафедра комп'ютерних технологій та економічної кібернетики

Освітньо-кваліфікаційний рівень – **бакалавр**
за освітньо-професійною програмою **«Інформаційні системи і технології»**
спеціальність **126 «Інформаційні системи та технології»**

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ д. е. н., проф. П. М. Грицюк

«_____» _____ 2021 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Живому Ярославу Віталійовичу
(прізвище, ім'я, по-батькові)

1. Тема роботи Платформа для розробки 3D-додатків

керівник роботи: Гладка Олена Миколаївна, канд. техн. наук, доцент
(прізвище, ім'я, по-батькові, науковий ступінь, вчене звання)

затверджена наказом по університету від "13" квітня 2021 р. С № 330

2. Термін здачі студентом закінченої роботи "11" червня 2021 р.

3. Вихідні дані до роботи розробити програму, що є платформою для створення програмних застосувань, які призначені для роботи з 3D-графікою; дослідити основні властивості (колір, освітленість, переміщення, розтягування, нахил, нормалізація тощо) 3D-об'єктів та способи відворення (візуалізації) цих властивостей.

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити) вступ; аналіз предметної області; вибір та обґрунтування засобів розробки програмного продукту; розробка і опис створеної платформи.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Презентація за матеріалами бакалаврської роботи

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
І	Гладка О. М.		

7. Дата видачі завдання “ 12 ” жовтня 2020 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Аналіз об’єкта дослідження, виявлення існуючих проблем	12.10.20 – 30.10.20	
2.	Аналіз існуючих інформаційних методів (технологій) вирішення проблеми	2.11.20 – 20.11.20	
3.	Вибір та обґрунтування засобів розробки програмного продукту	23.11.20 – 11.12.20	
4.	Проектування, розробка та реалізація програмної платформи	14.12.20 – 30.04.21	
5.	Підготовка пояснювальної записки	5.04.21 – 21.05.21	
6.	Підготовка презентації роботи	24.05.21 – 11.06.21	
7.	Відгук керівника, рецензування роботи, перевірка на плагіат	11.06.21 – 21.06.21	

Студент _____ ***Я. В. Живий***
(підпис) (прізвище і ініціали)

Керівник кваліфікаційної роботи

_____ ***О. М. Гладка***

(підпис) (прізвище і ініціали)

Анотація

УДК

Живий Я. В. / Платформа для розробки 3D-додатків / Дипломна робота / м. Рівне: НУВГП, 2021. – 98 с. Українською мовою.

Ілюстрацій 47, таблиць 17.

Дипломна робота присвячена розробці платформи за допомогою якої розробник буде мати можливість створити 3D додаток використовуючи API платформи разом із графічним та фізичним рушіями для 3D візуалізації та інтерактивності.

В роботі описуються технології в якості основ для графіки та фізичного рушія та інших інструментів платформи.

В процесі проектування та створення платформи були проаналізовані існуючі аналоги та були створені вимоги та завдання відповідно до яких необхідно розробити платформу. Також були розроблені діаграми для пояснення роботи системних компонентів платформи, графічного та фізичного рушіїв, і системи ECS. Крім цього було досліджено та розроблено алгоритм GJK фізичного рушія задля точнішого визначення точок зіткнення фізичних тіл.

Створена платформа призначена для розробників для проектування та розробки програмної реалізації за допомогою набору програмних засобів, інструментів та графічного і фізичного рушіїв.

Annotation

UDC

Zhyvyi Y. V. / Platform for development of 3D-applications / Thesis / Rivne: NUWE, 2021. - 98 p. In Ukrainian.

Illustrations 47, tables 17.

The thesis is devoted to the development of a platform through which the developer will be able to create a 3D application using the platform API together with graphical and physical engines for 3D visualization and interactivity.

The paper describes technologies as a basis for graphics and physics engine and other tools of the platform.

In the process of designing and creating the platform, the existing analogues were analyzed and the requirements and tasks according to which it is necessary to develop the platform were created. Diagrams have also been developed to explain the operation of the system components of the platform, the graphics and physics engines, and the ECS system. In addition, the GJK algorithm of the physics engine was researched and developed in order to more accurately determine the points of collision of physical bodies.

The created platform is intended for developers to design and develop software using a set of platform’s tools with graphics and physics engines.

ЗМІСТ

ВСТУП

Об’єктом вивчення розглядається процес та результат створення програмних систем низького рівня. Тобто тих систем які мають якомога менше абстракції між програмною частиною та компонентами комп’ютера.

Головною відмінністю системного програмування в порівнянні з прикладним програмуванням є те, що прикладне програмне забезпечення призначене для кінцевих користувачів (наприклад, текстові процесори, графічні редактори), тоді як результатом системного програмування є програми, які обслуговують апаратне забезпечення або операційну систему (наприклад, дефрагментація диска, графічні рушії, індустріальна автоматизація) що обумовлює значну залежності такого типу ПЗ від апаратної частини. Слід зазначити, що «звичайні» прикладні програми можуть використовувати у своїй роботі фрагменти коду, характерні для системних програм, і навпаки; тому чіткої межі між прикладним та системним програмуванням немає.

Оскільки різні операційні системи відрізняються як внутрішньою архітектурою, так і способами взаємодії з апаратним та програмним забезпеченням, то принципи системного програмування для різних ОС є відмінними. Тому розробка прикладних програм, які здійснюватимуть одні і ті ж дії на різних ОС, може суттєво відрізнятися.

Актуальність теми полягає у створенні ефективних та швидких рішень з можливостями 3D та 2D графіки із інтеграцією фізичного рушія для візуалізації, інтерактивної симуляції або розваг.

Предметами вивчення розглядаються парадигма програмування “Інформаційно-орієнтовний дизайн” (Data-oriented design, DOD), графічний інтерфейс програмування OpenGL, система “Сутність - компонент” (Entity Component System, ECS), імплементація та інтеграція фізичного рушія.

Завдання:

- Дослідити предметну галузь, а саме створення та архітектура програмних систем середньо-низького рівня
- Провести аналіз існуючих рішень та їх можливості для досягнення мети
- Розробити функціональні вимоги на основі існуючих рішень
- Розробити обробники графіки та фізики
- Представити структуру програмного застосунку
- Описати взаємодію модулів та компонентів
- Продемонструвати приклад роботи ПЗ

Теоретичною основою виконання бакалаврської роботи були онлайн-довідники, документації, статті та стандарти написання і проектування ПЗ.

Робота складається з вступу, трьох розділів, висновків, списку використаних джерел і додатків.

РОЗДІЛ 1. ХАРАКТЕРИСТИКА ТА АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Характеристика предметної галузі та об’єкта дослідження

У багатьох випадках графічні рушії надають набір інструментів візуальної розробки на додаток до програмних компонентів, що використовуються багаторазово. Ці інструменти, як правило, надаються в інтегрованому середовищі розробки, що дозволяє спростити швидкий розвиток програмних продуктів. Розробники ігрових механізмів часто намагаються запобігти потребам реалізаторів, розробляючи надійні програмні комплекти, що містять багато елементів, які розробнику ігор може знадобитися для побудови гри. Більшість наборів графічних рушіїв надають такі зручності, що полегшують розвиток, такі як графіка, звук, фізика та функції штучного інтелекту (ШІ).

Ці графічні рушії іноді називають "проміжним програмним забезпеченням", оскільки, як і в діловому сенсі цього терміну, вони забезпечують гнучку та багаторазову програмну платформу, яка забезпечує всі основні функціональні можливості, необхідні безпосередньо з коробки, для розробки ігрового додатку при одночасному зменшенні витрат , складність та час виходу на ринок – усі найважливіші фактори у висококонкурентній галузі відеоігор. Станом на 2001 р. Gamebryo, JMonkeyEngine та RenderWare широко використовувались проміжні програми такого типу. Як і інші типи проміжного програмного забезпечення, графічні рушії, як правило, забезпечують абстракцію платформи, дозволяючи одному і тому ж програмному застосунку працювати на різних платформах (включаючи ігрові консолі та персональні комп’ютери) з невеликими, якщо такі є, змінами, внесеними у вихідний код ПЗ.

Часто програмісти розробляють графічні рушії з архітектурою на основі компонентів, яка дозволяє замінити або розширити конкретні системи в двигуні на більш спеціалізовані (і часто дорожчі) компоненти графічного проміжного програмного забезпечення. Деякі графічні рушії містять серію слабо пов'язаних компонентів графічного проміжного програмного забезпечення, які можна вибірково комбінувати для створення власного механізму, замість більш поширеного підходу до розширення або налаштування гнучкого інтегрованого продукту. Однак досягнута, розширюваність залишається головним пріоритетом для ігрових двигунів завдяки широкому спектру використання, для якого вони застосовуються. Незважаючи на специфіку назви "графічний рушій", кінцеві користувачі часто переробляють графічні рушії для інших типів інтерактивних додатків із графічними вимогами в режимі реального часу - наприклад, демонстраційні маркетингові демонстрації, візуалізація архітектури, симуляції тренувань та середовища моделювання.

Деякі графічні рушії надають лише можливості 3D-рендерингу в режимі реального часу, а не широкий спектр функціональних можливостей, необхідних деяким ПЗ. Ці механізми покладаються на розробника ПЗ, щоб реалізувати решту цієї функціональності або зібрати її з інших компонентів графічного проміжного програмного забезпечення. Ці типи двигунів зазвичай називають "графічним двигуном", "механізмом рендерингу" або "двигуном 3D" замість більш охоплюючого терміна "ігровий рушій". Ця термінологія використовується непослідовно, оскільки багато повнофункціональних двигунів 3D-ігор Називає просто "3D-рушіями". Прикладами графічних механізмів є: Crystal Space, Genesis3D, Irrlicht, OGRE, RealmForge, Truevision3D та Vision Engine. Сучасні ігрові чи графічні рушії, Як правило, надають графік сцен – об’єктно-орієнтоване представлення 3D-віртуального світу, яке часто спрощує ігровий дизайн і може бути використано для більш ефективного відтворення величезних віртуальних світів.

У міру старіння технології компоненти рушія можуть застаріти або бути недостатніми для вимог даного проекту. Оскільки складність програмування абсолютно нового рушія може призвести до небажаних затримок (або вимагати перезапуску проекту з самого початку), команда розробників рушіїв може вибрати оновлення свого існуючого рушія новішими функціональними можливостями або компонентами.

У більш широкому розумінні цього поняття самі графічні рушії можна охарактеризувати як проміжне програмне забезпечення. Однак у контексті відеоігор термін "проміжне програмне забезпечення" часто використовується для позначення підсистем функціональності в ігровому механізмі. Деякі ігрові проміжні програми роблять лише одне, але роблять це переконливіше або ефективніше, ніж проміжне програмне забезпечення загального призначення.

Найпопулярнішими графічними рушіями є Unreal Engine та Unity. У наступній таблиці проведено їхнє порівняння (табл. 1.1):

Таблиця 1.1

Порівняння графічних рушіїв Unreal Engine та Unity

Unity	Unreal Engine
Мобільні застосунки	
Unity було створено з урахуванням мобільних додатків, тому розробка цих пристроїв дуже впорядкована як для 2D, так і для 3D заголовків. Більшість оптимізацій призначені для невеликих та інді-ігор з обмеженими вимогами до обробки.	Використання потужних графічних можливостей UE є надмірним для більшості мобільних додатків, проте оптимізація UE для додатків з високою обробкою робить його більш придатним для назв AAA та тих, хто орієнтований на висококласні пристрої.
Застосунки віртуальної реальності (VR)	
Іммерсивні програми, засновані на технології VR / MR / XR, підтримуються в обох двигунах, але	Це чудовий вибір для VR об’єктів, розроблених з великим рівнем деталізації. Він також пропонує

опція Unity ідеально підходить для створення прототипів та експериментів із VR функціями.	нижчий бар'єр для проникнення в VR дизайн завдяки своїй системі Blueprints
Графіка	
Це програмне забезпечення пропонує кілька вражаючих графічних функцій з самого початку (наприклад, глобальне освітлення та фізичну візуалізацію), але досягнення кращих візуальних ефектів часто передбачає велику кількість майстерності / редагування.	На додаток до вражаючих графічних функцій за замовчуванням, які надає аналог, об’єкти UE часто виглядають відполірованими прямо зі старту завдяки широкому діапазону попередньо встановлених налаштувань.
Спільнота	
Магазин активів на веб-сайті компанії величезний і наповнений 3D-активами, з деякими меншими, але помітними колекціями 2D-активів, шаблонів та VFX. Навчальних матеріалів на веб-сайті платформи багато.	Хоча ринок активів UE менший, він також пропонує пристойний вибір, переважно орієнтований на елементи гри. Розробники підтримуються незліченними посібниками та форумами.
Ціна	
Найпростіший тарифний план (Персональний) безкоштовний, але більш експансивні та орієнтовані на бізнес плани коштують $399 доларів на рік на один рахунок або більше.	Ця опція є безкоштовна, але вона використовує систему роялті, яка запускається, як тільки додаток монетизується, приносячи компанії (Epic Games) 5% прибутку.

Порівняльна характеристика переваг / недоліків Unity та Unreal Engine (табл. 1.2):

Таблиця 1.2

Порівняння переваг / недоліків Unity та Unreal Engine

Unity	Unreal Engine

+ Чудово підходить для створення простих мобільних додатків, включаючи прототипи та мобільні додатки AR / VR.	+ Спрямована на досягнення найвищої якості графіки
+ Найширша доступна підтримка між платформами.	+ Дозволяє користувачам налаштовувати шейдери без кодування
+ Простіше і швидше кодування за допомогою C #	+ Над-швидка пост-обробка
+ Вражаючий магазин активів	+ Усі групи користувачів мають доступ до вихідного коду
- Тюнінг графіки займає більше часу і роботи	- Модель ліцензування невигідна для великих комерційних проектів
- Візуалізація може бути повільною без оптимізації	- Занадто громіздкий для проектів меншого масштабу

РОЗДІЛ 2. ХАРАКТЕРИСТИКА ТЕХНОЛОГІЙ ПЛАТФОРМИ

2 Система подій

В інформатиці, подія (англ. event) — дія яка розпізнається програмним забезпеченням та оброблюється за допомогою певних інструкцій. Комп'ютерні події можуть бути згенеровані або ініціалізовані системно, користувачем або іншими способами. Як правило, події обробляються синхронно з програмою, тобто, програмне забезпечення може мати один або кілька виділених місць, де обробляються події, часто називається цикл подій.

Події виникають при виконанні користувачем певних дій, наприклад, натисканням клавіш на клавіатурі. Деякі події виникають по таймеру, наприклад перезапуск системи. Програмне забезпечення може також викликати свій власний набір подій в цикл обробки подій, наприклад, повідомити про завершення завдання.

Система (рис. 2.1) зазвичай використовує події, коли відбувається асинхронна зовнішня діяльність, яка повинна оброблятися програмою. Наприклад, це може бути користувач, який натискає мишкою. Система створює подію в циклі подій, який очікує якоїсь діяльності, наприклад, внутрішній сигнал тривоги. Коли один з них відбувається, система збирає дані про подію та відправляє в програму-обробник, який буде правильно розподіляти виконання тієї чи іншої події. Програма також може ігнорувати події.

Існують бібліотеки для подій, які програмісти розробляють для прослуховування певної події. Дані, пов'язані з подією, як мінімум, зберігають тип події, але також можуть включати в себе іншу інформацію. Події призначені для користувача інтерфейсом програми, де дії у

зовнішньому світі (натискання клавіш, зміни розміру вікна, повідомлення з інших програм) обробляються програмою як ряд подій.

Події також можуть бути використані на системному рівні, наприклад у консолі. В порівнянні з програмним та апаратним переривання, подія, обробляється синхронно, тобто програма явно перевіряє події, які будуть виконуватися, в той час як переривання може запросити виконання дій в будь-який момент.

У проекті реалізовано 3 типи диспетчерів подій:

- Диспетчер подій від системи (переривання)
- Диспетчер подій від платформи (повідомлення про помилку)
- Диспетчер подій від рушія графікою (помилки від OpenGL, діагностичні повідомлення)
- Диспетчер подій від рушія фізикою (повідомлення про колізію)
- Диспетчер подій від зовнішніх пристроїв (клавіатура, миша)

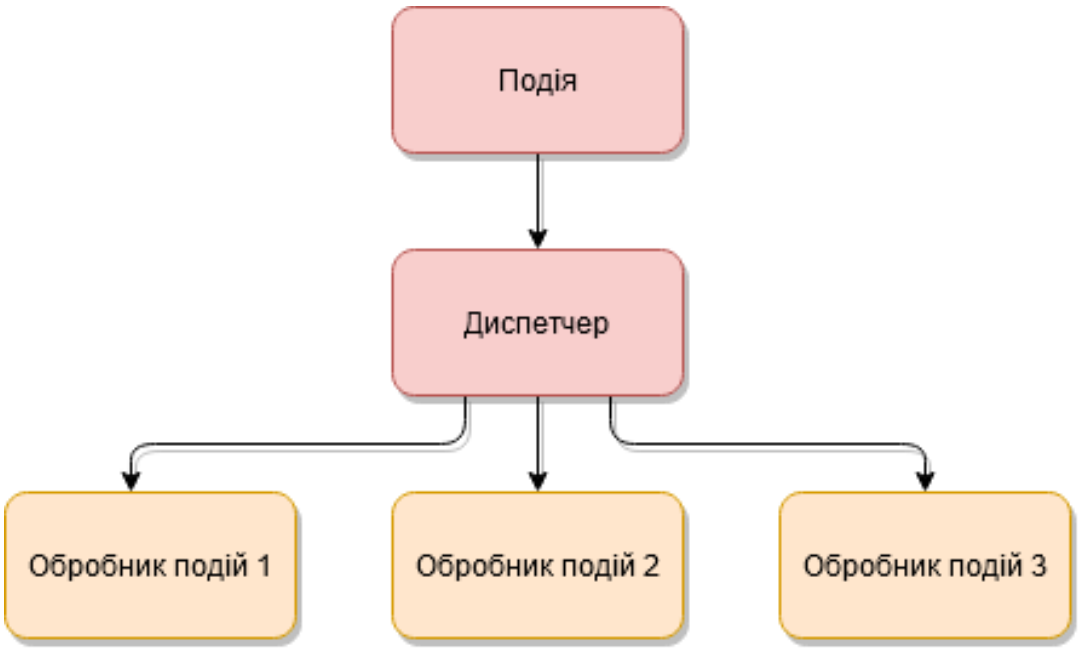


Рис. 2.1. Схема роботи системи подій

2.2 Інформаційно-орієнтовна (Data-oriented) парадигма програмування

У обчислювальній техніці орієнтований на дані дизайн - це підхід до оптимізації програм, мотивований ефективним використанням кешу центрального процесора, що використовується при розробці відеоігор. Підхід полягає в тому, щоб зосередитись на компонуванні даних, розділяючи та сортуючи поля відповідно до того, коли вони потрібні, і думати про перетворення даних.

Паралельний масив (або структура масивів) є основним прикладом проектування, орієнтованого на дані. Це протиставляється масиву структур, типових для об’єктно-орієнтованих конструкцій.

Ці методи стали особливо популярними в середині та наприкінці 2000-х років під час сьомого покоління відеоігор, що включали консолі PlayStation 3 (PS3) та Xbox 360 на базі IBM PowerPC. Історично ігрові приставки часто мають відносно слабкі центральні процесорні блоки (ЦП) у порівнянні з найпопулярнішими аналогами настільних комп'ютерів. Це вибір дизайну, щоб приділити більше енергії та транзисторного бюджету графічним процесорам (GPU). Наприклад, центральні процесори 7-го покоління не виготовлялись із сучасними процесорами виконання, що не працюють, а використовували процесори, що працюють в порядку, з високою тактовою частотою та глибокими конвеєрами. Крім того, більшість типів обчислювальних систем мають основну пам'ять, розташовану за сотні тактових циклів від елементів обробки (рис. 2.2). Крім того, оскільки процесори ставали швидшими разом із значним збільшенням обсягу основної пам'яті, спостерігається величезне споживання даних, що збільшує ймовірність пропусків кеш-пам’яті у спільній шині, інакше відому як вузьке вирішення Фон Неймана.

Отже, локальність еталонних методів була використана для контролю продуктивності, що вимагало вдосконалення шаблонів доступу до пам'яті для

виправлення вузьких місць. Деякі проблеми з програмним забезпеченням також були подібні до тих, що траплялися на Itanium, вимагаючи розгортання циклу для попереднього планування.

Твердження того факту, що полягає в тому, що традиційні принципи проектування об’єктно-орієнтованого програмування (ООП) призводять до поганої локалізації даних, тим більше, якщо використовується поліморфізм виконання (динамічне відправлення) (що особливо проблематично для деяких процесорів).

Хоча ООП, здається, "організовує код навколо даних", практика зовсім інша. ООР насправді стосується організації вихідного коду навколо типів даних, а не фізичного групування окремих полів та масивів у ефективному форматі для доступу за допомогою певних функцій (рис. 2.3). Більше того, він часто приховує деталі макета під шарами абстракції, тоді як програміст, орієнтований на дані, хоче розглянути їх насамперед.

Коли парадигма програмування дозволяє змінювати дані, розробникам доводиться додавати механізми захисту своїх даних. Наприклад, коли ми передаємо фрагмент даних (інкапсульований в об’єкт або на хеш-карту) функції, ми ніколи не можемо бути впевнені на 100%, що функція не змінить наші дані. У багато-потокових системах нам потрібні всі види мьютексів, щоб інші потоки не змінювали дані в несподіваний час. Мьютекси роблять наш код більш складним і спричиняють заваду при досягненні продуктивності.

Об’єктно-орієнтоване програмування протягом багатьох років навчило нас моделювати світ за допомогою об’єктів. Кожна частина інформації повинна бути інкапсульована в об'єкти, створені з класів: ми маємо класи для бізнес-структур, таких як клієнти та продукти, а також для універсальних концепцій програмування, таких як дати. В ООП немає можливості агрегувати частини інформації без створення класу. Коли дані інкапсульовані в об’єкт, він втрачає свою прозорість: ми більше не можемо легко перевірити

дані або серіалізувати їх загальним способом (без написання комплексного коду або використання рефлексії).

Існують дві основні речі, які DOD вважає, що програмі слід уникати:

- Зміна даних
- Зв'язок коду (логіки) та даних

Основними об'єктами DOD є незмінні колекції.

Під колекцією ми маємо на увазі щось на зразок словника, де ключі в основному є рядками, а значення є або примітивними типами, або колекціями. Під незмінною, ми маємо на увазі, що колекції не можуть бути змінені на місці, на відміну від хеш-карт (або словників) у більшості мов програмування.

Незмінні колекції мають 3 важливі властивості:

- Вони незмінні
- Вони не потребують креслення для створення з них
- Ними можна маніпулювати за допомогою загальних функцій

Підхід DOD допомагає нам думати про дані як про одиницю інформації. Одиниці інформації ніколи не змінюються. Візьмемо для прикладу число 42. Значення числа 42 назавжди залишиться 42, навіть коли ми додаймо до нього 10. У світі, орієнтованому на дані, те саме стосується колекцій. Колекція ніколи не змінюється, і це хорош є перевагою для наших програм. Усередині програм, які дотримуються парадигми незмінності DOD, колекціями маніпулюють з такою ж простотою, як ми маніпулюємо числами в будь-якій мові програмування.

Замість того, щоб створювати класи для моделювання світу, ми використовуємо універсальні колекції даних. Клієнти, товари, замовлення тощо ... представлені у вигляді словників із ключами та значеннями. Різниця між ними полягає в тому, що ключі мають різні імена, і значення не є однотипними.

Колекції є універсальними. Отже, ми можемо писати функції, які маніпулюють колекціями без конкретного знання сутності, яка представлена колекцією. Наприклад, ми можемо написати функцію, яка перевіряє поле адреси електронної пошти колекції та передає цій функції колекцію клієнта та ім'я поля, що містить адресу електронної пошти.

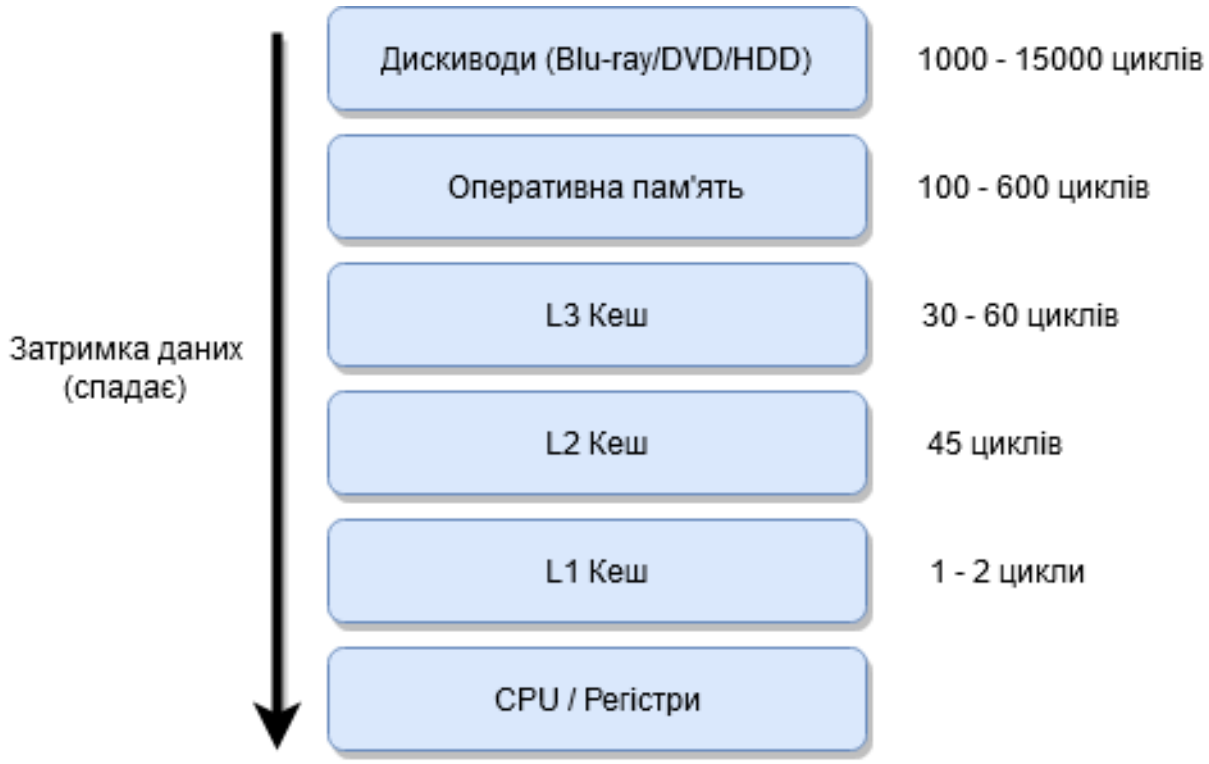


Рис. 2.2. Схема швидкодії пристроїв пам’яті

Рис. 2.3. Схема розміщення даних у пам’яті за парадигмою ООП

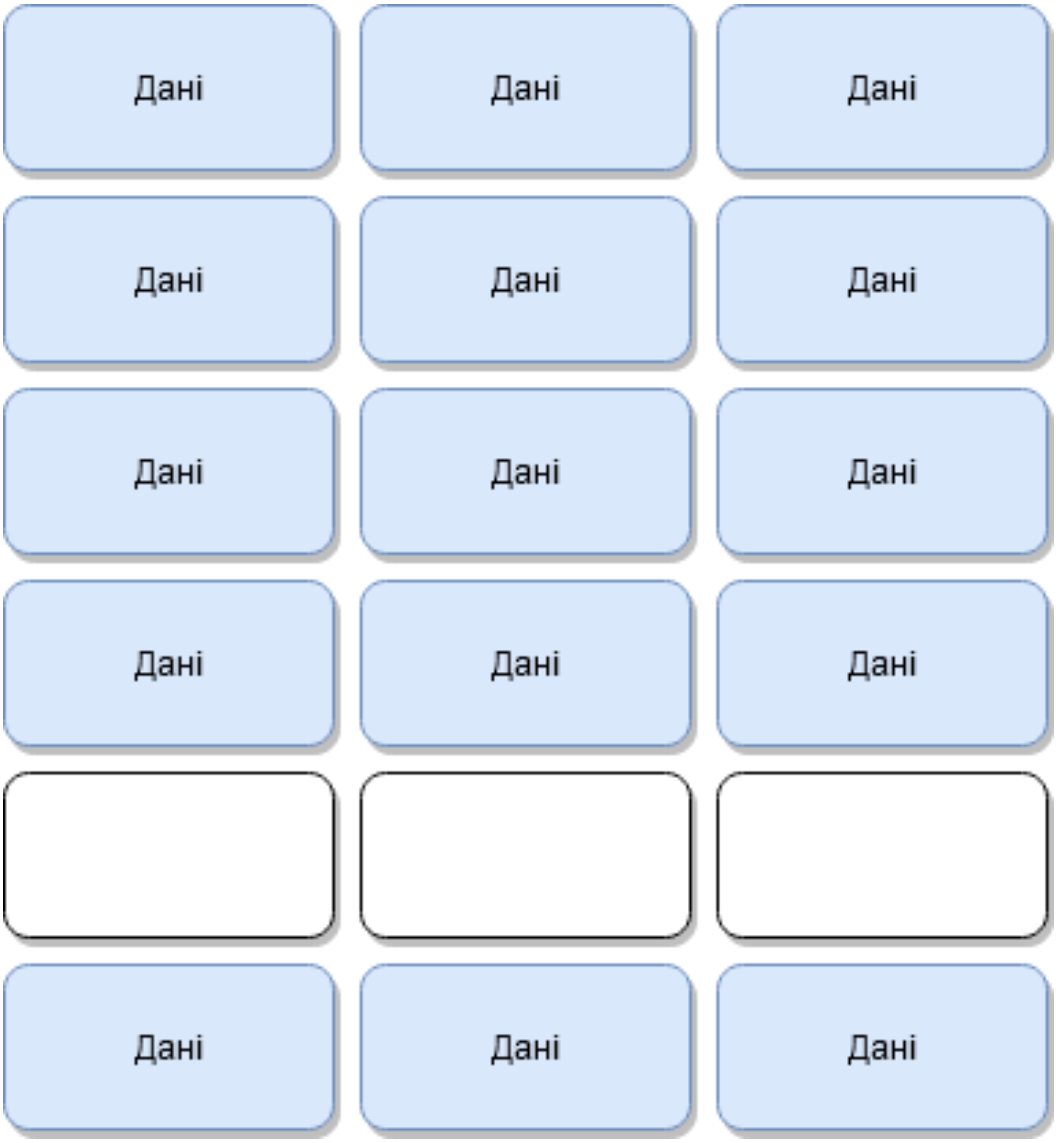


Рис. 2.4. Схема розміщення даних у пам’яті за парадигмою ІОД

2.3 Система компонентів та сутностей

Сутність-компонент-система (ECS) - це архітектурний шаблон, який в основному використовується при розробці відеоігор. ECS дотримується принципу композиції над успадкуванням, що дозволяє збільшити гнучкість у визначенні сутностей, де кожен об’єкт на ігровій сцені є суттю (наприклад, вороги, кулі, транспортні засоби тощо) (рис. 2.5).

Кожна сутність складається з одного або декількох компонентів, які містять дані або стан. Отже, поведінка сутності може бути змінена під час виконання системами, які додають, видаляють або змінюють компоненти. Це

усуває проблеми неоднозначності глибоких та широких ієрархій успадкування, які важко зрозуміти, підтримувати та поширити (рис. 2.6).

Поширені підходи ECS є дуже сумісними і часто поєднуються з орієнтованими на дані методами проектування.

У деяких архітектурах ECS всі сутності мають спільні біти даних, такі як інформація про перетворення, наприклад, положення, обертання та масштаб, або деякі основні дані про стан, чи є об’єкт на даний момент «живим» на сцені. Unity GameObjects -- один із прикладів цього стилю. В інших версіях шаблону всі можливі біти даних витягуються в компоненти, і сутність в кінцевому підсумку функціонально є просто контейнером для компонентів.

Розбиття всіх наших об'єктів на набори компонентів призводить до вирішення першої проблеми та способу, яким ми можемо реалізувати орієнтований на дані дизайн: ми можемо переставити дані з масиву структур на структуру масивів.

Для цього нам також потрібно відокремити поведінку та дані для кожного компонента. Наші компоненти тепер є структурами чистих даних, і частини поведінки, які діють на одному (або декількох) компонентах типу, називаються "системами" (рис. 2.7).

Деякі компоненти (наприклад, Position) спільно використовуються між системами. Інші компоненти можуть бути приватними для певної системи, що забезпечує приховування інформації та більшу оптимізацію макетів пам'яті.

Наприклад, скажімо, одна сцена в грі містить тисячу сутностей. Майже всі ці сутності містять позицію, тому ми могли б просто виділити великий масив з 1000 елементів для наших компонентів позиції та зберегти позицію для кожної сутності, індексованої її ідентифікатором сутності. Оскільки масив майже повністю заповнений, і ми хочемо регулярно посилатися на

позиції сутності, це, мабуть, найефективніший спосіб зберігання цієї інформації.

З іншого боку, можливо, лише відносно невелика кількість об’єктів нашої сцени має швидкість. У цьому випадку ми можемо використовувати деякі інші стратегії, щоб виділити достатньо пам’яті для швидкості, яку ми маємо, а потім зіставити ідентифікатори сутності в наш масив швидкостей.

Подібні стратегії дозволяють нам краще працювати, ніж загальнодоступний Garbage Collector (Збирач сміття) чи розподільник системи - хоча, звичайно, ми повинні бути обережними та постійно складати профілі, щоб переконатися, що насправді ми отримуємо вигоду від додаткової складності управління власною пам'яттю.

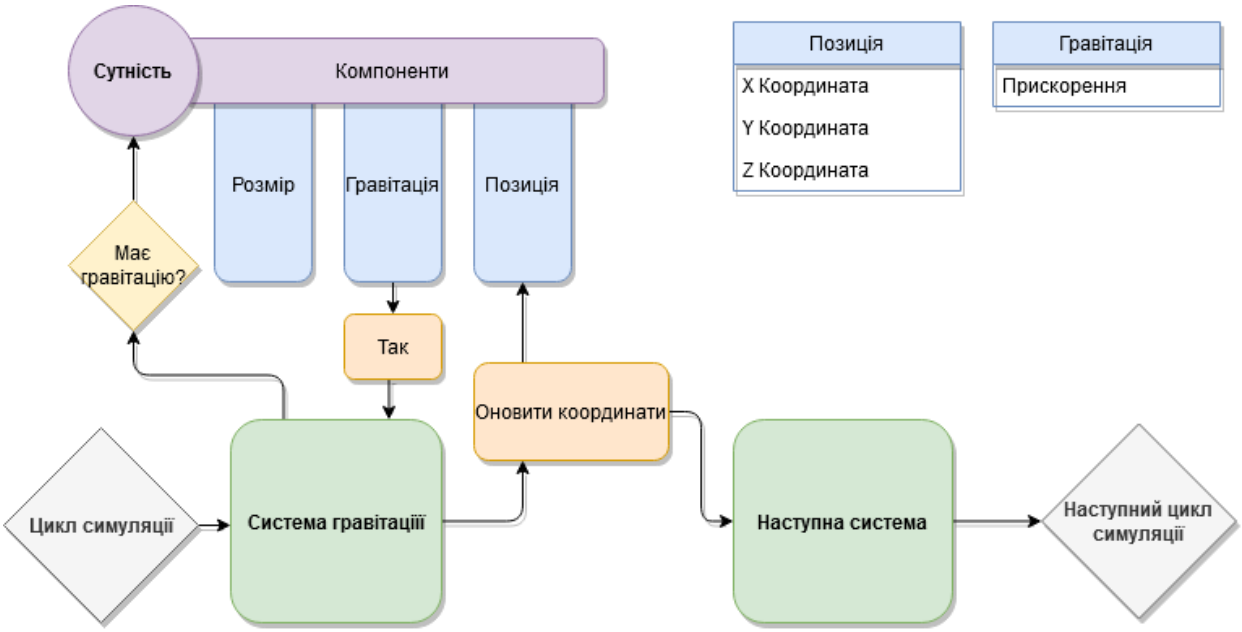


Рис. 2.5. Схема роботи ECS

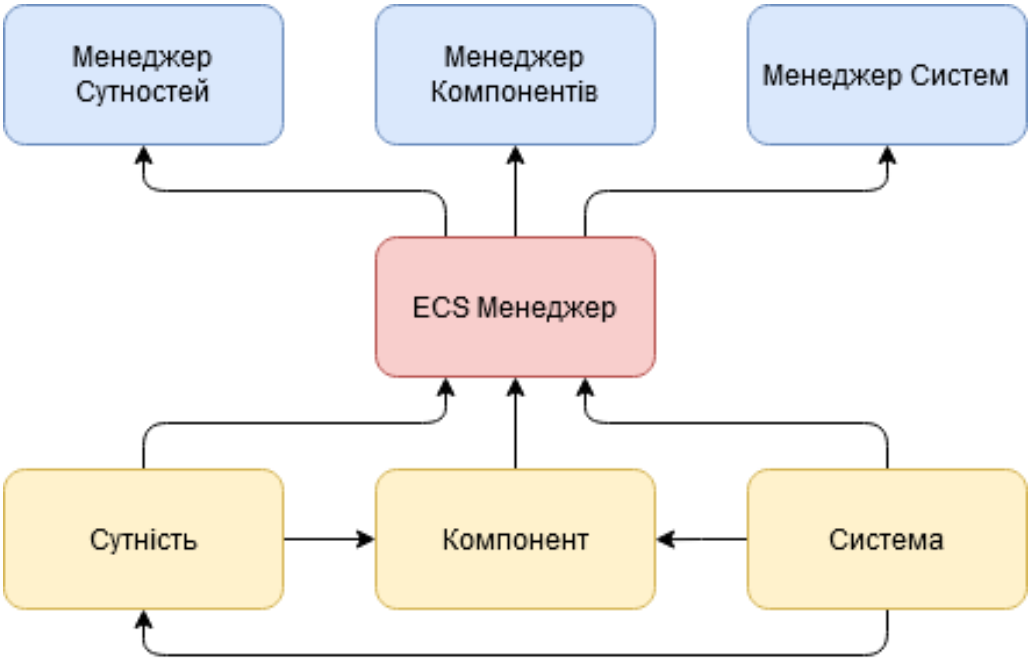


Рис. 2.6. Схема структури ECS

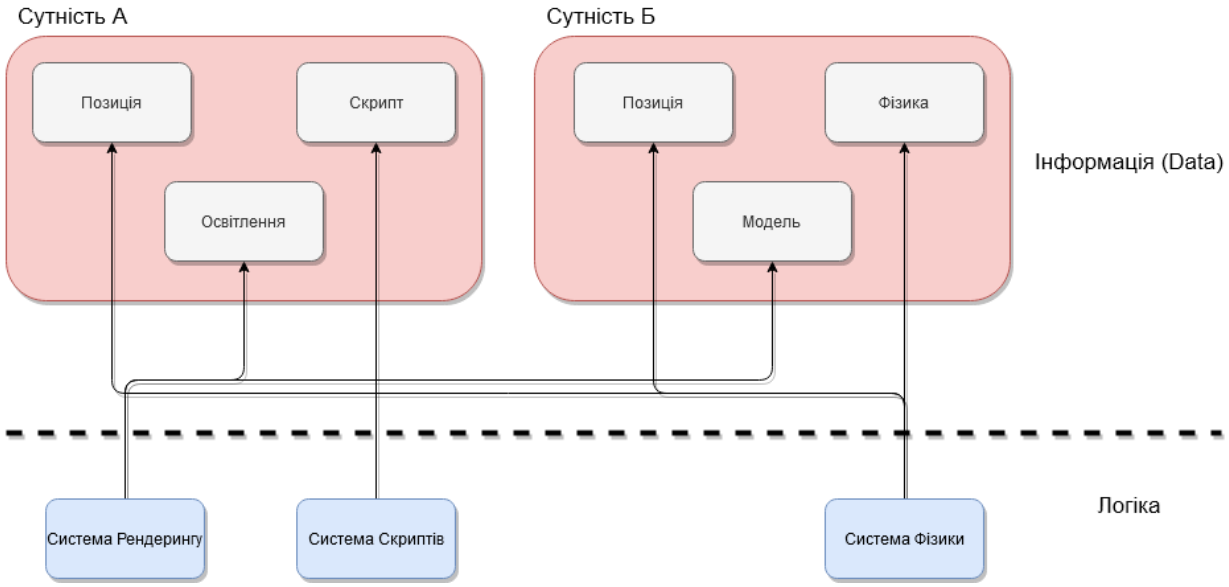


Рис. 2.7. Схема поділу інформації та логіки у ECS

Сутність (ECS)

Сутність є об'єктом загального призначення. Зазвичай він складається лише з унікального ідентифікатора. Вони "позначають кожен об'єкт грубої гри як окремий елемент". Реалізації зазвичай використовують для цього звичайне ціле число.

Компонент (ECS)

Компонент представляє вихідні дані для одного аспекту об’єкта та спосіб його взаємодії зі світом. "Позначає Суб'єкт господарювання як такого, що володіє цим аспектом". Реалізації зазвичай використовують структури, класи або асоціативні масиви.

Система (ECS)

Кожна система працює безперервно (як би кожна система мала свій приватний потік) і виконує глобальні дії над кожним Суб'єктом, який має Компонент того самого аспекту, що і ця Система.

2.4 Рушій графіки

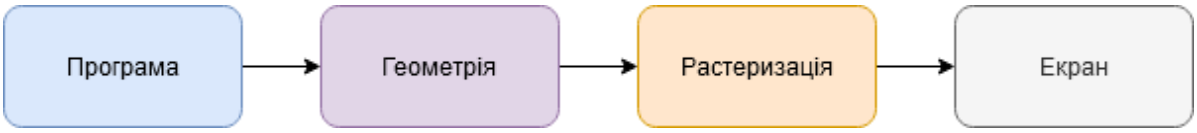


Рис. 2.8. Спрощена схема графічного конвеєру

У комп’ютерній графіці конвеєр комп’ютерної графіки, конвеєр візуалізації або просто графічний конвеєр - це концептуальна модель, яка описує, які кроки графічна система повинна виконати, щоб зробити 3D-сцену на 2D-екрані. Після створення 3D-моделі, наприклад, у відеоігри чи будь-якій іншій комп'ютерній анімації 3D, графічний конвеєр - це процес перетворення цієї 3D-моделі в те, що відображає комп'ютер. Оскільки кроки, необхідні для цієї операції, залежать від використовуваного програмного та апаратного забезпечення та бажаних характеристик дисплея, універсальний графічний конвеєр не підходить для всіх випадків. Однак графічні інтерфейси програмування програм (API), такі як Direct3D та OpenGL, були створені для уніфікації подібних кроків та управління графічним конвеєром даного апаратного прискорювача. Ці API абстрагують базове обладнання та утримують програміста від написання коду для маніпулювання апаратними прискорювачами графіки (AMD / Intel / NVIDIA тощо) (рис. 2.8).

Модель графічного конвеєру зазвичай використовується для рендерингу в режимі реального часу. Часто більшість кроків конвеєра реалізуються в апаратному забезпеченні, що дозволяє проводити спеціальні оптимізації. Термін "конвеєр" використовується в тому самому розумінні, що і конвеєр у процесорах: окремі щаблі конвеєр працюють паралельно, доки будь-який даний крок має те, що йому потрібно.

3D конвеєр зазвичай відноситься до найпоширенішої форми комп'ютерного 3D–рендерингу, яка називається 3D полігональне моделювання відмінна від трасування променів та кидання променів. При киданні променів виникає в точці, де знаходиться камера, і якщо цей промінь потрапляє на поверхню, розраховується колір і освітлення точки на поверхні, де обчислюється удар. У режимі рендерингу багатокутника в 3D відбувається зворотне – обчислюється площа, яка переглядається камерою, а потім створюються промені від кожної частини кожної поверхні з огляду на камеру і відслідковуються до камери.

Графічний конвеєр можна розділити на три основні частини (Рис. 2.8):

- Застосування
- Геометрія
- Растеризація

Програма

Крок програми виконується програмним забезпеченням на головному процесорі (CPU). На кроці програми внесення змін до сцени за потреби, наприклад, шляхом взаємодії користувача за допомогою пристрою введення або під час анімації. У сучасному ігровому рушії, такому як Unity, програміст займається майже виключно кроком програми та використовує мову високого рівня, таку як C #, на відміну від C або C ++. Нова сцена з усіма її примітивами, як правило, трикутниками, лініями та точками, переходить до наступного кроку в конвеєрі.

Прикладами завдань, які зазвичай виконуються на етапі програми, є методи виявлення зіткнень, анімація, морфінг та прискорення з використанням просторових схем підрозділу, таких як древо квадрантів чи дерево октантів. Вони також використовуються для зменшення обсягу основної пам'яті, необхідної в даний момент часу. "Світ" сучасної комп'ютерної гри набагато більший, ніж те, що могло б вписатись в пам'ять одразу.

Геометрія

Крок геометрії, який відповідає за більшість операцій з полігонами та їх вершинами, можна розділити на наступні п’ять завдань (рис. 2.9). Це залежить від конкретної реалізації того, як ці завдання організовані як фактичні паралельні кроки конвеєру.

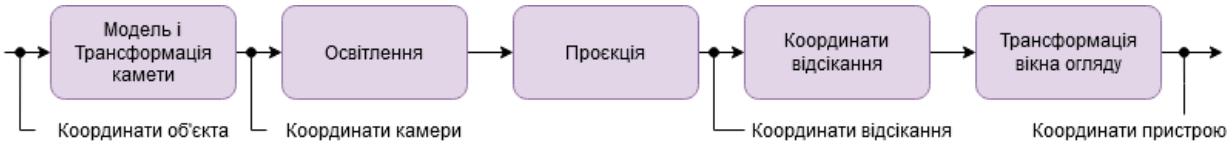


Рис. 2.9. Схема трансформації координат

Визначення полігонів

Вершина (множина: вершини) — точка у світі. Для з'єднання поверхонь використовується багато точок. В особливих випадках хмари точок малюються безпосередньо, але це все ж виняток.

Трикутник – найпоширеніший геометричний примітив комп’ютерної графіки. Він визначається його трьома вершинами і вектором нормалей — вектор служить для позначення передньої грані трикутника і є вектором, перпендикулярним поверхні. Трикутник може бути забезпечений кольором або текстурою (зображення "склеєне" зверху). Трикутники завжди існують в одній площині, тому вони віддають перевагу перед прямокутниками.

Світова система координат

Світова система координат – це система координат, в якій створюється віртуальний світ. Це повинно відповідати декільком умовам, щоб наступна математика була легко застосована:

- Це повинна бути прямокутна декартова система координат, у якій всі осі однаково масштабуються.

Як визначено одиницю системи координат, залишається розробнику. Від того, чи повинен одиничний вектор системи відповідати реально одному метру або іншій одиниці вимірювання, залежить від застосування.

- Використовувати декартовоі системи координат може визначати графічна бібліотека, яка буде використовуватися.

  Приклад: Якщо ми плануємо розробити тренажер польоту, ми можемо вибрати світову систему координат, щоб початок знаходився посередині землі, а одиниця була встановлена на один метр. Крім того, щоб полегшити посилання на реальність, ми визначаємо, що вісь X повинна перетинати екватор на нульовому меридіані, а вісь Z проходить через полюси. У правій системі вісь Y проходить через меридіан 90 ° – Схід (десь в Індійському океані). Тепер у нас є система координат, яка описує кожну точку на Землі у тривимірних декартових координатах. У цій системі координат ми зараз моделюємо принципи нашого світу, гір, долин та океанів.

  Примітка. Окрім комп'ютерної геометрії, для землі використовують географічні координати, тобто широту і довготу, а також висоту над рівнем моря. Орієнтовна конверсія – якщо не враховувати той факт, що земля не є точною сферою – проста:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} (R + hasl) * \cos(lat) * \cos(long) \\ (R + hasl) * \cos(lat) * \sin(long) \\ (R + hasl) * \sin(lat) \end{pmatrix}$$

  де R = радіус Землі [6.378.137m], lat = Широта, long = Довгота, hasl = висота над рівнем моря.

Об'єкти, що містяться в сцені (будинки, дерева, машини), часто проектуються у власній системі об'єктів координат (також її називають модельною системою координат або локальною системою координат) з міркувань більш простого моделювання. Для присвоєння цим об'єктам координат у світовій системі координат або глобальній системі координат всієї сцени координати об'єкта перетворюються за допомогою перекладу, обертання або масштабування. Це робиться шляхом множення відповідних матриць переходу. Крім того, з одного об’єкта може бути сформовано кілька різнорідно перетворених копій, наприклад ліс з дерева.

Ця методика називається інстанцією.

Для того, щоб розмістити модель літака у світі, ми спочатку визначаємо чотири матриці. Оскільки ми працюємо в тривимірному просторі, нам потрібні чотиривимірні однорідні матриці для наших розрахунків.

Спочатку нам потрібні три матриці обертання, а саме по одній для кожної з трьох осей літака (вертикальна вісь, поперечна вісь, поздовжня вісь).

Навколо осі X (зазвичай визначається як поздовжня вісь у системі координат об'єкта):

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(a) & \sin(a) & 0 \\ 0 & -\sin(a) & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Навколо осі Y (зазвичай визначається як поперечна вісь в об'єктній системі координат):

$$R_y = \begin{pmatrix} \cos(a) & 0 & -\sin(a) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(a) & 0 & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Навколо осі Z (зазвичай визначається як вертикальна вісь в системі об'єктів координат):

$$R_z = \begin{pmatrix} \cos(a) & \sin(a) & 0 & 0 \\ -\sin(a) & \cos(a) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Ми також використовуємо матрицю перекладу, яка переміщує літальний апарат до потрібної точки нашого світу:

$$T_{x,y,z} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{pmatrix}$$

Тепер ми обчислити положення вершин літака у світових координатах, помноживши кожну точку послідовно на ці чотири матриці. Оскільки множення матриці з вектором є досить дорогим (забирає багато часу), зазвичай проходить інший шлях і спочатку множить чотири матриці разом. Множення двох матриць ще дорожче, але повинно бути виконане лише один раз для всього об'єкта. Множення $\left(\left(\left(v * R_x\right) * R_y\right) * R_z\right) * T\right)$ та $v * \left(\left(\left(R_x * R_y\right) * R_z\right) * T\right)$ $\left(\left(\left(v * R_x\right) * R_y\right) * R_z\right) * T\right)$ та $v * \left(\left(\left(R_x * R_y\right) * R_z\right) * T\right)$ рівнозначні. Після цього отримана матриця може бути застосована до вершин. На практиці, однак, множення з вершинами все ще не застосовується, але матриці камери – див. Нижче – визначаються спочатку.

Порядок застосування матриць є важливим, оскільки множення матриць не є комутативним. Це стосується також трьох обертів, як це можна продемонструвати на прикладі: Точка (1, 0, 0) лежить на осі X, якщо повернути її спочатку на 90 ° навколо X–, а потім навколо осі Y, він закінчується на осі Z (обертання навколо осі X не впливає на точку, яка знаходиться на осі). Якщо, з іншого боку, спершу обертається навколо осі Y, а потім навколо осі X, отримана точка розташована на осі Y. Сама послідовність довільна до тих пір, поки вона завжди однакова. Послідовність з x, то y, тоді z (рулон, крок, заголовок) часто най-інтуїтивніша, оскільки обертання призводить до того, що напрямок компаса збігається з напрямком «носа».

Для визначення цих матриць також є дві конвенції, залежно від того, чи бажаєте ви працювати з векторами стовпців або векторами рядків. Тут різні графічні бібліотеки мають різні переваги. OpenGL віддає перевагу векторам стовпців, рядковим векторам DirectX. Рішення визначає, з якої сторони точкові вектори потрібно помножити на матриці перетворення. Для векторів стовпців множення виконується праворуч, тобто $v_{out} = M * v_{in}$, де v_{out} та v_{in} $v_{out} = M * v_{in}$, де v_{out} та v_{in} 4x1 стовпчикові вектори. Зв'язування матриць також робиться справа наліво, тобто, наприклад $M = T_x * R_x M = T_x * R_x$, при першому обертанні, а потім перемиканні.

У випадку векторів рядків це працює саме навпаки. Зараз множення відбувається зліва як $v_{out} = v_{in} * M$ $v_{out} = v_{in} * M$ з векторами 1x4–рядків і конкатенація $M = R_x * T_x M = R_x * T_x$ коли ми також спочатку обертаємося, а потім рухаємося. Матриці, показані вище, дійсні для другого випадку, тоді як ті для векторів стовпців переміщуються. Правило $(v * M)^t = M^T * v^T$ $(v * M)^t = M^T * v^T$ застосовується, що для множення з векторами означає, що ви можете перемикати порядок множення, переміщуючи матрицю.

Цікавим у цьому ланцюжку матриць є те, що нова система координат визначається кожним таким перетворенням. Це можна продовжити за бажанням. Наприклад, гвинт літака може бути окремою моделлю, яка потім розміщується перекладом на ніс літака. У цьому перекладі потрібно лише описати перехід від модельної системи координат до системи координат гвинта. Для того, щоб намалювати весь літальний апарат, спочатку визначається матриця перетворення для літального апарату, точки трансформуються, а потім матриця моделі гвинта примножується на матрицю літака, а потім перетворюються точки гвинта.

Матриця, обчислена таким чином, також називається матрицею світу. Він повинен бути визначений для кожного об'єкта у світі до надання. Додаток може ввести тут зміни, наприклад змінити положення літака відповідно до швидкості після кожного кадру.

Перетворення камери

Окрім об'єктів, сцена також визначає віртуальну камеру чи переглядач, який вказує положення та напрямок перегляду, з якого повинна бути відображена сцена. Для спрощення подальшого проектування та відсікання сцена перетворюється таким чином, що камера знаходиться біля початку, дивлячись вздовж осі Z. Отримана система координат називається системою координат камери, а перетворення називається трансформацією камери або переглядом трансформації (рис. 2.10).

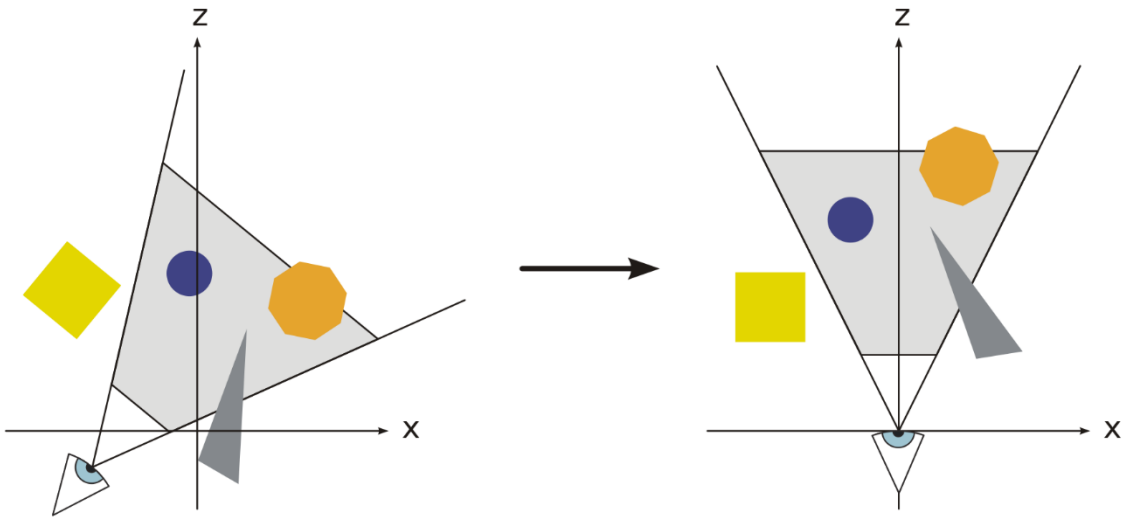


Рис. 2.10. Зліва положення та напрям віртуального переглядача (камери), визначений користувачем. Праворуч: Розташування об'єктів після перетворення камери. Світло–сіра зона – це видимий об'єм.

Матриця перегляду зазвичай визначається з положення камери, цільової точки (там, де дивиться камера) та "вектора вгору" ("вгору" з точки зору глядача). Спочатку потрібні три допоміжні вектори:

- Zaxis = нормальний (cameraPosition – cameraTarget)
- Xaxis = нормальний (крос (cameraUpVector, zaxis))
- Yaxis = хрест (zaxis, xaxis)

- При нормальному (v) = нормалізації вектора v;
- Вектор (v1, v2) = векторний добуток v1 і v2.

$$\begin{pmatrix} xaxis.x & yaxis.x & zaxis.x & 0 \\ xaxis.y & yaxis.y & zaxis.y & 0 \\ xaxis.z & yaxis.z & zaxis.z & 0 \\ -dot(xaxis, camPos) & -dot(yaxis, camPos) & -dot(zaxis, camPos) & 1 \end{pmatrix}$$

де крапка dot(v1, v2) = скалярний добуток v1 і v2.

Проекція

Крок 3D–проекції перетворює об'єм перегляду в куб з координатами кутових точок (–1, –1, 0) та (1, 1, 1); Інколи також використовуються інші цільові обсяги. Цей крок називається проекцією, хоча він перетворює об'єм в інший об'єм, оскільки отримані Z координати не зберігаються на зображенні, а використовуються лише у Z-буферизації на наступному етапі растрування. У перспективній ілюстрації використовується центральна проекція. Для обмеження кількості відображуваних об'єктів використовуються дві додаткові площини відсікання. Таким чином, візуальний об'єм є зрізаною пірамідою. Паралельна або ортогональна проекція використовується, наприклад, для технічних зображень, оскільки вона має перевагу в тому, що всі паралелі в об'єктному просторі також є паралельними в просторі зображення, а поверхні та об'єми мають однаковий розмір незалежно від відстані від глядача. Карти використовують, наприклад, ортогональну проекцію (так званий ортофотографія), але косі зображення пейзажу не можуть бути використані таким чином – хоча їх технічно можна зобразити, вони здаються такими спотвореними, що ми не можемо використовувати їх. Формула для обчислення матриці перспективного відображення:

$$\begin{pmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & \frac{far}{near - far} & -1 \\ 0 & 0 & \frac{near * far}{near - far} & 0 \end{pmatrix} \begin{pmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & \frac{far}{near - far} & -1 \\ 0 & 0 & \frac{near * far}{near - far} & 0 \end{pmatrix},$$

Де h = cot (fieldOfView / 2.0) (кут діафрагми камери); w = h / aspectRatio (співвідношення сторін цільового зображення); поблизу = Найменша відстань, яку видно; далеко = Найдовша відстань, яку видно.

Причини, з яких тут потрібно надати найменшу та найбільшу відстань, – це, з одного боку, те, що ця відстань ділиться на, щоб досягти масштабування сцени (більш віддалені об'єкти менші за перспективним зображенням, ніж поблизу об’єктів), а з іншого боку, для масштабування значень Z до діапазону 0..1 для заповнення Z – буфера. Цей буфер часто має лише роздільну здатність 16 біт, тому значення близького та далекого значення слід вибирати ретельно. Занадто велика різниця між близьким і далеким значенням призводить до так званих Z–боїв через низьку роздільну здатність буфера Z –. З формули також видно, що близьке значення не може бути 0, оскільки ця точка є точкою фокусування проекції. На даний момент немає жодної картини.

Для повноти формула паралельної проекції (ортогональна проекція):

$$\begin{pmatrix} \frac{2.0}{w} & 0 & 0 & 0 \\ 0 & \frac{2.0}{w} & 0 & 0 \\ 0 & 0 & \frac{1.0}{near-far} & -1 \\ 0 & 0 & \frac{near}{near-far} & 0 \end{pmatrix} \begin{pmatrix} \frac{2.0}{w} & 0 & 0 & 0 \\ 0 & \frac{2.0}{w} & 0 & 0 \\ 0 & 0 & \frac{1.0}{near-far} & -1 \\ 0 & 0 & \frac{near}{near-far} & 0 \end{pmatrix},$$

Де w = ширина цільового куба (розмірність в одиницях світової системи координат); H = w / aspectRatio (співвідношення сторін цільового зображення); поблизу = Найменша відстань, яку видно; далеко = Найдовша відстань, яку видно.

З метою ефективності матриця камери та проекції зазвичай поєднується в матрицю перетворення, щоб система координат камери була опущена. Отримана матриця зазвичай однакова для одного зображення, тоді як світова матриця виглядає по–різному для кожного об'єкта. Тому на практиці огляд та проекція попередньо розраховуються так, що під час

відображення має бути адаптована лише світова матриця. Однак можливі і більш складні перетворення, такі як вершинне змішування. Також можуть бути виконані вільно програмовані шейдери геометрії, що змінюють геометрію.

На етапі фактичного відображення обчислюється матриця проекції матриці світової матриці * камери *, а потім остаточно застосовується до кожної окремої точки. Таким чином, точки всіх об'єктів переносяться безпосередньо в систему координатних екранів (принаймні майже, діапазон значень осей все ще –1..1 для видимого діапазону, див. Розділ «Вікно – Відображення – Трансформація»).

Освітлення

Часто сцена містить джерела світла, розміщені в різних положеннях, щоб зробити освітлення предметів більш реалістичним. У цьому випадку коефіцієнт посилення текстури розраховується для кожної вершини на основі джерел світла та властивостей матеріалу, пов'язаних із відповідним трикутником. На пізньому етапі растерізації значення вершин трикутника інтерполюються над його поверхнею. На всі поверхні застосовується загальне освітлення (навколишнє світло). Це дифузна і, таким чином, напрямково–незалежна яскравість сцени. Сонце – це спрямоване джерело світла, яке можна вважати нескінченно далеко. Освітленість, здійснювана сонцем на поверхні, визначається формуванням скалярного добутку спрямованого вектора від сонця та нормального вектора поверхні. Якщо значення негативне, поверхня повернута до сонця (рис. 2.11).

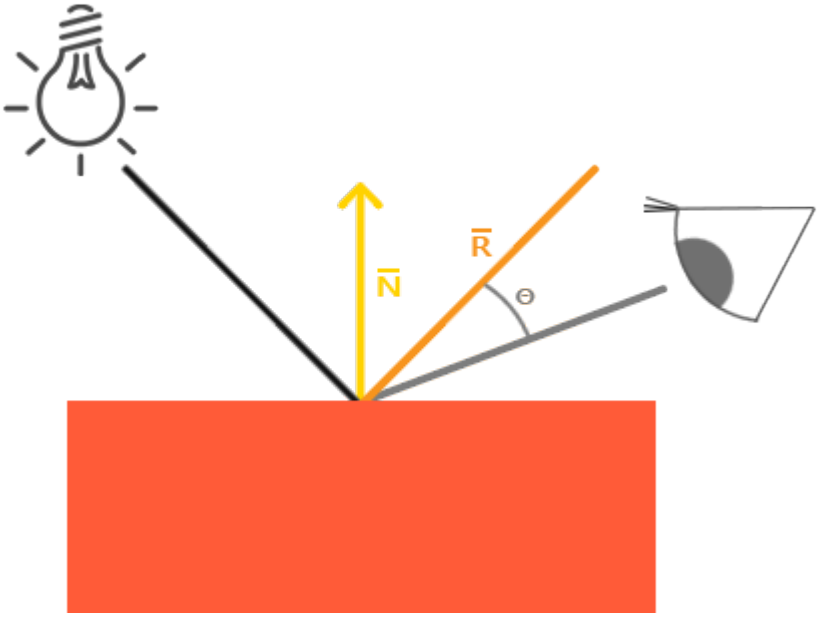


Рис. 2.11. Спрощена схема освітлення

Відсікання

Тільки примітиви, що знаходяться у візуальному обсязі, насправді потрібно вирощувати (малювати) (Рис. 2.12). Цей візуальний об'єм визначається як внутрішня частина зрізаної піраміди, форма у вигляді піраміди із відрізаною верхівкою. Примітиви, які повністю знаходяться поза зоровим об'ємом, викидаються; Це називається випаданням зрізаної пераміди. Подальші способи відсікання, такі як зворотне відсікання, які зменшують кількість примітивів, які слід розглядати, теоретично можуть бути виконані на будь–якому етапі графічного конвеєра. Примітиви, які знаходяться лише частково всередині куба, повинні бути притиснутим до куба. Перевага попереднього кроку проектування полягає в тому, що відсікання завжди відбувається проти одного куба. На остаточний крок пересилаються лише примітиви, можливо відрізані, які знаходяться у візуальному обсязі.

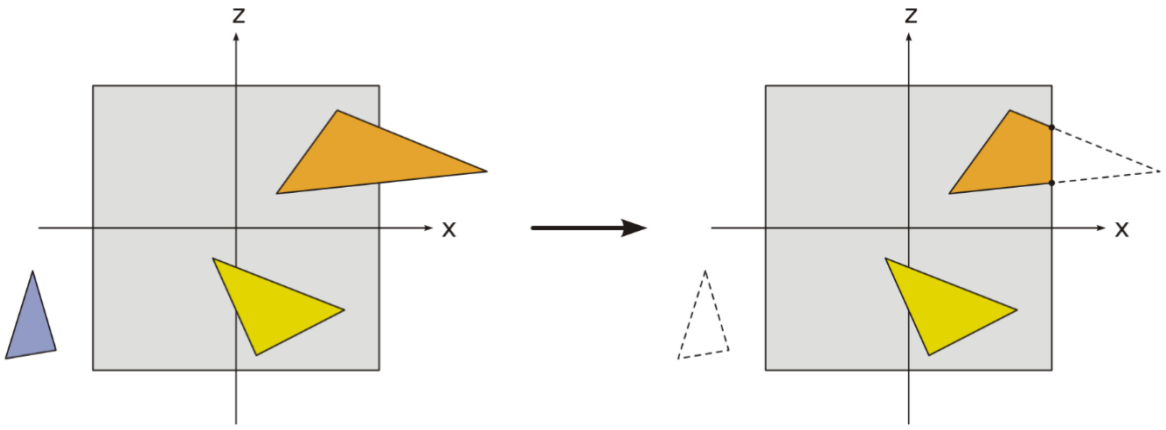


Рис. 2.12. Відсікання примітивів проти куба. Синій трикутник відкидається, коли помаранчевий трикутник обрізається, створюючи дві нові вершини

Перетворення вікна-перегляду

Для виведення зображення в будь–яку цільову область (вікно перегляду) екрану необхідно застосувати ще одне перетворення, перетворення Вікно – Перегляд (рис. 2.13). Це зсув з подальшим масштабуванням. Отримані координати – це координати пристрою вихідного пристрою. Вікно перегляду містить 6 значень: висота та ширина вікна у пікселях, лівий верхній кут вікна у координатах вікна (зазвичай 0, 0) та мінімальні та максимальні значення для Z (зазвичай 0 та 1).

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} vp.X + (1.0 + v.X) * vp.\dfrac{width}{2.0} \\ vp.Y + (1.0 + v.Y) * vp.\dfrac{height}{2.0} \\ vp.minz + v.Z * (vp.maxz - vp.minz) \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} vp.X + (1.0 + v.X) * vp.\dfrac{width}{2.0} \\ vp.Y + (1.0 + v.Y) * vp.\dfrac{height}{2.0} \\ vp.minz + v.Z * (vp.maxz - vp.minz) \end{pmatrix},$$

де vp = Viewport (Вікно відсікання); v = Точка після проекції

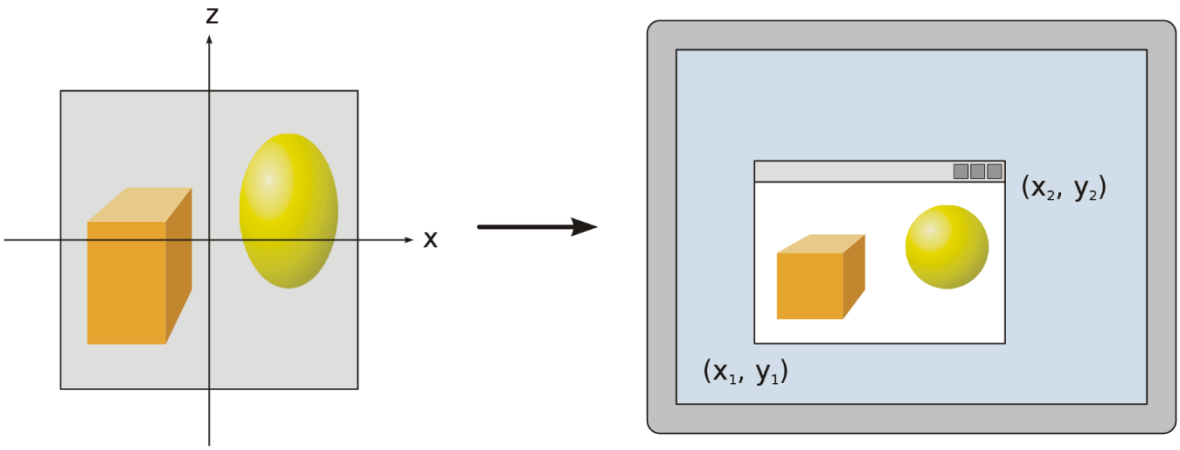


Рис. 2.13. Вікно – Вигляд – Трансформація

Растеризація

Етап растерізації є завершальним кроком перед фрагментним конвеєром шейдерів, де всі примітиви растрифіковані. На етапі растерізації створюються дискретні фрагменти із суцільних примітивів.

На цій стадії графічного конвеєра точки сітки також називають фрагментами, задля більшої виразності. Кожному фрагменту відповідає один піксель у буфері кадру, і це відповідає одному пікселю екрана. Вони можуть бути кольоровими (і, можливо, освітленими). Крім того, необхідно визначити видимий, ближчий до спостерігача фрагмент, у разі перекриття полігонів. Для цього так званого визначення прихованої поверхні визначення прихованої поверхні зазвичай використовується Z – буфер. Колір фрагмента залежить від освітленості, текстури та інших матеріальних властивостей видимого примітиву і часто інтерполюється за допомогою властивостей вершини трикутника. Якщо є, шейдер фрагмента (також називають Pixel Shader'ом) виконується на етапі растрування для кожного фрагмента об'єкта. Якщо фрагмент видно, його тепер можна змішати з уже наявними значеннями кольорів на зображенні, якщо використовується прозорість або багатопробність. На цьому кроці один або кілька фрагментів перетворюються на піксель.

Щоб уникнути того, що користувач бачить поступову растеризацію примітивів, відбувається подвійна буферизація. Растеризація проводиться в спеціальній області пам’яті. Після того, як зображення було повністю растровано, воно копіюється у видиму область пам'яті зображення.

Шейдер

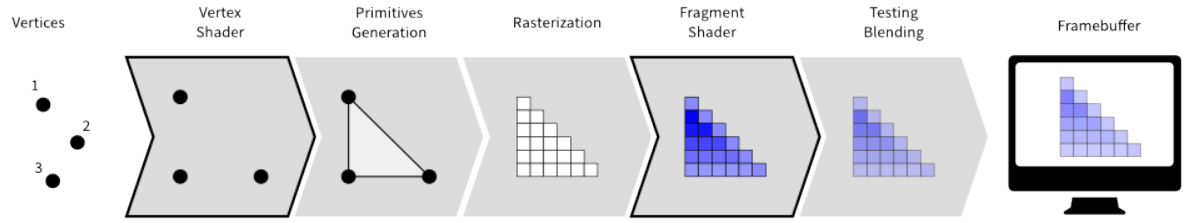


Рис. 2.14. Схема роботи шейдерів

У комп’ютерній графіці шейдер - це тип комп’ютерної програми, який спочатку використовувався для затінення в тривимірних сценах (отримання відповідних рівнів світла, темряви та кольору на відтвореному зображенні) (рис. 2.14). Зараз вони виконують різноманітні спеціалізовані функції в різних областях, що належать до категорії спеціальних ефектів комп’ютерної графіки, або ж виконують обробку відео, не пов’язану із затіненням, або навіть виконують функції, не пов’язані з графікою.

Традиційні шейдери обчислюють ефекти візуалізації на графічному обладнанні з високим ступенем гнучкості. Більшість шейдерів кодуються для (і працюють на) графічному процесорі (GPU), хоча це не є суворою вимогою. Мови шейдерів використовуються для програмування конвеєра візуалізації графічного процесора, який здебільшого замінив конвеєр з фіксованою функцією минулого, що дозволяв лише загальні функції перетворення геометрії та піксельні затінення; з шейдерами можна використовувати індивідуальні ефекти. Положення та колір (відтінок, насиченість, яскравість та контраст) усіх пікселів, вершин та / або текстур, що використовуються для побудови кінцевого відтвореного зображення, можуть бути змінені за допомогою алгоритмів, визначених у шейдері, і можуть бути змінені

зовнішніми змінними або текстурами представлений комп’ютерною програмою, що викликає шейдер.

Шейдери широко використовуються в пост-обробці кінотеатрів, комп'ютерних зображеннях та відеоіграх для створення ряду ефектів. Окрім простих моделей освітлення, більш складне використання шейдерів включає: зміну відтінку, насиченості, яскравості (HSL / HSV) або контрасту зображення; створення розмиття, світлового цвітіння, об'ємне освітлення, нормальне відображення (для глибинних ефектів), затінення, постеризація, відображення променевих ударів, спотворення, кольоровість (для так званих ефектів "блюз-екран / зелений екран"), виявлення краю та руху, як а також психоделічні ефекти, такі як ті, що спостерігаються в різних демосценах.

2.5 Рушій фізики

Фізичні двигуни відповідають за те, щоб з часом з’ясувати, де знаходиться кожен об’єкт у сцені. Об'єкти можуть зіткнутися один з одним, а потім вибрати відповідь на зіткнення кількома способами. Це загальна проблема, яку користувач може налаштувати на декількох різних рівнях.

Динаміка

Динаміка полягає в обчисленні, де нові положення об’єктів, базуючись на їх швидкості та прискоренні. Є чотири кінематичні рівняння разом із трьома законами Ньютона, які описують рух предметів. Ми будемо використовувати лише перше та третє кінематичні рівняння, інші корисніші для аналізу ситуацій, а не для симуляції. Це залишає нам:

$$v = v_0 + at$$

$$\Delta x = v_0 t + \frac{1}{2} a t^2$$

Ми можемо надати собі більше контролю, використовуючи другий закон Ньютона, подаючи прискорення, що дає нам:

$$v = v_0 + \frac{F}{m} t$$

$x = x_0 + vt$

Кожен об’єкт повинен зберігати ці три властивості: швидкість (v), масу (m) та чиста силу (net force, F). Тут ми знаходимо перше рішення, яке ми можемо прийняти щодо дизайну рушія, сила може бути або списком, або окремим вектором. У розв’язанні прикладних задач ми зазвичай складаємо діаграми сил і підсумовуємо сили, маючи на увазі, що нам слід зберігати список. Це призводить до того, що ви маємо можливість встановити силу, але нам потрібно буде її видалити пізніше, що може бути недоліком.

Отже, чиста сила (F) - це дійсно загальна сила, що застосовується в одному кадрі, тому ми можемо використовувати вектор і очистити його в кінці кожного оновлення. Це дозволяє користувачеві застосовувати силу, додаючи її, але видаляти це автоматично. Це скорочує наш код і дає збільшення продуктивності, оскільки не відбувається підсумовування сил, це і є загальна сума сил.

Колізія та виявлення колізії

Виявлення зіткнень вимагає більше обчислень, порівняно з динамікою, але ми можемо полегшити навантаження, використовуючи деякі розумні трюки. Давайте подумаємо, що потрібно спершу знайти. Якщо ми розглянемо деякі приклади зіткнення предметів, то помітимо, що в більшості випадків на кожній фігурі є найточніша точка всередині іншої.

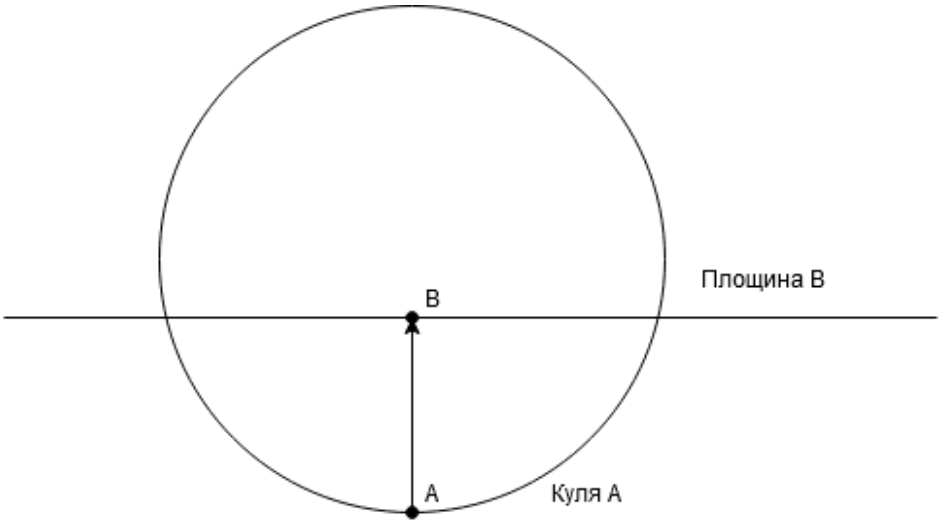


Рис. 2.15. Схема колізії кулі та площини

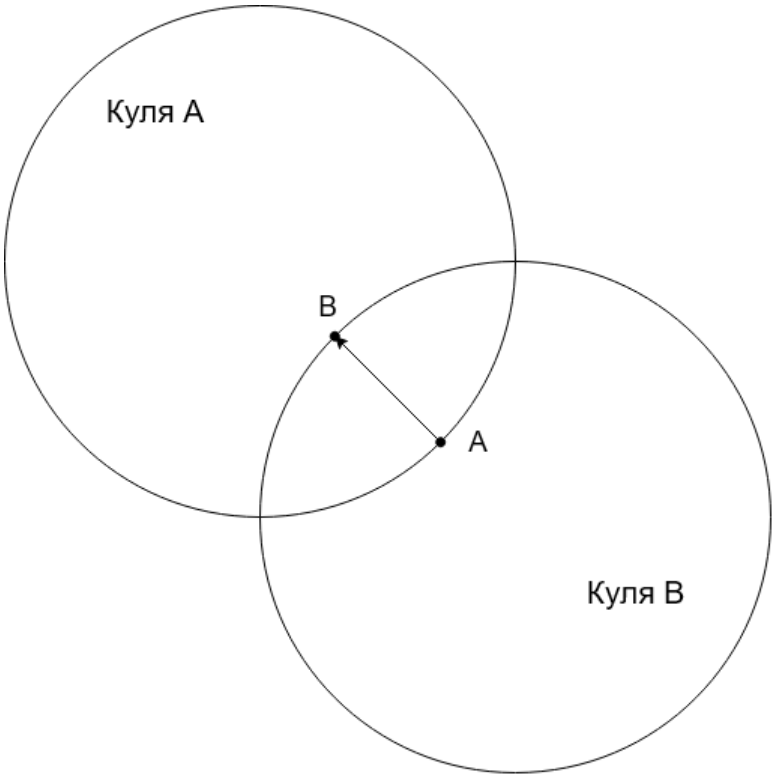


Рис. 2.16. Схема колізії двох куль

Це виявляється все, що нам потрібно, щоб відповісти на зіткнення. З цих двох точок ми можемо знайти нормаль і те, наскільки глибоко предмети знаходяться один в одному. Це величезне, оскільки це означає, що ми можемо абстрагувати ідею різних форм і турбуватися лише про моменти у відповіді.

Кожна фігура матиме різний тип колайдера, який зберігатиме свої властивості, і основу, що дозволяє зберігати їх. Будь-який тип колайдера повинен мати можливість перевірити на зіткнення з будь-яким іншим типом, тому ми додамо функції в основу для кожного з них. Ці функції будуть приймати трансформації, тому коллайдери можуть використовувати відносні координати.

Відповідь колізії

У контексті класичного моделювання механіки та фізичних двигунів, що застосовуються у відеоіграх, реакція на зіткнення стосується моделей та

алгоритмів для моделювання змін у русі двох твердих тіл після зіткнення та інших форм контакту.

Два твердих тіла (Rigidbody) в необмеженому русі, потенційно під дією сил, можуть бути змодельовані шляхом вирішення їх рівнянь руху за допомогою чисельних методів інтегрування. Під час зіткнення кінетичні властивості двох таких тіл, здається, миттєво змінюються, в результаті чого тіла відскакують одне від одного, ковзають або осідають у відносному статичному контакті, залежно від еластичності матеріалів та конфігурації зіткнення (рис. 2.17).

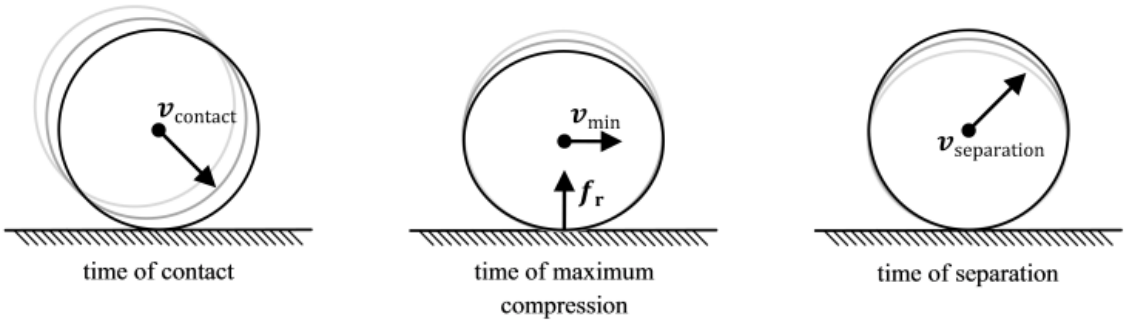


Рис. 2.17. Схема фаз стиснення та розширення колізії двох твердих тіл

Походження явища відскоку, або реакції, можна простежити за поведінкою реальних тіл, які, на відміну від своїх абсолютно жорстких ідеалізованих аналогів, під час зіткнення зазнають незначного стиснення з подальшим розширенням до поділу. Фаза стиснення перетворює кінетичну енергію тіл у потенційну енергію і певною мірою тепло. Фаза розширення перетворює потенційну енергію назад у кінетичну.

Під час фаз стиснення та розширення двох тіл, що стикаються, кожне тіло генерує реактивні сили на інше в точках контакту, такі що сумарні сили реакцій одного тіла за величиною рівні, але протилежні за напрямком силам іншого, як відповідно до ньютонівського принципу дії та реакції. Якщо ігнорувати ефекти тертя, розглядається зіткнення, що впливає лише на

компонент швидкостей, які спрямовані вздовж контакту в нормальному режимі, і залишаючи тангенціальні компоненти без змін.

Реакція

Ступінь відносної кінетичної енергії, що зберігається після зіткнення, що називається відновленням, залежить від еластичності матеріалів тіла. Коефіцієнт відновлення між двома заданими матеріалами моделюється як відношення $e \in [0..1] e \in [0..1]$ відносної швидкості після зіткнення точки дотику вздовж нормалі контакту щодо відносної швидкості попереднього зіткнення тієї ж точки вздовж тієї ж нормалі. Ці коефіцієнти, як правило, визначаються емпірично для різних пар матеріалів, таких як дерево проти бетону або гума проти дерева. Значення для е, близькі до нуля, вказують на нееластичні зіткнення, такі як шматок м'якої глини, що вдаряється об підлогу, тоді як значення, близькі до одного, представляють високо-еластичні зіткнення, такі як гумова кулька, що відбивається від стіни. Втрати кінетичної енергії відносно одного тіла відносно іншого. Таким чином, загальний імпульс обох тіл щодо деякого загального посилання незмінний після зіткнення, відповідно до принципу збереження імпульсу.

Тертя

Іншим важливим явищем контакту є тертя "поверхня-до-поверхні" – сила, яка перешкоджає відносному руху двох поверхонь, що контактують, або руху тіла в рідині. У цьому розділі ми обговорюємо тертя поверхні до поверхні двох тіл у відносному статичному контакті або ковзанні. У реальному світі тертя обумовлено недосконалою мікроструктурою поверхонь, виступи яких з’єднуються між собою, створюючи реактивні сили, дотичні до поверхонь (рис. 2.18).

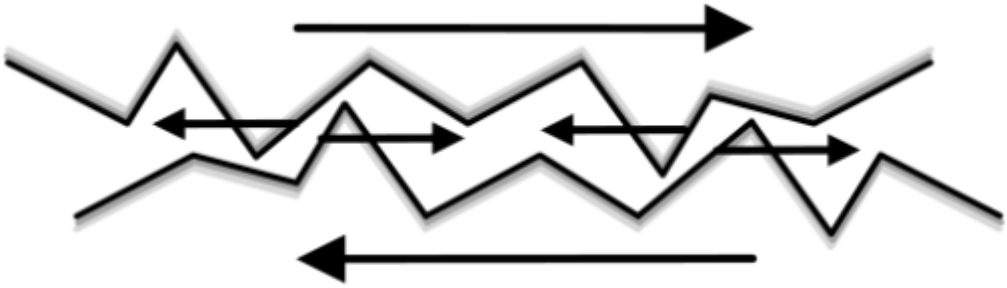


Рис. 2.18. Спрощена схема тертя двох поверхонь

Щоб подолати тертя між двома тілами при статичному контакті, поверхні повинні якимось чином віддалятися одна від одної. Потрапляючи в рух, ступінь спорідненості поверхні зменшується, і, отже, тіла при ковзному русі, як правило, надають менший опір руху. Ці дві категорії тертя відповідно називають статичним та динамічним тертям.

Прикладена сила

Прикладена сила – це сила, яка застосовується до об’єкта іншим об’єктом або особою. Напрямок прикладеної сили залежить від того, як застосовується сила.

Сила нормалі

Сила нормалі – це сила опори, що діє на об'єкт, який контактує з іншим стійким об'єктом. Силу нормалі іноді називають силою натискання, оскільки її дія тисне на поверхню. Звичайна сила завжди спрямована на об'єкт і діє перпендикулярно прикладеній силі.

Сила тертя

Сила тертя - це сила, яку чинить поверхня, коли предмет рухається по ній або докладає зусиль, щоб рухатись по ній. Сила тертя протистоїть руху предмета. Тертя виникає, коли дві поверхні стиснуті між собою, викликаючи привабливі міжмолекулярні сили між молекулами двох різних поверхонь. Таким чином, тертя залежить від природи двох поверхонь і від ступеня їх

стиснення. Тертя завжди діє паралельно поверхні, що контактує, і протилежному напрямку руху. Силу тертя можна розрахувати за допомогою рівняння.

Імпульсна контактна модель

Сила $f(t) \in R^3$ $f(t) \in R^3$, залежно від часу $t \in R$ $t \in R$, діючи на тіло з передбачуваною постійною масою $m \in R$ $m \in R$ для інтервалу часу $[t_0..t_1]$ $[t_0..t_1]$ генерує зміну імпульсу тіла $\mathbf{p}(t) = mv(t)\mathbf{p}(t) = mv(t)$, де $\mathbf{v}(t)\mathbf{v}(t)$ – це результуюча зміна швидкості. Таким чином, зміна імпульсу, що називається імпульсом і позначається $j \in R^3$ $j \in R^3$, обчислюється як:

$$j = \int_{t_0}^{t_1} \mathbf{f} dt$$

Для фіксованого імпульсу j j рівняння передбачає, що $t_1 \to t_0 \Rightarrow |\mathbf{f}| \to \infty$ $t_1 \to t_0 \Rightarrow |\mathbf{f}| \to \infty$, тобто менший інтервал часу повинен бути компенсований більшою силою реакції для досягнення того самого імпульсу. При моделюванні зіткнення між ідеалізованими твердими тілами недоцільно моделювати фази стиснення та розширення геометрії тіла за інтервал часу зіткнення. Однак, припускаючи, що можна знайти силу f f , яка дорівнює 0 скрізь, крім як при t_0 t_0, і такий, що обмеження:

$$\lim_{t_1 \to t_0} \int_{t_0}^{t_1} \mathbf{f} dt$$

існує і дорівнює j j , поняття миттєвих імпульсів може бути введене для імітації миттєвої зміни швидкості після зіткнення.

Модель реакції на основі імпульсу

Вплив сили реакції $f_r(t) \in R^3$ $f_r(t) \in R^3$ за інтервал зіткнення $[t_0..t_1]$ $[t_0..t_1]$, отже, може бути представлений миттєвим імпульсом реакції $j_r(t) \in R^3$ $j_r(t) \in R^3$, обчислюється як:

$$j_r = \int_{t_0}^{t_1} \mathbf{f}_r dt$$

Виведенням з принципу дії та реакції, якщо імпульс зіткнення, поданий першим тілом на друге тіло в точці контакту $p_r \in R^3$ $p_r \in R^3$, лічильник

імпульсу, який подає друге тіло на перше, $-j_r - j_r$. Розкладання $\pm j_r = \pm j_r n$ $\pm j_r = \pm j_r n$ на величину імпульсу $j_r \in R$ $j_r \in R$ та напрямок уздовж нормальної точки контакту $n\ n$ та його заперечення $-n\ -n$ дозволяє вивести формулу для обчислення зміни лінійних та кутових швидкостей тіл, що виникають внаслідок імпульсів зіткнення. У подальших формулах $n\ n$ завжди передбачається спрямовувати в сторону від тіла 1 і до тіла 2 в точці контакту (рис. 2.19).

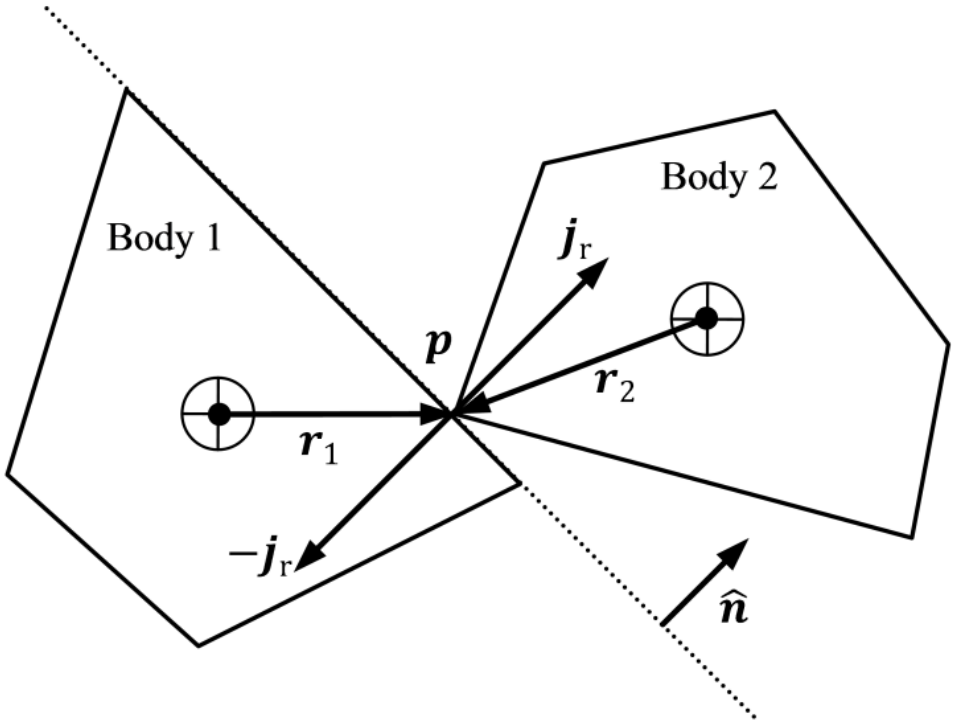


Рис. 2.19. Схема застосування імпульсів в точці зіткнення

Припускаючи, що величина імпульсу зіткнення $j_r\, j_r$ задана і за допомогою законів руху Ньютона співвідношення між тілами пре- та пост-лінійним швидкостями випливає наступне:

$$v_1' = v_1 - \frac{j_r}{m_1} n \quad v_1' = v_1 - \frac{j_r}{m_1} n \tag{1a}$$

$$v_2' = v_2 - \frac{j_r}{m_2} n \quad v_2' = v_2 - \frac{j_r}{m_2} n \tag{1b}$$

де для і-го тіла $v_i \in R^3\, v_i \in R^3$—лінійна швидкість перед зіткненням, $v_i' \in R^3$ $v_i' \in R^3$—це лінійна швидкість після зіткнення.

Аналогічно для кутових швидкостей:

$$\omega_1^{'} = \omega_1 - j_r I_1^{-1}(r_1 * n)\omega_1^{'} = \omega_1 - j_r I_1^{-1}(r_1 * n) \quad (2a)$$

$$\omega_2^{'} = \omega_2 - j_r I_2^{-1}(r_2 * n)\omega_2^{'} = \omega_2 - j_r I_2^{-1}(r_2 * n) \quad (2b)$$

де для і-го тіла $\omega_i \in R^3 \omega_i \in R^3$ — кутова швидкість перед зіткненням, $\omega_i^{'} \in R^3$ $\omega_i^{'} \in R^3$ – кутова швидкість після зіткнення, $I_i \in R^3 * 3\ I_i \in R^3 * 3$ – тензор інерції у точці зору світу і $r_i \in R^3 r_i \in R^3$ – зміщення спільної контактної точки р від центру маси.

Швидкості $v_{p1} v_{p1}$, $v_{p2} \in R^3 v_{p2} \in R^3$ тіл у точці контакту можна обчислити у відповідних лінійних та кутових швидкостях, використовуючи:

$$v_{pi} = v_i + \omega_i * r_i v_{pi} = v_i + \omega_i * r_i \quad (3)$$

для $i = 1,2\ i = 1,2$. Коефіцієнт відновлення $e\ e$ відноситься до відносної швидкості перед зіткненням $v_r = v_{p2} - v_{p1} v_r = v_{p2} - v_{p1}$ контакту вказують на відносну швидкість після зіткнення $v_r^{'} = v_{p2}^{'} - v_{p1}^{'} v_r^{'} = v_{p2}^{'} - v_{p1}^{'}$ уздовж контакту нормалі $n\ n$ наступним чином:

$$v_r^{'} * n = -ev_r * n\ v_r^{'} * n = -ev_r * n \quad (4)$$

Підставивши рівняння (1a), (1b), (2a), (2b) та (3) у рівняння (4) та вирішивши величину імпульсу реакції $j_r, j_r,$ вийде:

$$j_r = \frac{-(1+e)v_r * n}{m_1^{-1} + m_2^{-1} + \left(I_1^{-1}(r_1 * n) * r_1 + I_2^{-1}(r_2 * n) * r_2\right) * n}$$

$$j_r = \frac{-(1+e)v_r * n}{m_1^{-1} + m_2^{-1} + \left(I_1^{-1}(r_1 * n) * r_1 + I_2^{-1}(r_2 * n) * r_2\right) * n} \quad (5)$$

Обчислення реакції на основі імпульсу

Таким чином, процедура обчислення лінійних швидкостей після зіткнення $v_j^{'} v_j^{'}$ та кутових швидкостей $\omega_j^{'} \omega_j^{'}$ має такий вигляд:

1. Обчислити величину імпульсу реакції $j_r j_r$ за допомогою $v_r, m_1, m_2, I_1, I_2, r_1, r_2, n\ v_r, m_1, m_2, I_1, I_2, r_1, r_2, n$ і $n\ n$ використовуючи рівняння (5)

2. Обчислити вектор імпульсу реакції $j_r j_r$ з точки зору величини $j_r j_r$ та контактну нормаль $n\ n$, використовуючи $j_r = j_r n\ j_r = j_r n$.
3. Обчислити нові лінійні швидкості $v_i^{'} v_i^{'}$ з точки зору старих швидкостей $v_i v_i$ та вектор імпульсу реакції $j_r j_r$ за допомогою рівнянь (1a) та (1b)
4. Обчислити нові кутові швидкості $\omega_i^{'} \omega_i^{'}$ з точки зору старих кутових швидкостей $\omega_i \omega_i$, тензори інерції $I_i I_i$ та імпульс реакції $j_r j_r$, використовуючи рівняння (2a) та (2b)

GJK Алгоритм

Алгоритм відстані Гілберта – Джонсона – Кірті - це метод визначення мінімальної відстані між двома опуклими множинами. На відміну від багатьох інших алгоритмів відстані, він не вимагає, щоб дані геометрії зберігалися у будь-якому конкретному форматі, а натомість покладається виключно на функцію підтримки, щоб ітеративно генерувати наближені спрощені до правильної відповіді, використовуючи перешкоду простору конфігурації (CSO) двох опуклих фігур , більш відомий як різниця Мінковського.

Алгоритми "Розширеного GJK" використовують інформацію про край, щоб пришвидшити алгоритм, слідуючи за ребрами при пошуку наступного симплексу. Це значно покращує продуктивність для багатогранників з великою кількістю вершин.

GJK використовує субалгоритм відстані Джонсона, який в загальному випадку обчислює точку тетраедра, найближчу до початку координат, але, як відомо, страждає від проблем числової стійкості. У 2017 році Монтанарі, Петрініч та Барб'єрі запропонували новий субалгоритм, заснований на підписаних обсягах, який дозволяє уникнути множення потенційно малих кількостей і домогтися пришвидшення від 15% до 30%.

Алгоритми GJK часто використовуються поступово в системах моделювання та відеоіграх. У цьому режимі остаточний симплекс з попереднього рішення використовується як початковий здогад у наступній

ітерації, або "кадрі". Якщо позиції в новому кадрі близькі до позицій у старому кадрі, алгоритм буде збігатися за одну або дві ітерації. Це дає системи виявлення зіткнень, які працюють майже в постійний час.

Стійкість, швидкість та невеликий розмір алгоритму роблять його популярним для виявлення зіткнень у реальному часі, особливо у фізичних двигунах для відеоігор.

Тестування на зіткнення між сферами неважко, оскільки в системі є лише дві точки. Це залишає нам єдиний вектор, який ми можемо порівняти із сумою їх радіусів, щоб визначити, чи є зіткнення.

З багатокутниками ми не можемо зробити такі спрощення. Вони зроблені з декількох вершин, видаляючи будь-який очевидний спосіб пошуку їх відстані та властивості чіткого радіуса для порівняння. Нам потрібен розумніший спосіб тестування на зіткнення.

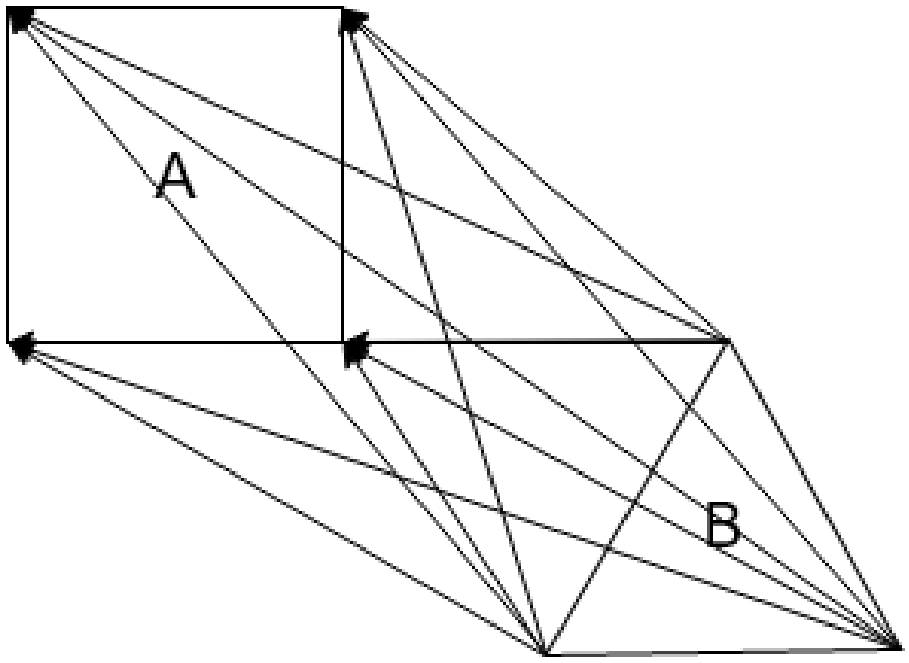


Рис. 2.20. Схема різниці Мінковського

Віднімання двох багатокутників з однаковою кількістю вершин просто, але якщо ми хочемо підтримувати різні полігони, нам потрібно відняти

кожну вершину від кожної вершини іншого багатокутника (рис. 2.20). Оскільки існує кілька вершин, ми не маємо одного вектора, а багатьох, що утворюють інший багатокутник. Це призводить до хмари вершин чисел $A * B\ A * B$, яку нам потрібно обробити далі, щоб вибрати зовнішню опуклу оболонку.

Цей зовнішній корпус відомий як різниця Мінковського. Він представляє відстань між кожною точкою двох багатокутників. Ми використаємо його для перетворення двох багатокутників в один, який ми зможемо проаналізувати для виявлення зіткнення. Ключ у тому, що якщо початок координат знаходиться всередині різниці, то мали бути дві точки, що віднімаються до 0; тобто десь є перекриття.

Різниця Мінковського приємна для візуалізації, але занадто дорога для обчислення в режимі реального часу; нам потрібен спосіб спростити це.

Алгоритм GJK стосується лише зовнішньої оболонки нашої хмари вершин, тому це суттєво пришвидшиться, якщо ми зможемо скоротити час, витрачений на їх пошук. Давайте подумаємо, що ставить вершину на корпус. Якщо ми придивимося уважніше, зауважимо, що ці вершини мають найбільш екстремальні компоненти. Вони дісталися до своїх місць відніманням між двома іншими вершинами, тому, щоб одна була найекстремальнішою, вона, мабуть, походила з найбільш екстремальних вершин вихідних багатокутників. Якщо ми визначимо "найекстремальніший" як найдальший у якомусь напрямку, ми можемо пограти з математикою, щоб отримати цю швидкість збільшення.

Знаходження найдальшої вершини здійснюється шляхом перебору по множині вершин і знаходження тієї, що має найбільший крапковий добуток у напрямку. Нехай $\vec{D}\vec{D}$ буде напрямком, а $A - B\ A - B$ – хмарою вершин.

$$\max\left\{\vec{D} * (A - B)\right\}$$

Обчислення $A - B$ займало $A * B$ кроків; що робить цю функцію операцією $O(n^2)$. Але, ми можемо розподіляти і ніколи не обчислювати повну різницю.

Якщо ми розподілимо крапковий добуток та функцію max, у нас залишиться таке:

$$\max\left\{\left(\vec{D} * A\right) - \left(\vec{D} * B\right)\right\}$$

$$\max\left\{\left(\vec{D} * A\right)\right\} - \max\left\{\left(-\vec{D}\right) * B\right\}$$

Тепер нам потрібні лише кроки $A + B$, що перетворить нашу квадратичну функцію часу в лінійну.

Нам потрібно змінити напрямок для A, коли ми розподіляємо max, оскільки хочемо зберегти максимальне значення. Ми хочемо, щоб найменшу крайню вершину від B відняли від самої крайньої вершини з A

Мета алгоритму GJK – визначити, чи походження знаходиться в межах різниці Мінковського. Це було б легко, але ми викинули повну різницю заради ефективності. У нас є лише функція підтримки, яка дає нам по одній вершині за раз. Нам потрібно ітеративно шукати та будувати те, що називають симплексом, навколо походження.

Симплекс визначається як фігура, яка має $N + 1$ кількість вершин, причому N є кількістю розмірностей. Практично це представляє найпростішу форму, яка може «вибрати» регіон у просторі. Наприклад, 2D трикутник - це найпростіша фігура, яка може вибрати область, що містить певну точку. Ці фігури мають прості тести, за допомогою яких ми можемо визначити, яка вершина, ребро чи грань найближчі до початку координат. Залежно від того, яка функція найближча, ми будемо вилучати, додавати або міняти місцями, щоб зробити симплекс ближчим до початку координат. Якщо ми виявимо, що найближча характеристика вже є найближчою з можливих, але початок відсутній всередині, ми знаємо, що зіткнення немає.

В іншому випадку, якщо ми знайдемо початок всередині симплексу, ми знаємо, що сталося зіткнення.

Ми бачимо, що нам потрібно мати справу з двома випадками: пряма і трикутник. Нам потрібен ще один випадок у формі тетраедра, щоб вибрати об’єкт, якщо ми хочемо виявити 3D зіткнення.

Ми почнемо з прямою. Є три можливі регіони, в яких може бути точка початку, але реально лише два регіони. Ми почали з точки B і шукали в напрямку A, що означає, що початок координат не може бути в червоній області. Це залишає нам одну перевірку між векторами AB і AO. Якщо AO знаходиться всередині зеленої області, ми рухаємось далі. Якщо AO знаходиться у синій області, ми повернемося до справи, але B буде замінено (рис. 2.21).

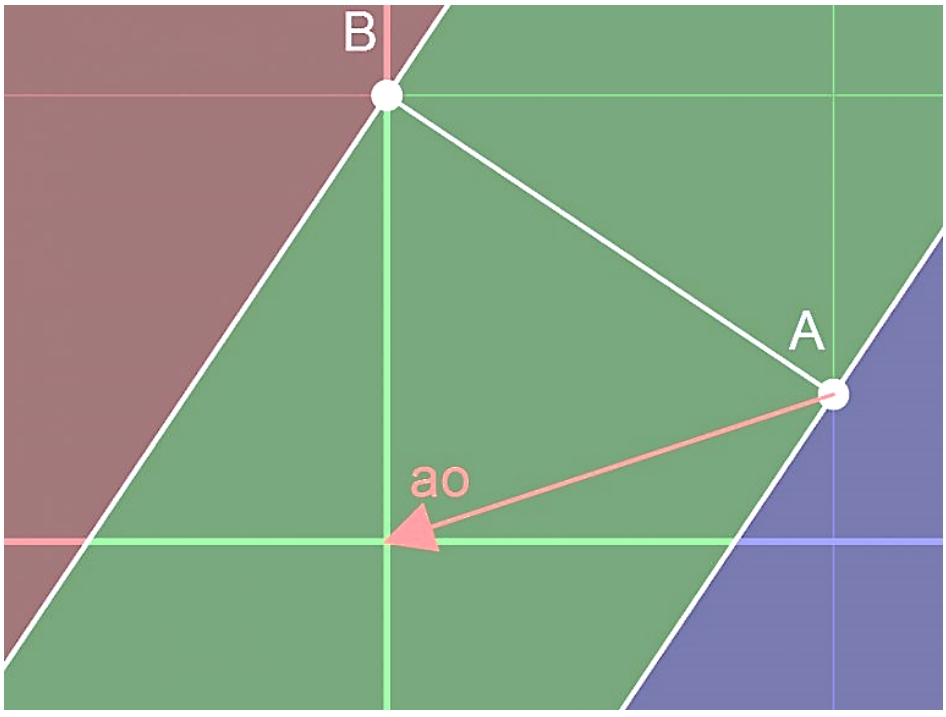


Рис. 2.21. Схема GJK алгоритму для прямої

У цьому випадку AO рухається в тому ж напрямку, що і AB, тому ми знаємо, що це в зеленій області. Ми встановимо напрямок пошуку, що вказує на початок, і рухаємось далі. У 2D нам не потрібно буде використовувати

перехресні продукти, але в 3D походження може бути де завгодно в циліндрі навколо лінії, тому вони нам потрібні, щоб отримати правильний напрямок.

Трикутник має сім областей, але знову ж таки ми можемо вирішити деякі неможливості. Жовтий, червоний та фіолетовий не можуть мати початок координат, оскільки новою точкою, яку ми додали, була A, що означає, що початок координат не може бути в напрямку грані до BC. Це залишає нам чотири регіони, які нам потрібно перевірити (рис. 2.22).

Якщо початок координат знаходиться за межами трикутника на грані AC, ми перевіримо, чи воно також у напрямку AC. Якщо це так, тоді ми видалимо B із симплексу і рухаємось далі, якщо ні, то зробимо ту саму перевірку між AB. Якщо початок координат не був спрямований до грані AC, ми перевіримо грань AB. Якщо він є, ми зробимо той самий випадок між AB. Нарешті, якщо обидві перевірки не вдаються, ми знаємо, що це має бути всередині трикутника. У 2D ми б закінчили і результат тесту був би позитивним, але в 3D нам потрібно перевірити, чи є початок координат вище чи нижче трикутника, і рухатися далі.

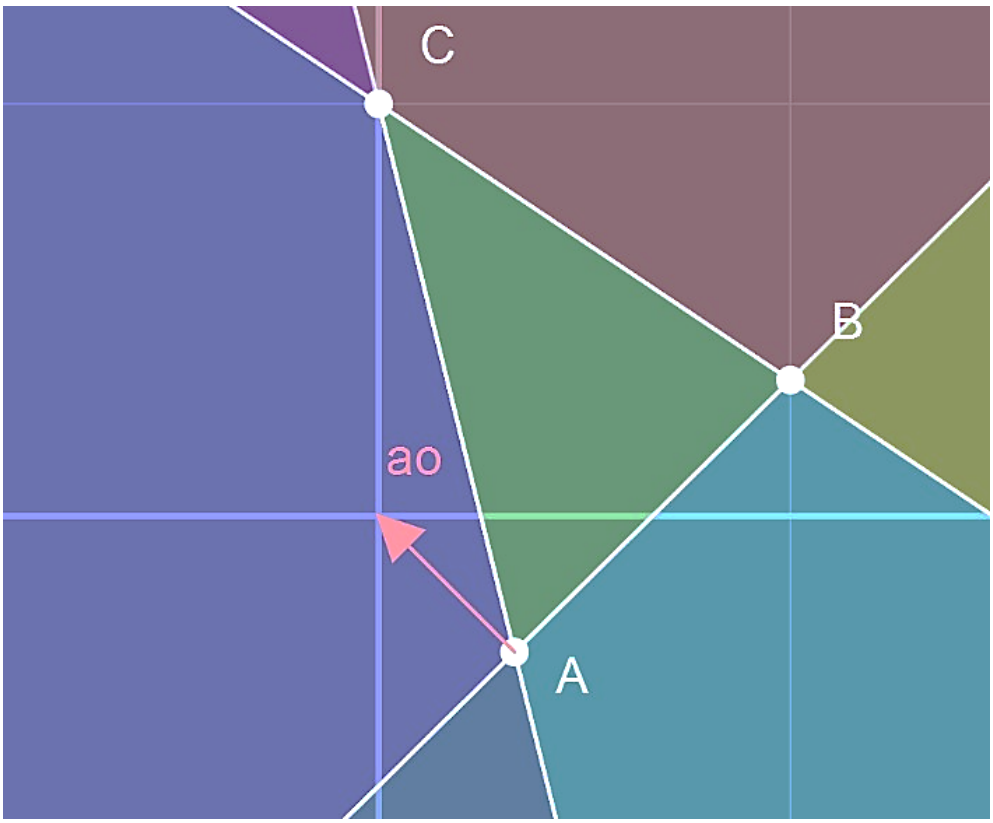


Рис. 2.22. Схема алгоритму GJK для трикутника

Випадок тетраедра є найскладнішим, але майже повністю складається з випадків трикутників. Нам не потрібно перевіряти початок координат нижче тетраедра з тієї самої причини, що і раніше. Нам потрібно лише визначити в напрямку до якої грані, якщо вона є, початок координат. Якщо вони є, ми повернемося до випадку трикутника з цією гранню як симплекс, але якщо ні, ми знаємо, що він повинен бути всередині тетраедра, і результат тесту буде позитивним (рис. 2.23).

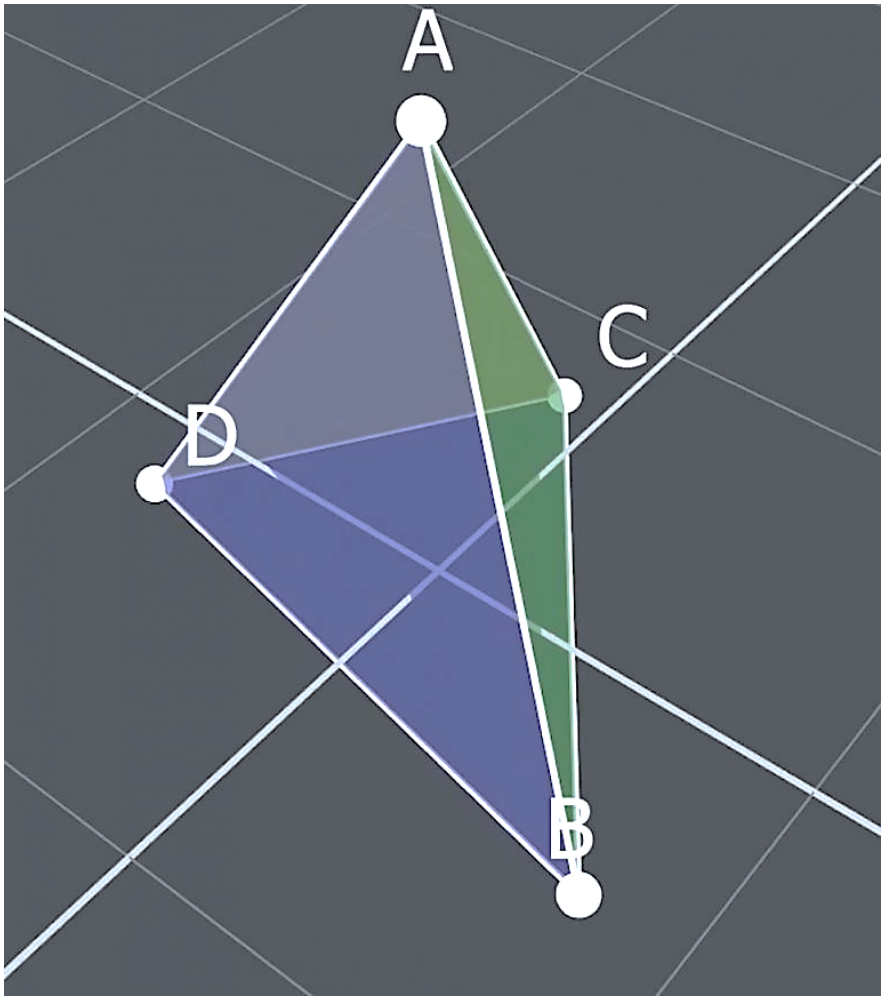


Рис. 2.23. Схема алгоритму GJK для тетраедра

2.6 Система обліку часу

За визначенням, дельта-час – це час завершення в секундах з моменту останнього кадру. Це допомагає нам зробити незалежну частоту кадрів програми. Тобто, незалежно від FPS, програма буде виконуватися з однаковою швидкістю.

Припустимо, наша програма працює зі швидкістю 60 кадрів в секунду. Це означає, що програма оновлюється 60 разів на секунду, іншими словами, є 60 кадрів.

Примітка: FPS означає кадри в секунду

Тепер давайте подивимось на час між кадрами (рис. 2.24):

По-перше, після початку симуляції, кадр 1 виконується, потім кадр 1 закінчує, і кадр 2 виконується. дельта-час почав обчислювати, коли закінчується кадр 1 і закінчував обчислювати, коли закінчується кадр 2.

Рис. 2.24. Схема кроку дельта-часу

- Перед виконанням кадру 1 час дельти дорівнює 0
- Потім додаток продовжується, а кадр 2 виконується. Отже, час між цими 2 кадрами становить 0,05 секунди.
- Отже, час дельти становить 0,05 секунди. Тому ми називаємо це часом, що минув з останнього кадру.

У програмі, фізичний рушій використовує дельта-час в якості способу просунути вперед симуляцію фізики на деякий невеликий проміжок часу.

Але як вибрати цю дельта-величину часу? Це може здатися тривіальною

темою, але насправді існує безліч різних способів це зробити, кожен із яких має свої сильні та слабкі сторони.

Фіксоване значення дельта-часу

Найпростіший спосіб зробити крок вперед - це фіксований дельта-час, наприклад 1/60 секунди:

```
double t = 0.0;
double dt = 1.0 / 60.0;

while ( !quit )
{
   physics_step( state, t, dt );
   render( state );
   t += dt;
}
```

Багато в чому цей код є достатнім. Якщо користувачу пощастило, щоб дельта-час відповідав частоті оновлення дисплею, то можна переконатися, що цикл оновлення займає менше одного кадру в реальному часі, тоді вже є ідеальне рішення для оновлення фізичної симуляції.

Але в реальному світі ми можемо не знати швидкість оновлення дисплея заздалегідь. VSYNC може бути вимкнено, або ми можемо працювати на повільному комп'ютері, який не може оновити та зробити кадр досить швидко, щоб представити його зі швидкістю 60 кадрів в секунду.

У цих випадках симуляція буде працювати швидше або повільніше, ніж передбачено.

Змінне значення дельта-часу

Виправити це здається просто. Просто виміряємо, скільки часу займає попередній кадр, а потім повернемо це значення назад як дельта-час для наступного кадру. Звичайно, це має сенс, адже якщо комп’ютер занадто повільно оновлюється на частоті 60 Гц і йому доводиться падати до 30 кадрів в секунду, ми автоматично пройдемо до 1/30 як час дельти. Те саме, що стосується частоти оновлення дисплея 75 Гц замість 60 Гц або навіть

випадку, коли VSYNC (Вертикальна синхронізація) вимкнена на швидкому комп’ютері:

```
double t = 0.0;
double currentTime = hires_time_in_seconds();

while ( !quit )
{
   double newTime = hires_time_in_seconds();
   double frameTime = newTime - currentTime;
   currentTime = newTime;

   integrate( state, t, frameTime );
   t += frameTime;

   render( state );
}
```

Але з цим підходом існує велика проблема. Проблема полягає в тому, що поведінка фізичної симуляції залежить від дельтового часу, який ми пропускаємо. Ефект може бути тонким, оскільки програма має дещо інше "відчуття" залежно від частоти кадрів, або ефект може бути таким екстремальним, як симуляція пружини, що вибухає до нескінченності, швидко рухаються предмети, що пробиваються крізь стіни, а об’єкти сцени падають крізь підлогу.

Одне можна сказати напевно, але це зовсім нереально сподіватися, що симуляція правильно обробляє будь-який дельта-час, що в неї потрапив. Щоб зрозуміти, чому, подумайте, що сталося б, якби ми пропустили 1/10 секунди як дельта-час? Як щодо однієї секунди? 10 секунд? 100? Зрештою ми натрапимо на фатальну помилку програми.

Напівфіксований крок часу

Набагато реалістичніше сказати, що симуляція добре налаштована лише в тому випадку, якщо дельта-час менше або дорівнює якомусь максимальному значенню. Зазвичай це набагато простіше на практиці, ніж спроба зробити нашу імітацію стійкою до розривів при широкому діапазоні значень часу дельта.

Маючи ці знання під рукою, ось простий прийом, щоб ми ніколи не пропустили дельта-час, що перевищує максимальне значення, при цьому працюючи з правильною швидкістю на різних машинах:

```
double t = 0.0;
double dt = 1 / 60.0;

double currentTime = hires_time_in_seconds();

while ( !quit )
{
   double newTime = hires_time_in_seconds();
   double frameTime = newTime - currentTime;
   currentTime = newTime;

   while ( frameTime > 0.0 )
   {
      float deltaTime = min( frameTime, dt );
      integrate( state, t, deltaTime );
      frameTime -= deltaTime;
      t += deltaTime;
   }

    render( state );
}
```

Перевага цього підходу полягає в тому, що ми тепер маємо верхню межу дельта-часу. Ця межа ніколи не перевищує цього значення, тому що якщо вона є, ми поділяємо крок часу. Недоліком є те, що ми зараз робимо кілька кроків за оновлення дисплею, включаючи один додатковий крок, щоб витратити залишок часу кадру, який не ділиться на дельта-час. Це не проблема, якщо ми тільки обмежені обробкою графіки, але якщо симуляція є найдорожчою частиною нашого кадру, ми можемо зіткнутися з так званою "спіраллю смерті".

Що таке спіраль смерті? Це те, що відбувається, коли фізична симуляція не може встигнути за кроками, які її просять зробити. Наприклад, якщо симуляції сказано: «Добре, будь-ласка, змоделюй X секунд фізики», і якщо це займає Y секунд реального часу, коли Y>X, то не займе багато часу, щоб зрозуміти, що з часом симуляція відстає. Це називається спіраллю смерті, оскільки відставання змушує оновлення фізики імітувати більше

кроків, щоб наздогнати, що у свою чергу змушує симуляцію відставати далі і моделювати більше кроків ...

То як нам уникнути цього? Щоб забезпечити стабільне оновлення, нам потрібно залишити більше часу в запасі. Нам дійсно потрібно переконатися, що для оновлення фізичної симуляції на X секунд реального часу потрібно значно менше X секунд реального часу. Якщо ми можемо це зробити, тоді фізичний рушій може «наздогнати» будь-який тимчасовий стрибок, імітуючи більше кадрів. Крім того, ми можемо затискати максимальну кількість кроків на кадр, і симуляція, здається, сповільнюється під великим навантаженням. Можливо, це краще, ніж "спіраль до смерті", особливо якщо велике навантаження - це лише тимчасовий стрибок.

Фізика з дельта-часом

Тепер давайте зробимо ще один крок далі. Що робити, якщо нам необхідно точна відтворюваність від одного циклу до наступного з огляду на ті самі ведення? Це стає в нагоді при спробі з'єднати фізичну симуляцію за допомогою детермінованого кроку, але також, як правило, в якості переваги, наша симуляція поводиться абсолютно однаково від одного запуску до наступного, без будь-яких можливостей для різної поведінки залежно від частоти кадрів рендерингу.

Тоді нам необхідно отримати перевагу з обох сторін: фіксованого значення дельта-часу для моделювання та можливість рендерингу з різною частотою кадрів. Ці дві речі здаються абсолютно суперечливими, і вони є – якщо ми не знайдемо способу розділити симуляцію та візуалізацію частоти кадрів.

```
double t = 0.0;
const double dt = 0.01;

double currentTime = hires_time_in_seconds();
double accumulator = 0.0;

while ( !quit )
```

```
{
   double newTime = hires_time_in_seconds();
   double frameTime = newTime - currentTime;
   currentTime = newTime;

   accumulator += frameTime;

   while ( accumulator >= dt )
   {
      integrate( state, t, dt );
      accumulator -= dt;
      t += dt;
   }

   render( state );
}
```

2.7 UI в реальному часі

Графічний користувальницький інтерфейс (далі ГІ) безпосереднього режиму (GUI), також відомий як IMGUI – це графічний шаблон дизайну інтерфейсу користувача, який використовує графічну бібліотеку безпосереднього режиму для створення графічного інтерфейсу. Інший основний шаблон дизайну API у графічних бібліотеках - це збережений режим.

Більшість наборів віджетів графічного інтерфейсу користувача безпосереднього режиму реалізовано в системних елементах керування за замовчуванням та спеціальному рендерингу для розробки ігор, графічних додатків, бібліотеки: форма масштабу та Dear ImGui.

Для реалізації графічного інтерфейсу потрібно дотримуватися таких вимог:

- ГІ повинен оновлюватися синхронно із графічною сценою або складною графікою.
- ГІ повинен бути накладеним на графічну сцену або складну графіку (що особливо легко в обох випадках, коли і графічний інтерфейс, і графічна сцена контролюються програмним циклом).

- ГІ повинен мати незвичний зовнішній вигляд або бути прикрашеним складною графікою. Це означає, що в графічному інтерфейсі користувача безпосереднього режиму клієнтський код має власні примітиви візуалізації та дизайн API, що впливає на реалізацію графічного конвеєру.

Набір інструментів віджетів графічного інтерфейсу безпосереднього режиму:

- є більш зручнішим в тому сенсі, що дерево віджетів часто є деревом викликів функцій, що робить цей набір гнучким, але з яким важко взаємодіяти.
- є одночасно складним і легшим для розуміння (з точки зору меншої кількості неявних припущень на виклик API набору інструментів). Зазвичай це також призводить до меншої функціональності.
- є більш досконалим для створення та управління (як правило, потрібно більше викликів API набору інструментів), якщо це більше, ніж просто дерево віджетів, включаючи макет (абсолютне та відносне позиціонування, що стосується батьківських або дочірніх елементів).
- має менш витончене вилучення оклюзії (z-буферизація), тестування звернень, обробку змін стану, прокрутку та анімацію фокусування / гарячого контролю (віджет). Це також передбачає необхідність управління самим логічним деревом / візуальним деревом.
- повинен повністю перебудувати буфери вершин з нуля для кожного нового кадру.
- може постійно навантажувати процесор, якщо не використовує шейдери, завантажені на графічний процесор.

Набори інструментів графічного інтерфейсу безпосереднього режиму - хороший вибір для тих, хто віддає перевагу простому, легко мінливому та розширюваному графічному інтерфейсу. Зазвичай вони є загальними, з відкритим кодом та крос-платформні. Одним із способів отримати гнучкість і зручність графічного інтерфейсу безпосереднього режиму без недоліків

збереження дерева віджетів лише у викликах функцій, з відсутністю безпосереднього контролю за тим, як графічний графічний інтерфейс малюється в механізмі візуалізації, було б використання віртуального дерева віджетів , як і технологія React використовує віртуальний DOM.

РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ ПЛАТФОРМИ

3 Огляд засобів розробки

Мова програмування C++

C++ – це статично набрана, скомпільована, загальноприйнята, чутлива до регістру, мова програмування у вільній формі, що підтримує процедурне, об’єктно-орієнтоване та загальне програмування; вона розглядається як мова середнього рівня, оскільки містить комбінацію обох високо-рівневих та низько-рівневих мовних особливостей. Мова C++ була розроблена Бьярном Страструпом, починаючи з 1979 року в лабораторіях Bell в Мюррей-Хілл, штат Нью-Джерсі, як вдосконалення мови C і спочатку називалася C з класами, але пізніше була перейменована в C++ у 1983 році. C++ базується на мові програмування C, і практично будь-яка легальна програма C є легальною програмою C++.

Стандартні бібліотеки

Стандарт C++ складається з трьох важливих частин:

- Основна мова, що надає всі будівельні блоки, включаючи змінні, типи даних і літерали тощо.
- Стандартна бібліотека C++, що надає багатий набір функцій, що обробляють файли, рядки тощо.
- Стандартна бібліотека шаблонів (STL), що надає багатий набір методів маніпулювання структурами даних тощо.

Стандарт ANSI

Стандарт ANSI – це спроба переконатись, що C++ є портативним; цей код, який ви пишете для компілятора Microsoft, буде компілюватися без помилок, використовуючи компілятор на Mac, UNIX, вікні Windows або Alpha.

Стандарт ANSI є стабільним, і всі основні виробники компіляторів C++ підтримують стандарт ANSI.

Використання C++

C++ використовується сотнями тисяч програмістів практично в кожному домені програми.

C++ широко використовується для написання драйверів пристроїв та іншого програмного забезпечення, яке покладається на пряме маніпулювання апаратним забезпеченням у режимі реального часу.

C++ широко використовується для викладання та досліджень, оскільки він досить чистий для успішного викладання основних понять.

Той, хто використовував Apple Macintosh або ПК під управлінням Windows, побічно використовував C++, оскільки основні користувальницькі інтерфейси цих систем написані на C++.

Інтегроване середовище розробки Visual Studio (2019)

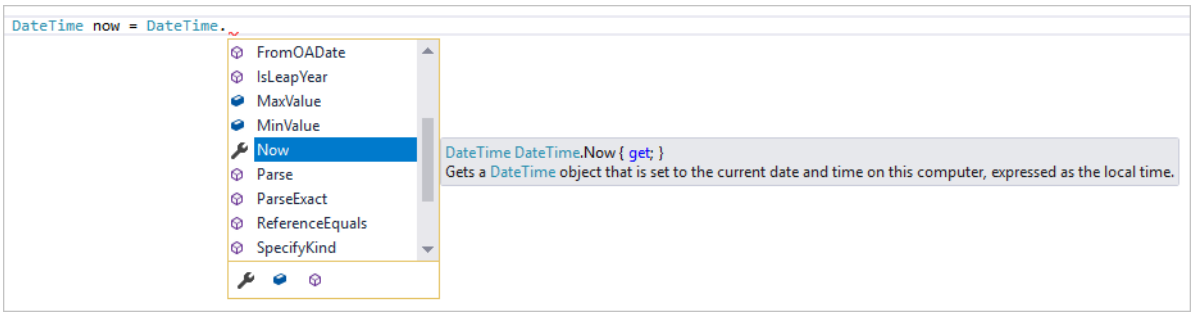
Вбудоване середовище розробки Visual Studio - це креативна стартова площадка, яку ви можете використовувати для редагування, налагодження та побудови коду, а потім опублікувати програму. Інтегроване середовище розробки (IDE) - це багатофункціональна програма, яка може бути використана для багатьох аспектів розробки програмного забезпечення. Крім стандартного редактора та налагоджувача, які надає більшість середовищ розробки, Visual Studio включає компілятори, засоби доробки коду, графічні дизайнери та багато інших функцій для полегшення процесу розробки програмного забезпечення.

Особливості IDE Visual Studio:

- IntelliSense

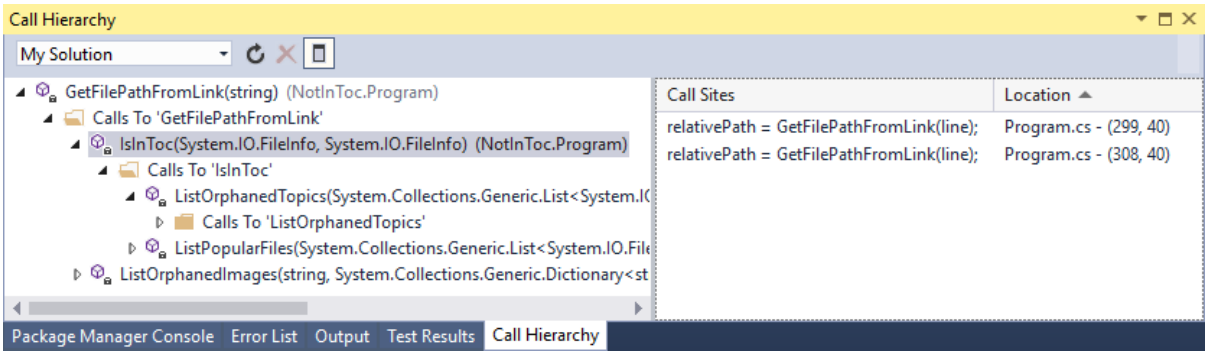
  IntelliSense – це термін для набору функцій, який відображає інформацію про ваш код безпосередньо в редакторі, а в деяких випадках пише для вас невеликі шматочки коду. Це як мати вбудовану основну документацію в редакторі, яка позбавляє вас від необхідності

шукати інформацію про тип в іншому місці. На наступному малюнку показано, як IntelliSense відображає список членів для типу:



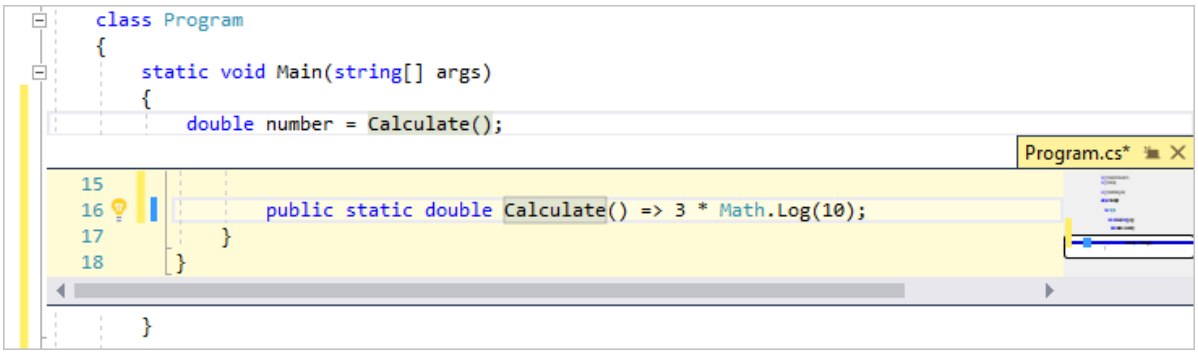
- Call Hierarchy (Ієрархія викликів)

  У вікні ієрархії викликів відображаються методи, які викликають вибраний метод. Це може бути корисною інформацією, коли ви думаєте про зміну або вилучення методу, або коли намагаєтеся виявити помилку.



- Peek definition

  У вікні Peek Definition відображається визначення методу або типу без фактичного відкриття окремого файлу.



3.2 Структура програми

Платформа складається із 17-ти компонентів які пов’язані у 8 груп:

- Основа програми:
  - Ядро програми
  - Система керування часом
- Утиліти:
  - Утиліти загального використання
  - Платформні утиліти
- Система керування подіями
- Система обліку часом
- Діагностика
  - Метрика
  - Логування
- Рушій ECS (Entity Component System)
  - Сутності ECS
  - Компоненти ECS
  - Системи ECS
- Рушій фізики
  - Обробник фізики
  - Визначення колізії
  - Вирішення колізії
- Рушій графіки
  - Обробник графіки
  - Шейдер
  - OpenGL

Компоненти та їхні зв’язки зображені у схемі (рис. 3.1).

Код у проекті відповідає вимогам 2-ох парадигм:

- Об’єктно-орієнтовний дизайн (ООД)

- Інформаційно-орієнтовний дизайн (ІОД)

Всі компоненти які стосуються власне систем проекту написані за правилами ООД, а компоненти які обробляють або містять інформацію написані за правилами ІОД.

ООД дозволяє розробнику швидко та ефективно імплементувати та розширити систему. Єдиний недолік ООД це непослідовне розміщення створених об’єктів у пам’яті, і тому це рішення не є оптимальним для великої кількості об’єктів.

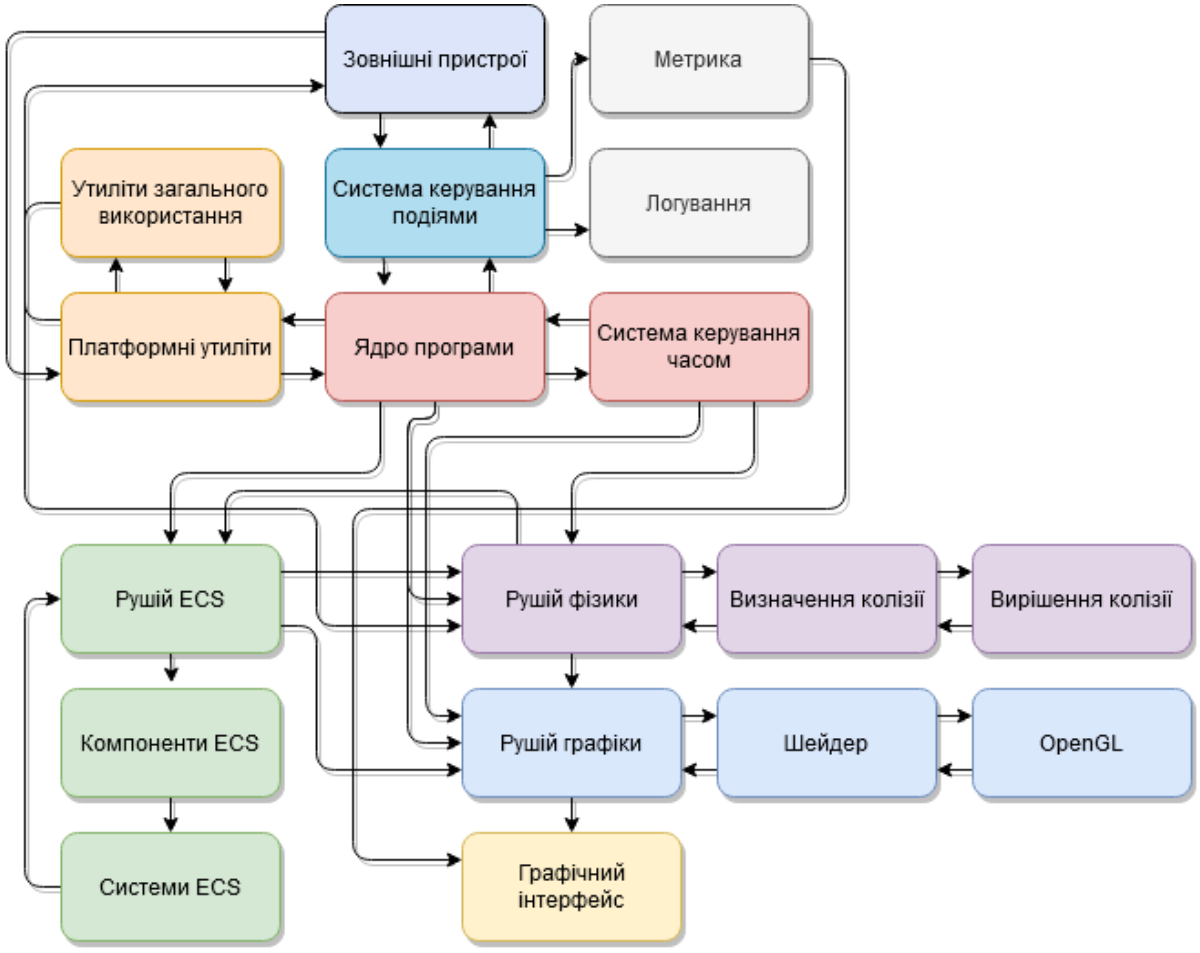


Рис. 3.1. Структура проекту та взаємодія компонентів

Використані бібліотеки

У програмі було використано допоміжні бібліотеки задля скорочення часу розробки та спрощення деяких алгоритмів (табл. 3.1):

Таблиця 3.1

Допоміжні бібліотеки, використані для розробки

Назва бібліотеки	Призначення
GLEW	Бібліотека OpenGL Extension Wrangler (GLEW) – це крос-платформна бібліотека для завантаження розширень C / C++. GLEW забезпечує ефективні механізми виконання для визначення того, які розширення OpenGL підтримуються на цільовій платформі. Функціонал ядра та розширення OpenGL представлений в одному заголовковому файлі. GLEW був протестований на різних операційних системах, включаючи Windows, Linux, Mac OS X, FreeBSD, Irix та Solaris.
GLM	OpenGL Mathematics (GLM) – це хедер (header) математичної бібліотеки C++ для графічного програмного забезпечення на основі специфікацій OpenGL Shading Language (GLSL). GLM надає класи та функції, розроблені та реалізовані з тими самими правилами іменування та функціональністю, що і GLSL, так що кожен, хто знає GLSL, може використовувати GLM також у C++.
Dear ImGui	Dear ImGui – це бібліотека графічного інтерфейсу для C++. Вона виводить оптимізовані буфери вершин, які можна відтворити в будь-який час у програмі з підтримкою 3D-конвеєра. Він швидкий, портативний, агностичний та автономний (без зовнішніх залежностей).
SDL2	Simple DirectMedia Layer – це крос-платформна бібліотека розробки, розроблена для забезпечення

	низько-рівневого доступу до аудіо, клавіатури, миші, джойстика та графічного обладнання через OpenGL та Direct3D. Він використовується програмним забезпеченням для відтворення відео, емуляторами та популярними іграми, включаючи нагороджений каталог Valve та багато ігор Humble Bundle. (У проекті використовується тільки для аудіо, клавіатури та миші).
Assimp	Бібліотека імпорту Open Asset (коротка назва: Assimp) – це портативна бібліотека з відкритим кодом для єдиного імпорту різних відомих форматів 3D-моделей. Остання версія також знає, як експортувати 3D-файли, і тому підходить як перетворювач 3D-моделей загального призначення.
EnTT	EnTT – це бібліотека, яка реалізує швидку ECS (та багато іншого), написана сучасною C++. (У платформі використовуються алгоритми з бібліотеки для крашої швидкодії).

Початок роботи програми

Перед початком програми, користувач має можливість обрати версію програми під свою платформу за допомогою спеціальних прапорців (табл. 3.2).

Таблиця 3.2

Налаштування програми під конкретну платформу користувача

Прапорець	Призначення
OX_PLATFORM_WINDOWS	Підлаштовує роботу програми для платформи Windows

OX_PLATFORM_LINUX	Підлаштовує роботу програми для платформ та дистрибутивів Linux (підтримуються не всі)
OX_PLATFORM_MAC	Підлаштовує роботу програми для платформи MacOS
OX_DEBUG	Підлаштовує роботу програми для діагностики та виправлення багів

Програма починає своє існування із точки входу, а саме з функції int main() (табл. 3.2):

```
int main(int argc, char* argv[])
{
#ifdef OX_DEBUG
	_CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif
	auto app = Onyx::CreateApp();
	int runtime = app->OnExecute();
	delete app;
#ifdef OX_DEBUG
	_CrtDumpMemoryLeaks();
#endif

	return runtime;
}
```

Таблиця 3.3

Опис змінних int main()

Змінна	Призначення
app	Є вказівником до програми, має у собі всі параметри та функції для її контролю.
runtime	Код, який повертає програма після її завершення.

Тут також використовуємо прапорець OX_DEBUG для задання аналітики визначення витоків пам’яті.

Ядро програми

Ядро відповідає за ініціалізацію та основний цикл програми.

Опис методів класу App (табл 3.4), алгоритми у додатку А:

Таблиця 3.4

Опис методів класу App

Метод	Призначення
Get()	Повертає екземпляр програми в якості вказівника
OnExecute()	Виконує основний цикл програми
GetWindowWidth()	Повертає ширину вікна програми
GetWindowHeight()	Повертає висоту вікна програми
GetWindow()	Повертає екземпляр вікна
GetGLContext()	Повертає екземпляр OpenGL
GetMetrics()	Повертає діагностичні дані
OnInit()	Ініціалізує модулі
OnEvent()	Оброблює події програми
OnCleanup()	Очищує пам’ять коли програма завершує роботу
ResizeWindow()	Змінює розміри вікна програми

Програмувальні модулі

Кожен 3Д-додаток складається з так званих “модулів”, Ці модулі можуть бути чим завгодно, від логіки додатку до реалізації штучного інтелекту. Система була створена таким чином, щоб розробник не мав лімітів відносно реалізації його ідей.

Схема структури модулів та їх взаємодії із системою (рис. 3.2):

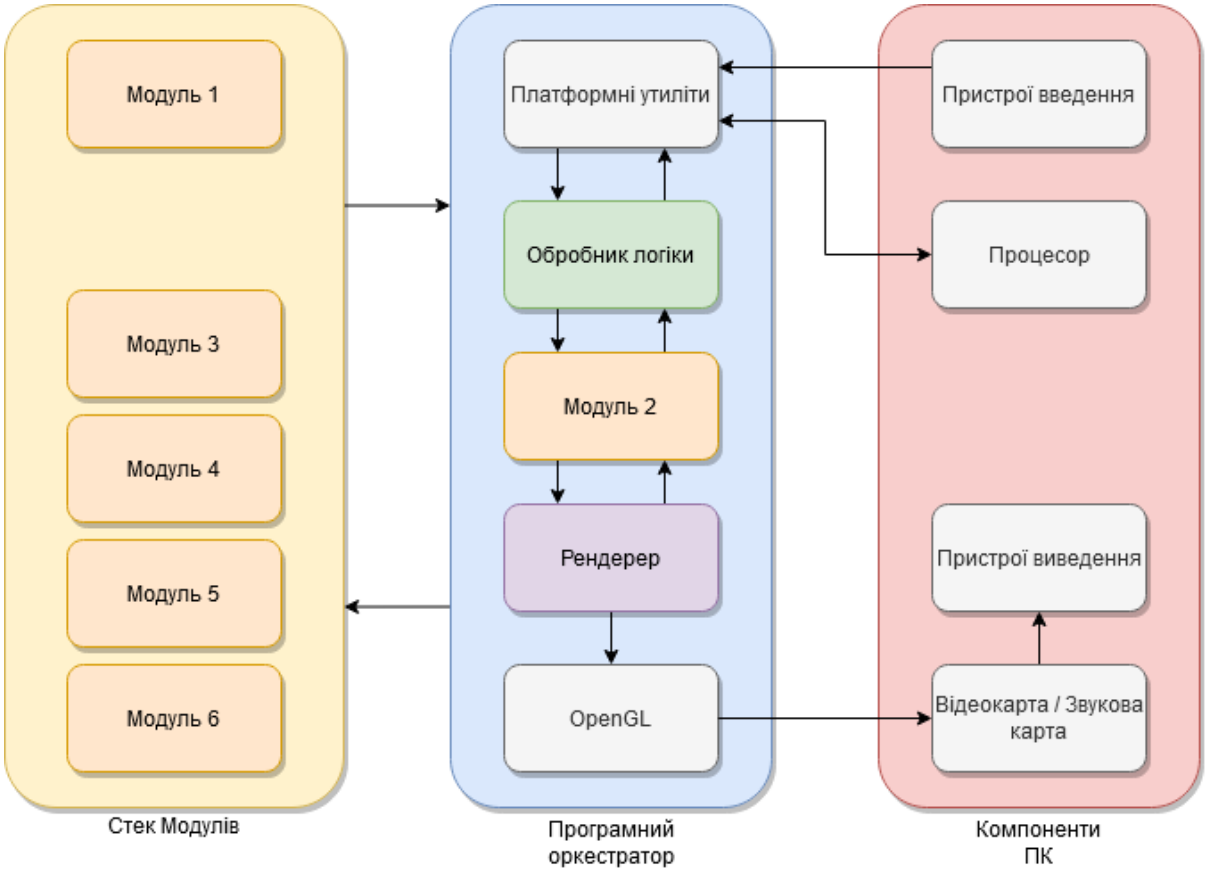


Рис. 3.2. Схема структури модулів та їх взаємодії із системою

Таким чином, кожен модуль послідовно оброблюється програмним оркестратором. Послідовність виконання є недоліком, адже всі модулі виконуються на одному потоці очікуючи своєї черги. Кращим варіантом є інтеграція багатопоточності, що дозволить модулям системи виконуватися не на одному, а на багатьох ядрах, паралельно, покращуючи продуктивність.

Фрагмент коду програмного оркестратора:

```
static float accumulator = 0.f;
while (Running)
{
        Time::UpdateTime();

        while (SDL_PollEvent(&event))
                OnEvent(&event);

        for (Layer* layer : m_LayerStack)
        {
                layer->StartFrame();
                layer->DrawGui();

                if (Time::RawFixedTime() == 0)
```

```
                continue;

            int fixedIterations{ 0 };

            accumulator += Time::DeltaTime();

            while (accumulator >= Time::RawFixedTime() && fixedIterations < 10)
            {
                layer->OnFixedUpdate(Time::FixedTime());
                accumulator -= Time::RawFixedTime();
                fixedIterations++;
            }

            Running = layer->OnUpdate(Time::DeltaTime());
            if (!Running)
                break;

            metrics->GetStats().timeStats.frameTime = Time::DeltaTime();
            metrics->GetStats().timeStats.frameRate = 1000.f / Time::DeltaTime();

            layer->OnRender(Time::DeltaTime());

            layer->EndFrame();
        }
        SDL_GL_SwapWindow(Window);
}
```

На кожен модуль викликаємо функцію StartFrame() (Початок кадру), після того малюємо інтерфейс і оновлюємо логіку модуля, потім рендерим об’єкти у сцені і закінчуємо функцією EndFrame() (Кінець кадру).

Кожен програмний модуль має стандартний набір функцій які мають бути оброблені розробником:

- OnAttach() – “При приєднанні” – ця функція оброблюється на початку існування програмного модулю, тому виконується один раз. У цій функції зазвичай ініціалізуються всі причетні до модуля компоненти.
- OnInit() – “При ініціалізації” – ця функція оброблюється коли всі програмні модулі виконали функцію OnAttach(), тому виконується один раз Використання цієї функції є винятковим, тому вона є необов’язковою і розробник може вирішити її не використовувати
- OnDetach() – “При від’єднанні” – ця функція оброблюється в кінці існування програмного модуля, тому виконується один раз. У цій функції зазвичай “звільняється” вся пам’ять яку використовував модуль: вказівники, об’єкти, тощо.

- OnUpdate() – “При оновленні” – ця функція оброблюється на кожному кадрі, тому виконується поки програмний модуль не завершить свого існування. У цій функції знаходиться вся логіка яка має оновлюватися постійно (штучний інтелект, анімація, інші внутрішні системи і тд.).
- OnRender() – “При рендері” – ця функція оброблюється на кожному кадрі, тому виконується поки програмний модуль не завершить свого існування. У цій функції знаходиться вся логіка, пов’язана з показом чогось на екрані.
- DrawGui() – “При рендерингу інтерфейсу” – ця функція оброблюється на кожному кадрі, тому виконується поки програмний модуль не завершить свого існування. У цій функції функції знаходиться все, що причетне до інтерфейсу користувача.
- OnEvent() – “При події” – ця функція оброблюється на кожному кадрі, тому виконується поки програмний модуль не завершить свого існування. У цій функції виконується обробка всіх подій у модулі, це може бути, для прикладу, натиск клавіші на клавіатурі.
- OnResize() – “При зміні розміру вікна” – ця функція оброблюється на кожному кадрі, тому виконується поки програмний модуль не завершить свого існування. Ця функція оброблює зміну розміру вікна програми.
- StartFrame() – “Початок кадру” – ця функція виконується на початку кожного кадру.
- EndFrame() – “Кінець кадру” – ця функція виконується в кінці кожного кадру.

Фрагмент коду програмного модуля EditorModule (модуль редактора):

```
class EditorModule : public Layer
{
private:
        float clientDeltaTime;
        std::shared_ptr<Scene> activeScene;
        std::shared_ptr<ECSManager> ecsManager;
        std::shared_ptr<ECS::PhysicsProcessor> physicsProcessor;
        Framebuffer framebuffer;

        std::shared_ptr<sol::state> lua;

        Entity editorCamera;
        OnyxSerializer serializer;

        bool mouseMove = true;

        std::shared_ptr<ImGuiModule> imGuiModule;
```

```
        float counter = 0;

public:
        EditorModule();
        ~EditorModule() = default;

        static bool Running;

        void OnInit() override;
        void OnAttach() override;
        void OnDetach() override;
        bool OnUpdate(const float& deltaTime) override;
        void OnRender(const float& deltaTime) override;
        void OnResize() override;
        void OnEvent(SDL_Event* event) override;
        void OnFixedUpdate(const float& deltaTime) override;

        void StartFrame() override;
        void EndFrame() override;

        void DrawGui() override;

        void KeyboardHandler(const float& deltaTime);

        std::shared_ptr<ImGuiModule> GetImGui() const { return imGuiModule; }
};
```

Утиліти загального використання

Ці утиліти призначені для використання усією програмою у будь якому місті та у будь-який час.

До утиліт загального використання входять:

- Математичні утиліти
- Допоміжні функції для платформи

До математичних утиліт входять такі функції:

- Функція max() яка знаходить максимальне значення із двох змінних a і b:
- Функція min() яка знаходить мінімальне значення із двох змінних a і b:
- Функція lerp(), яка інтерполює значення a і b за допомогою значення альфи w:
- Функція clamp(), яка приймає 3 значення. X, мінімум і максимум. Якщо X знаходиться в діапазоні між мінімальним і максимальним значенням,

він повертає X. Якщо X нижчий за мінімальне значення, він повертає мінімальне, а якщо X вищий за максимальне значення, повертає максимальне.

- Функція DecomposeTransform() перетворює матрицю 4x4 на 3 вектори (У програмі перетворює матрицю об’єкта в просторі на вектори: позицію, ротацію, розмір):

Платформні утиліти

Ці утиліти призначені для використання визначеним платформам (операційним системам). Реалізовані утиліти для ОС Windows (табл. 3.5):

Таблиця 3.5

Реалізовані утиліти для ОС Windows

Функція	Призначення
CheckFileExists()	Повертає true, якщо існує вказаний файл
GetCurrentDir()	Повертає рядок поточної директорії
CC_ToString()	Конвертує дані типу const char* у std::string
GetEngineDir()	Повертає рядок головної директорії платформи
GetEngineShaderDir()	Повертає рядок директорії з файлами шейдерів
GetEngineModelsDir()	Повертає рядок із файлами 3D моделей

Система обліку часом

Розраховує все що зв’язане з часом у платформі: дельта-час, загальний час, фіксований час, точний час на даний момент (табл. 3.6):

Таблиця 3.6

Функції системи курування часом

Функція	Призначення
UpdateTime()	Виконується через кожен програмний цикл, оновлює лічильники часу
Ticks()	Повертає акумулятор кількості оновлень від початку

	роботи програми
TotalTime()	Повертає час який пройшов від початку роботи програми
DeltaTime()	Повертає дельта-час
FixedTime()	Повертає фіксований час

Метрика

Метрика записує діагностичні дані для статистики та виправлення помилок (табл. 3.7).

Приклад використання:

Metrics metrics = metrics->getInstance();*

metrics->GetStats().timeStats.frameTime = Time::DeltaTime();

Клас метрики розроблений за шаблоном Сінглтон (Singleton), тобто під час роботи програми існує тільки один екземпляр класу і діагностичні дані можна вилучити де завгодно і у будь-який момент життєвого циклу платформи (окрім закінчення).

Таблиця 3.7

Функції метрики

Функція	Призначення
getInstance()	Повертає екземпляр класу метрики
GetStats()	Повертає діагностичні дані

Логування

Логування призначене для друкування повідомлень діагностичного характеру на консоль (табл. 3.8).

Ці функції також є у вигляді препроцесорних макросів і їх можна використовувати де завгодно і під час всього життєвого циклу платформи (навіть закінчення).

Таблиця 3.8

Функції класу логування

Функція	Призначення
LogTrace()	Логує (друкує на консоль) трасувальне повідомлення (значення змінної для прикладу). (Синій колір)
LogInfo()	Логує загальну інформацію (типу: «Модуль завантажився успішно»). (Білий колір)
LogError()	Логує помилку (Червоний колір)
LogWarn()	Логує зауваження але це не вважається помилкою (Жовтий колір)
LogFatal()	Логує фатальну помилку що призводить до закриття програми (Фіолетовий колір)

Сутність (ECS)

Є можливість представити сутність ECS в якості класу для доступу до допоміжних функцій. У ECS зберігається тільки ІД сутності, тому представлення в якості класу Сутності не впливає на продуктивність (клас не є вказівником, тому знищується як тільки вийде виконання за межі довільної функції) (табл. 3.9):

Використання:

```
Entity entity1 = ecsManager->CreateEntity("EntityName1");
entity1.AddComponent<Component1>();
auto& comp2 = entity1.GetComponent<Component2>();
```

Таблиця 3.9

Функції класу сутності

Функція	Призначення
AddComponent()	Добавляє компонент до ECS для заданої сутності
GetComponent()	Повертає компонент сутності вилучивши його з ECS
HasComponent()	Повертає true, якщо компонент існує для заданої сутності

RemoveComponent()	Видаляє компонент з ECS для заданої сутності
GetEntityId()	Повертає ідентифікатор сутності

Компонент (ECS)

Компонент є контейнером з інформацією для сутності:

Для прикладу фрагмент коду:

```
struct ColorComponent
{
	glm::vec3 color{ 0.f, 0.f, 0.f };
};
```

Система (ECS)

Система (інша назва: «Процесор») є набором функцій (логіки) для роботи з компонентами:

Для прикладу фрагмент коду:

```
void ECSProcessor::OnReloadLuaScriptComponents(ECSManager& ecsManager)
{
	ecsManager.registry.view<LuaScriptComponent>().each([&ecsManager](auto entity,
LuaScriptComponent& lsc)
	{
		if (!lsc.instance && !lsc.scriptFile.empty() && !lsc.scriptClass.empty() &&
lsc.InstantiateScript)
		{
			lsc.instance = lsc.InstantiateScript(lsc.lua, lsc.scriptFile, lsc.scriptClass);
			lsc.instance->scriptableObject = Entity{ entity,
std::addressof(ecsManager) };
			lsc.instance->OnCreate();
		}
	});
}
```

Фізика (Динаміка)

Динаміка відповідає за генерування постійної гравітаційної сили, за розрахунок інших потенційних сил, а також для реагування на потенційні колізії з іншими фізичними тілами (табл. 3.10).

Фізичне тіло у платформі представлено в якості компонента (табл. 3.11):

Таблиця 3.10

Функції класу динаміки

Функція	Призначення
AddRigidbody()	Додає фізичне тіло у колекцію
AddConstraint()	Додає фізичне обмеження у колекцію
Step()	Крок симуляції
Gravity()	Повертає значення глобальної гравітації
SetGravity()	Задає значення глобальної гравітації
Constraints()	Повертає колекцію з фізичними обмеженнями
BeforeStep()	Виконується перед кроком симуляції
AfterStep()	Виконується після кроком симуляції
TrySetGravity()	Задає глобальну гравітацію
TryApplyGravity()	Відтворює глобальну гравітацію
PredictTransforms()	Передбачає матриці трансформації

Таблиця 3.11

Змінні структури фізичного тіла

Змінна	Призначення
type	Тип фізичного об’єкта (Фізичне тіло або Об’єкт для колізії)
gravity	Локальна гравітаційна сила
netForce	Загальна сила яка діє на фізичне тіло
Velocity	Швидкість фізичного тіла
netTorque	Ротаційна сила
angularVelocity	Ротаційна швидкість
Inertia	Інерція
axisLock	Замикання по даній осі
isAxisLocked	Чи замкнений на даній осі
Mass	Маса
invMass	Інвертована маса (1 / маса)
takesGravity	Чи впливає локальна гравітація на фізичне тіло

simulateGravity	Чи симулюється гравітація для фізичного тіла
isKinematic	Чи фізичне тіло на колізію, але без відповіді на неї (без імпульсу)
staticFriction	Коефіцієнт статичного тертя (зупиняє фізичне тіло)
dynamicFriction	Коефіцієнт динамічного тертя (прискорює фізичне тіло)
Restitution	Коефіцієнт відскоку
isGrounded	Чи є фізичне тіло приземленим
lastCollisionNor-mal	Нормаль останньої колізії
lastTransform	Остання матриця трансформації
nextTransform	Наступна матриця трансформації
isDynamic	Чи є фізичне тіло динамічним

Фізика (Колізії)

У цьому компоненті відбувається тестування та визначення колізій і розрахунок імпульсів та корекція позицій фізичних тіл (табл. 3.12).

Таблиця 3.12

Функції класу колізії

Функція	Призначення
AddCollisionObject()	Додає сутність до колекції об’єктів колізії
RemoveCollisionObject()	Видаляє сутність з колекції об’єктів колізії
AddSolver()	Додає алгоритм вирішення колізії
RemoveSolver()	Видаляє алгоритм вирішення колізії
ResolveConstraints()	Розраховує фізичні обмеження
TestCollider()	Тестує коллайдер
TestObject()	Тестує об’єкт

Графіка (Обробник графіки)

Обробник графіки отримує дані для шейдерів від ECS і напрявляє ці дані на обробку на графічному конвеєрі (табл. 3.13):

Таблиця 3.13

Функції класу рендерера

Функція	Призначення
StartRender()	Оброблює початок кадру
EndRender()	Оброблює кінець кадру
RenderWireframe()	Обробить об’єкт в якості 3D сітки
RenderNormal()	Обробить об’єкт для показу нормалей
RenderUnlit()	Обробить об’єкт без освітлення
RenderLit()	Обробить об’єкт з освітленням та текстурами
RenderLitLow()	Обробить об’єкт з освітленням без текстур
RenderGrid()	Обробить сітку редактора

Графіка (Шейдер)

Шейдер містить в собі код GLSL, графічні властивості та параметри шейжера та функції роботи з шейдерами (компіляція шейдера, його валідація та верифікація) (табл. 3.14):

Таблиця 3.14

Функції класу шейдера

Функція	Призначення
LoadShaders()	Загружає шейдери у відеокарту
UseShader()	Використовує шейдер
Cleanup()	Очищує дані шейдера
GetProperties()	Повертає властивості та параметри шейдера

3.3 Функціональні можливості платформи

Платформа – це набір інструментів, які зв’язують функції системи та відеокарти для створення 3D-додатків. Створені програми мають можливість скористатися функціями графічного рушія для зображення 3D сцен, також фізичного рушія для симуляції фізики між фізичними тілами. Оскільки платформа безпосередньо працює з графікою та симуляціями реального часу, є певні мінімальні вимоги перед запуском (табл. 3.15):

Таблиця 3.15

Технічні вимоги до апаратного забезпечення

Вимога	Характеристика
Мінімум набір інструкцій відеокарти	SSE2
Мінімум пам’ять	4 Gb

Також платформа на момент написання звіту адаптована під ОС Windows, тому цій ОС доступні всі функції та можливості.

Редактор є стандартною програмою для платформи і призначений для створення та маніпулювання 3D сцен, створення 3D об’єктів та зміни їхніх параметрів, прив’язання скриптів до об’єктів та їх виконання, симуляції фізики і т.д.

Після запуску платформи з’являються два вікна: вікно консолі та вікно редактора. Вікно консолі (рис. 3.1) друкує повідомлення з метою проінформувати користувача про події у платформі. Вікно редактора (рис. 3.3) є основною програмою з якою користувач буде взаємодіяти.

Кожне повідомлення на консолі має в собі час коли ця подія відбулася: [Onyx 10:27:3 TRC], тому користувач має можливість профілювання платформу на швидкість, звіряючи час між операціями. Платформа також

видає помилки OpenGL для полегшеного діагностування графічних помилок (рис. 3.1).

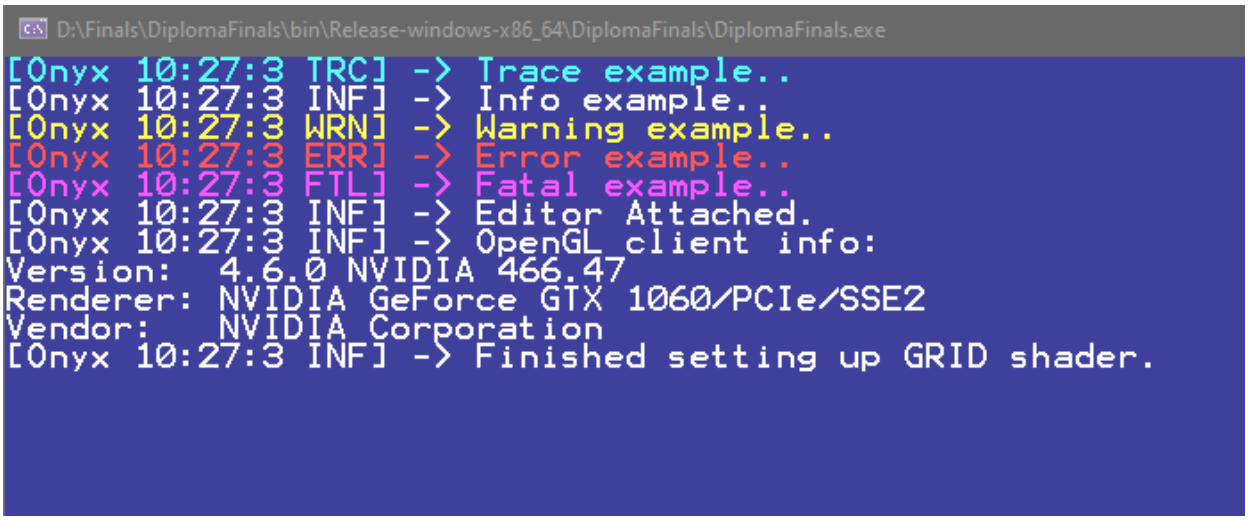


Рис. 3.1. Вікно консолі

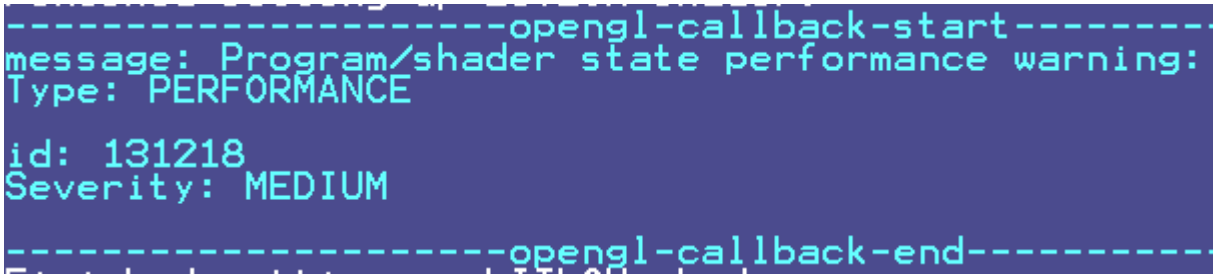


Рис. 3.2. Приклад повідомлення від OpenGL

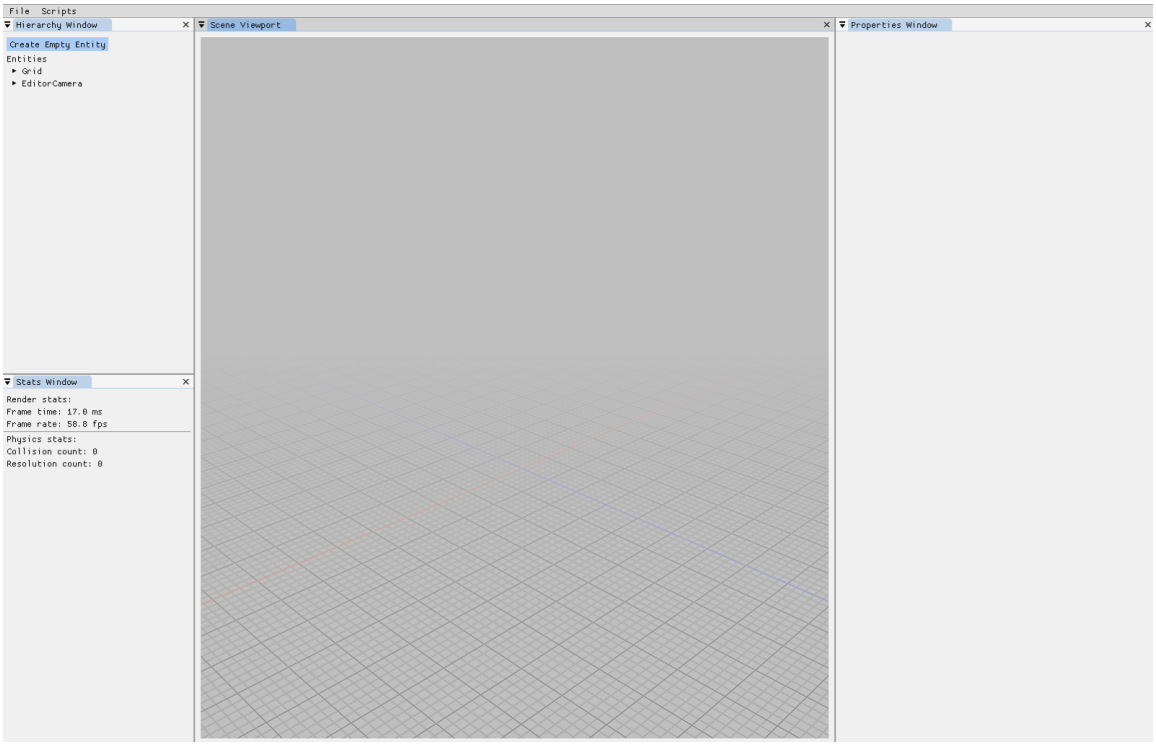


Рис. 3.3. Вікно редактора після запуску платформи

Інтерфейс редактора має в собі п’ять елементів:

- Головне меню
- Панель ієрархії
- Панель параметрів
- Панель діагностики
- Панель відображення сцени

Інтерфейс редактора є гнучким – користувач має можливість згорнути панель, змінити її розташування (рис. 3.4), та поєднати з іншими панелями.

Головне меню

Головне меню має два пункти:

- File
- Scripts

Пункт File (рис. 3.5) призначений для створення нової сцену, завантаження існуючої сцени та для запису поточної сцени.

Пункт Scripts (рис. 3.6) призначений для перезавантаження скритів, додавання скриптів та іх видалення.

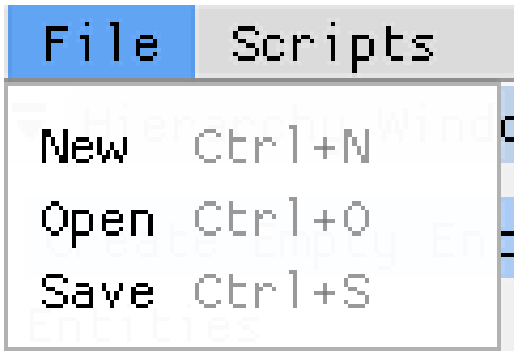


Рис. 3.5. Пункт File

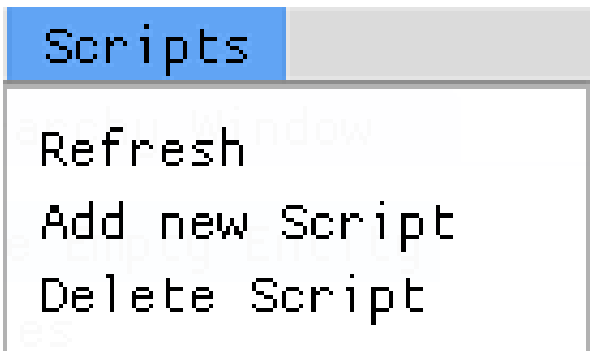


Рис. 3.6. Пункт Scripts

Панель ієрархії

Панель ієрархії містить в собі список сутностей які присутні на сцені, є можливість вибрати сутність зі списку, видалити сутність та створити сутність без компонентів (рис. 3.7).

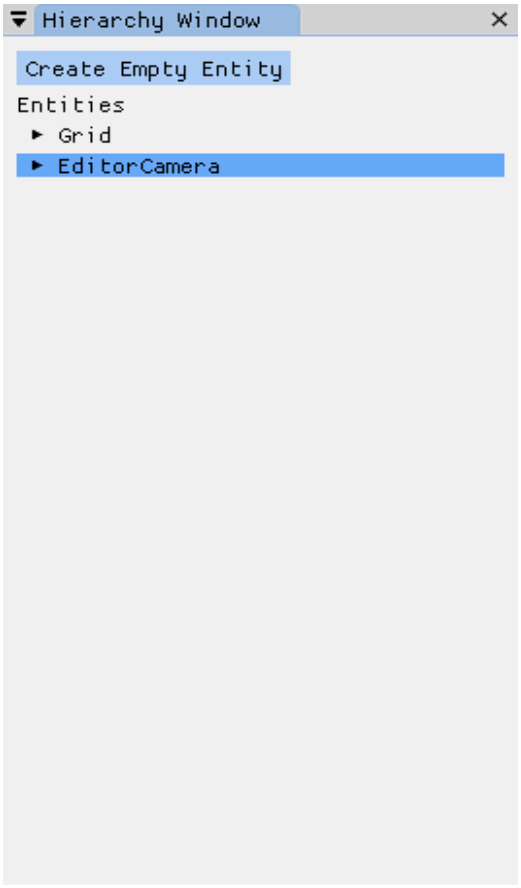


Рис. 3.7. Панель ієрархії

Панель параметрів

Панель параметрів призначена для керування компонентами сутностей. Є можливість зміни назви сутності, додати новий компонент та видалити існуючий (рис. 3.8). Кожен компонент має елементи керування, які потрібні компоненту для маніпуляції його параметрів, до елементів курування входять:

- Поле для введення тексту
- Кнопка
- Випадаючий список
- Елемент керування векторами із трьома змінними
- Елемент вибору кольору
- Прапорці
- Радіо-кнопки

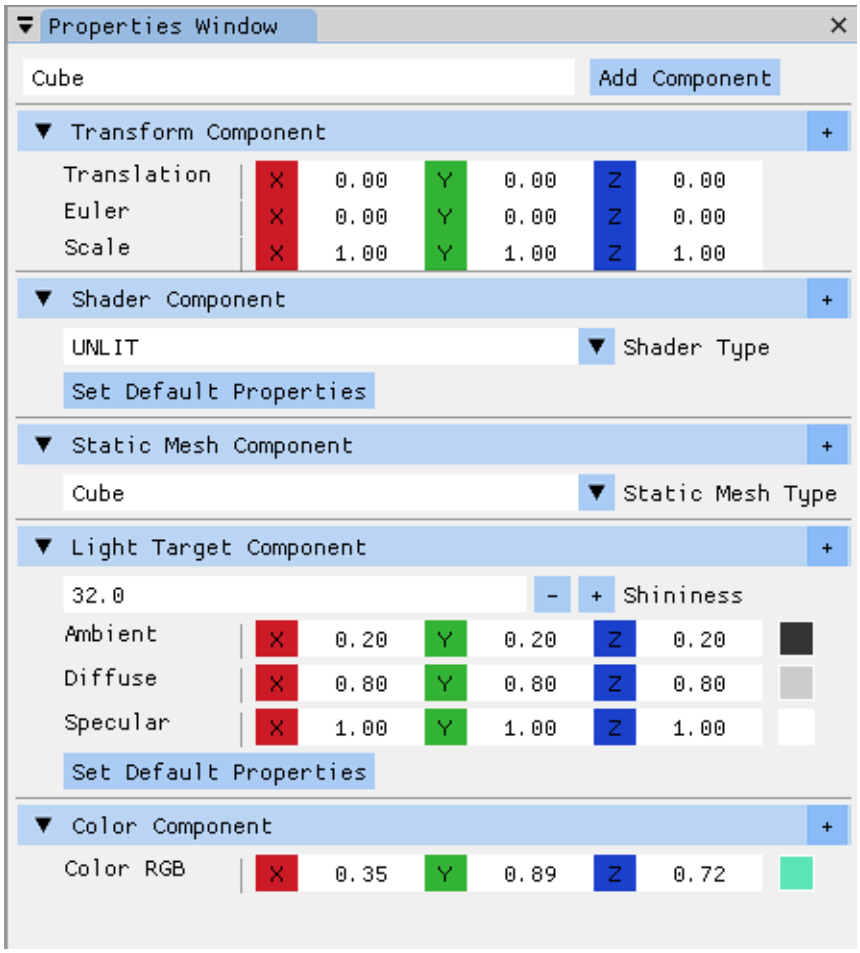


Рис. 3.8. Приклад довільної панелі параметрів сутності

Панель діагностики

Панель діагностики містить дані стану платформи. Наразі показує час між кадрами (у ms), кількість кадрів у секунду (fps), та загальну кількість колізій між фізичними тілами (рис. 3.9).

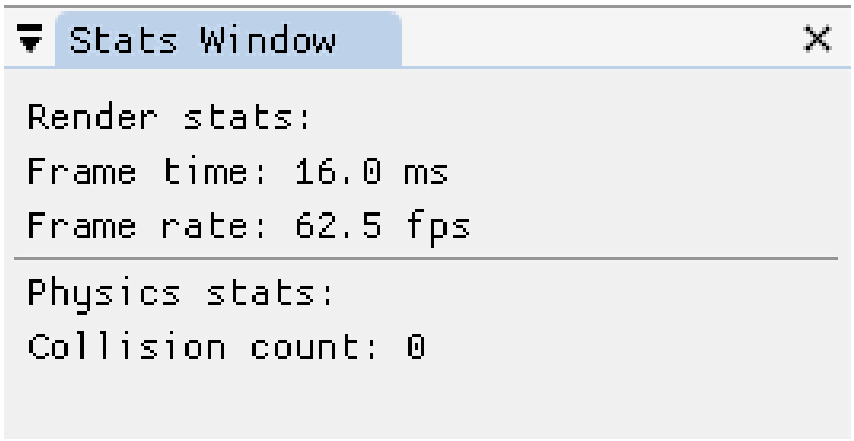


Рис. 3.9. Панель діагностики

Панель відображення сцени

Панель відображення сцени зображує 3D віртуальне середовище з 3D об’єктами у реальному часі. 3D сцена має у собі допоміжну сітку для орієнтації, вона розміщена на координаті Y:0 і генерується нескінченно по осям X та Z. Ця сітка також має синю (Z ось) та червону (X ось) лінії, їх перетин є абсолютним центром віртуального світу (рис. 3.10).

За допомогою шейдерів, є можливість зміни вигляду 3D об’єктів, наразі існує 4 різні шейдери які можна застосувати до сутностей:

- UNLIT (без освітлення) (рис. 3.11)
- LITLOW (з освітленням але без текстур) (рис. 3.12)
- LIT (з освітленням та текстурами) (рис. 3.14)
- NORMAL (зображення нормалей) (рис. 3.13)

У редакторі також є можливість завантаження моделей .obj формату та текстур до них (рис. 3.14).

Інструмент Гізмо (Gizmo) призначений для маніпуляції позиції (рис. 3.15), ротації (рис. 3.16) та розміру (рис. 3.17) 3D об’єкта га сцені.

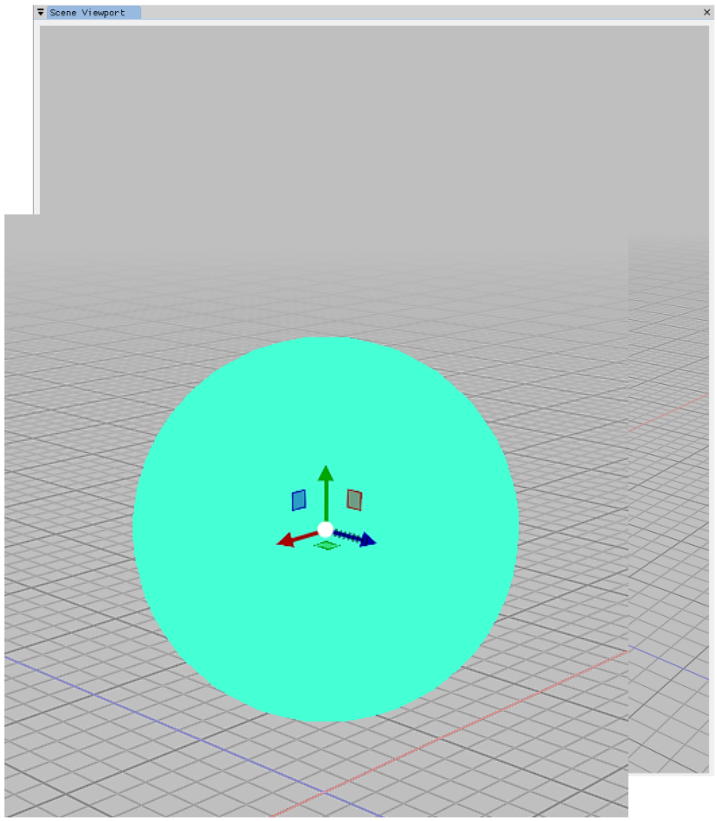


Рис. 3.10. 3D сцена з допоміжною сіткою та без сутностей

Рис. 3.11. Куля без освітлення

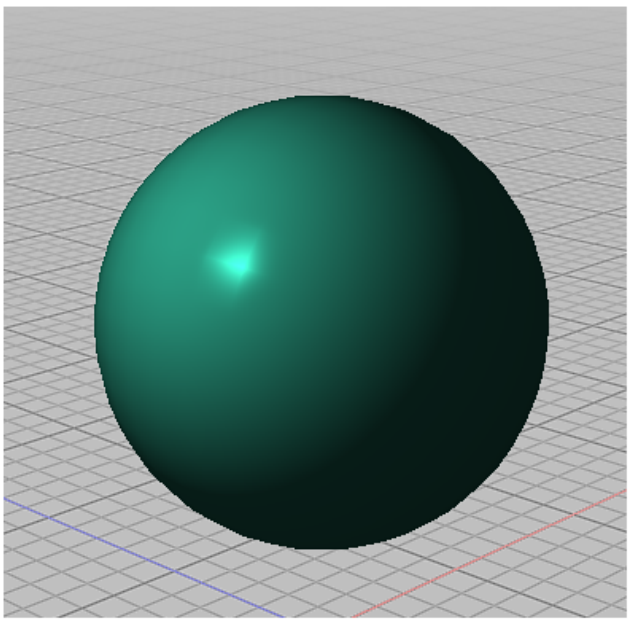


Рис. 3.12.

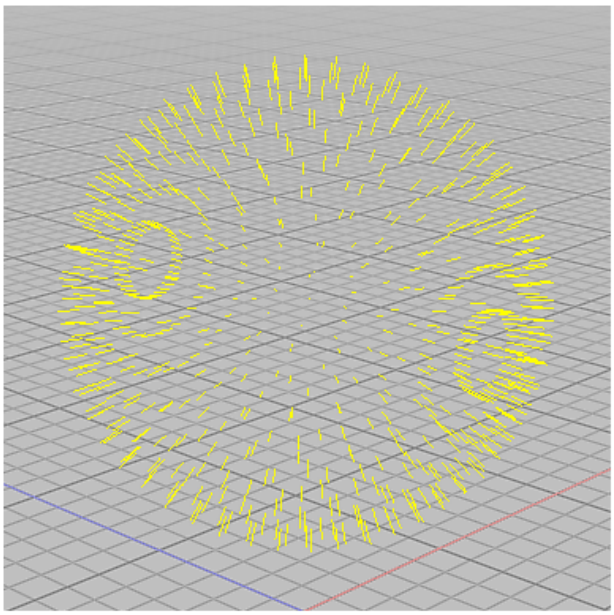
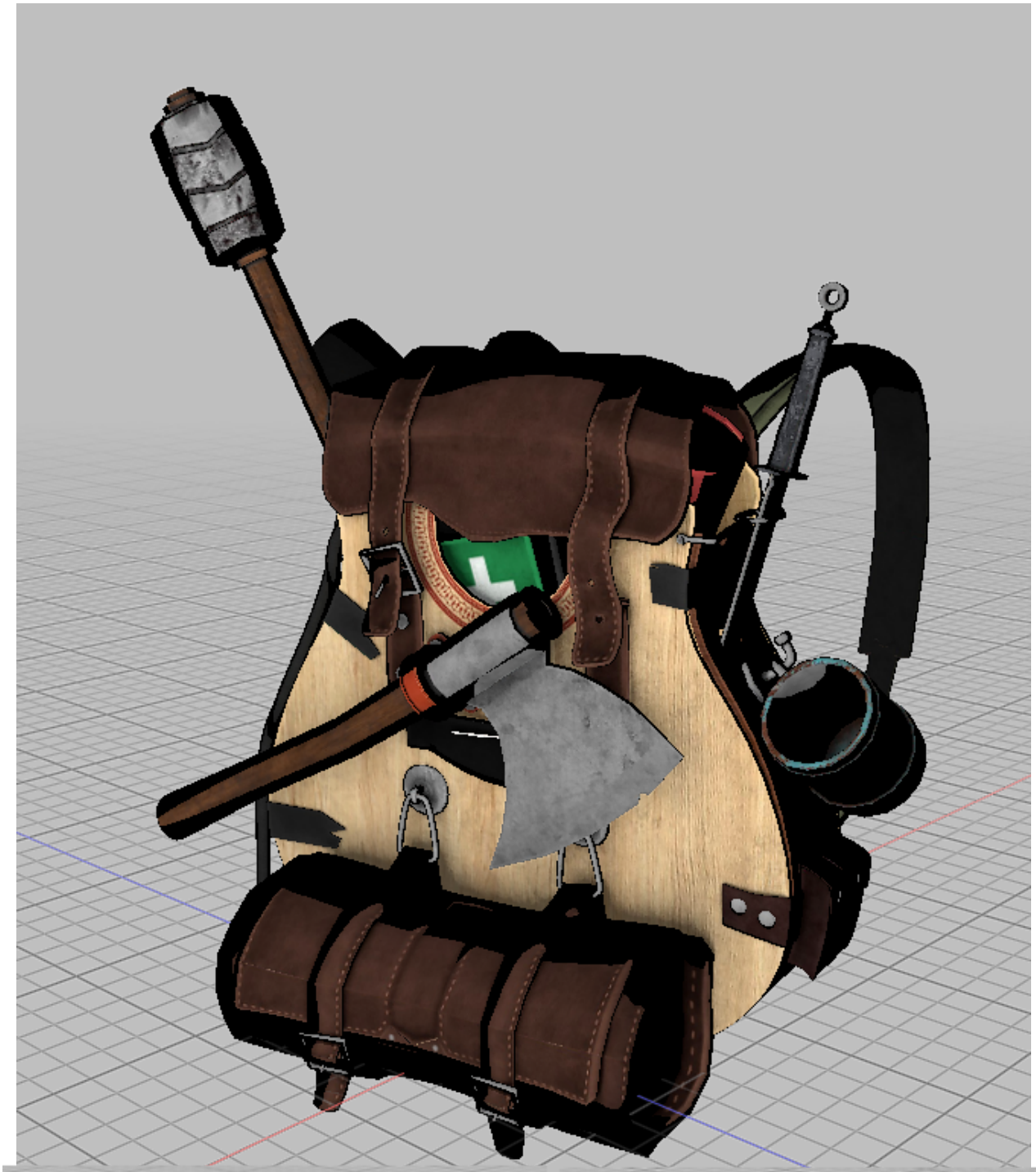


Рис. 3.14. Відобра-



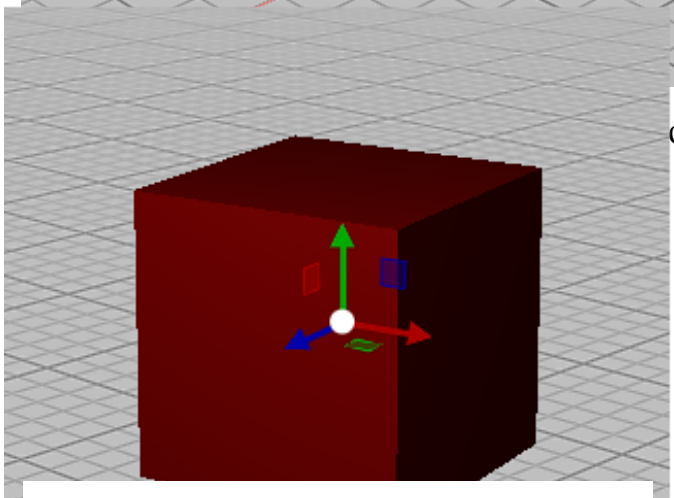


Рис. 3.15. В центрі куба інструмент Гізмо, режим роботи – трансляція

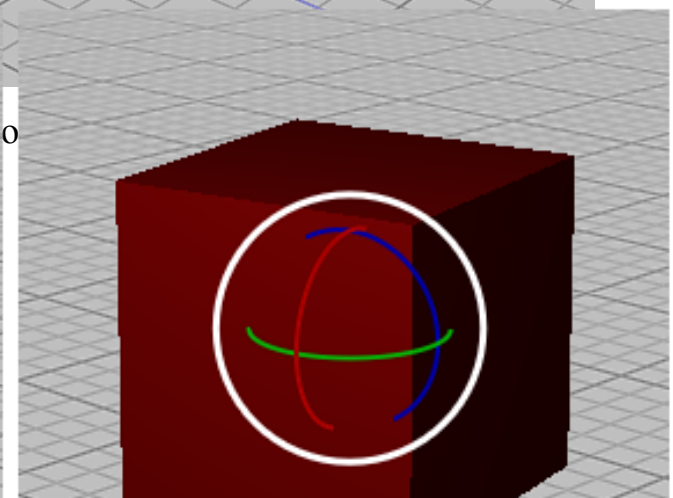


Рис. 3.16. Гізмо, режим роботи – ротація

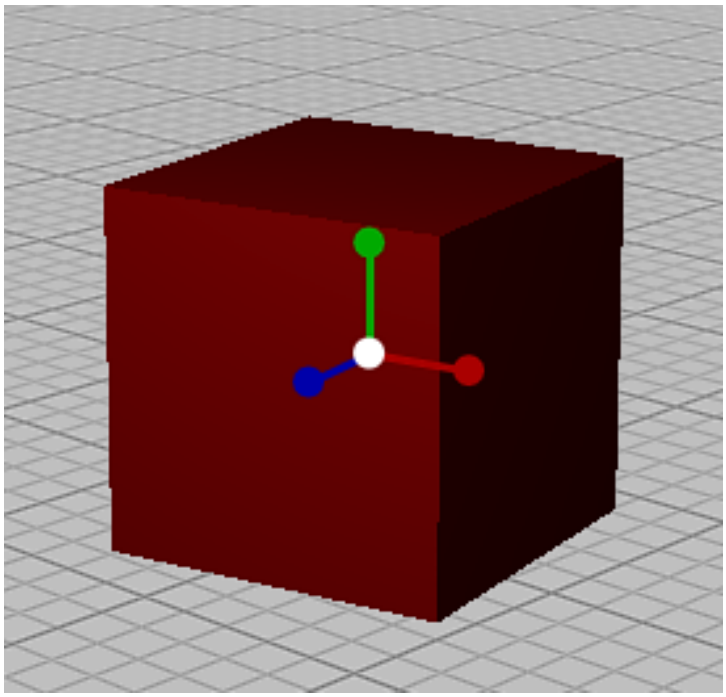


Рис. 3.17. Гізмо, режим роботи – зміна розміру

Фізичний рушій підтримує понад 10,000 фізичних тіл, 3D сцена підтримує до 200 з нормативною швидкодією і після 300 буде значний спад у швидкодії (рис. 3.18).

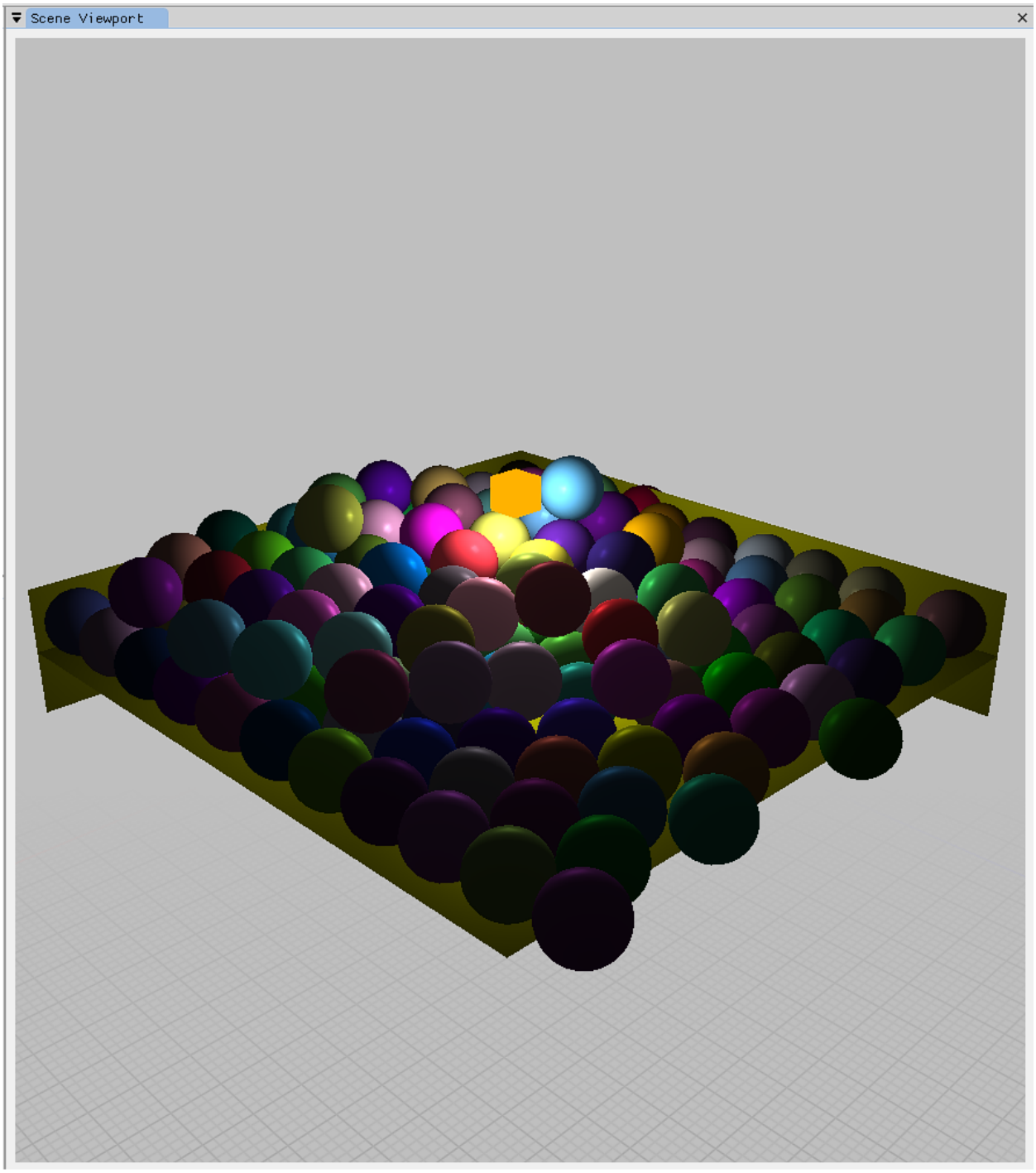


Рис. 3.18. Сцена симулює 150 фізичних тіл

ВИСНОВКИ

На першому етапі було досліджено предметну галузь, а саме суть графічного рушія, його застосування, основні функції та задачі. Також було проведено порівняльну характеристику найпопулярніших графічних рушіїв і за їхніми функціями та особливостями будо впроваджено вимоги та завдання до програмної реалізації бакалаврської роботи.

На другому етапі було представлено технології які були досліджені для програмної реалізації, також за цими технологіями були відповідно створені зв’язки та код для інтеграції у програмну реалізацію задля задоволення поставлених потреб та вимог.

На третьому етапі було проведено характеристику програмного функціоналу, функції та методи за якими будо спроектовано програмну реалізацію. Також було показано приклад використання функціоналу.

У результаті виконання трьох етапів було спроектовано та розроблено графічний рушій загального призначення та його складових до яких входять: система обробки подій, система ECS, обробник графіки, обробник фізики.

Платформа може бути використана для реалізацій інтерактивних програмних застосунків із графічними вимогами в режимі реального часу, до яких входять:

- Маркетингові демонстрації
- Архітектурні візуалізації
- Навчальні симуляції
- Моделювання середовища

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. https://en.cppreference.com/w/
2. https://www.tutorialspoint.com/cplusplus/index.htm
3. https://www.learncpp.com/
4. https://learnopengl.com
5. https://www.khronos.org/opengl/wiki/
6. https://neil3d.github.io/reading/assets/slides/Introduction_to_Data-Oriented_Design_2014DICE.pdf
7. https://gamesfromwithin.com/data-oriented-design
8. https://www.dataorienteddesign.com/dodbook/
9. https://hellocplusplus.com/data-oriented-design/
10. https://medium.com/@gamevanilla/data-oriented-design-is-more-than-justperformance-d3aad3bf3b5a
11. http://entity-systems.wikidot.com/
12. https://www.haroldserrano.com/blog/how-a-physics-engine-works-an-overview
13. https://nphysics.org/continuous_collision_detection/
14. https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection
15. https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/previousinformation/physics6collisionresponse/2017%20Tutorial%206%20-%20Collision%20Response.pdf
16. https://dyn4j.org/2010/04/gjk-gilbert-johnson-keerthi/

ДОДАТОК

```
App* App::s_Instance = nullptr;

App::App()
{
	Running = true;

	s_Instance = this;

	SCREEN_WIDTH = 1600u;
	SCREEN_HEIGHT = 1024u;

	Time::UpdateTime();

	ONYX_CORE_TRACE("Trace example..");
	ONYX_CORE_INFO("Info example..");
	ONYX_CORE_WARN("Warning example..");
	ONYX_CORE_ERROR("Error example..");
	ONYX_CORE_FATAL("Fatal example..");
}

int App::OnExecute()
{
	if (!OnInit(SCREEN_WIDTH, SCREEN_HEIGHT))
		return -1;

	std::string glVersion = reinterpret_cast<const char*>(glGetString(GL_VERSION));
	std::string glRenderer = reinterpret_cast<const char*>(glGetString(GL_RENDERER));
	std::string glVendor = reinterpret_cast<const char*>(glGetString(GL_VENDOR));
	ONYX_CORE_INFO("OpenGL client info: \nVersion:  " + glVersion +
		"\nRenderer: " + glRenderer +
		"\nVendor:   " + glVendor);

	for (Layer* layer : m_LayerStack)
		layer->OnInit();

	SDL_Event event{};
	Metrics* metrics = metrics->getInstance();
	Time::UpdateTime();

	static float accumulator = 0.f;
	while (Running)
	{
		Time::UpdateTime();

		while (SDL_PollEvent(&event))
			OnEvent(&event);

		for (Layer* layer : m_LayerStack)
		{
			layer->StartFrame();
			layer->DrawGui();

			if (Time::RawFixedTime() == 0)
				continue;

			int fixedIterations{ 0 };
```

```
                        accumulator += Time::DeltaTime();

                        while (accumulator >= Time::RawFixedTime() && fixedIterations < 10)
                        {
                                layer->OnFixedUpdate(Time::FixedTime());
                                accumulator -= Time::RawFixedTime();
                                fixedIterations++;
                        }

                        Running = layer->OnUpdate(Time::DeltaTime());
                        if (!Running)
                                break;

                        metrics->GetStats().timeStats.frameTime = Time::DeltaTime();
                        metrics->GetStats().timeStats.frameRate = 1000.f / Time::DeltaTime();

                        layer->OnRender(Time::DeltaTime());

                        layer->EndFrame();
                }
                SDL_GL_SwapWindow(Window);
        }

        OnCleanup();

        return 0;
}

void GLAPIENTRY MessageCallback(GLenum source,
        GLenum type,
        GLuint id,
        GLenum severity,
        GLsizei length,
        const GLchar* message,
        const void* userParam)
{
        if (type != GL_DEBUG_TYPE_OTHER)
        {
                //if (severity != GL_DEBUG_SEVERITY_LOW && severity !=
GL_DEBUG_SEVERITY_MEDIUM)
                ONYX_CORE_TRACE("----------------------opengl-callback-start------------");
                std::string msg = message;
                ONYX_CORE_TRACE("message: " + msg);
                switch (type) {
                case GL_DEBUG_TYPE_ERROR:
                        ONYX_CORE_TRACE("Type: ERROR");
                        break;
                case GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR:
                        ONYX_CORE_TRACE("Type: DEPRECATED_BEHAVIOR");
                        break;
                case GL_DEBUG_TYPE_UNDEFINED_BEHAVIOR:
                        ONYX_CORE_TRACE("Type: UNDEFINED_BEHAVIOR");
                        break;
                case GL_DEBUG_TYPE_PORTABILITY:
                        ONYX_CORE_TRACE("Type: PORTABILITY");
                        break;
                case GL_DEBUG_TYPE_PERFORMANCE:
                        ONYX_CORE_TRACE("Type: PERFORMANCE");
                        break;
```

```
		case GL_DEBUG_TYPE_OTHER:
			ONYX_CORE_TRACE("Type: OTHER");
			break;
		}
		ONYX_CORE_TRACE("");

		ONYX_CORE_TRACE("id: " + std::to_string(id));
		switch (severity) {
		case GL_DEBUG_SEVERITY_LOW:
			ONYX_CORE_TRACE("Severity: LOW");
			break;
		case GL_DEBUG_SEVERITY_MEDIUM:
			ONYX_CORE_TRACE("Severity: MEDIUM");
			break;
		case GL_DEBUG_SEVERITY_HIGH:
			ONYX_CORE_TRACE("Severity: HIGH");
			break;
		}
		ONYX_CORE_TRACE("");
		ONYX_CORE_TRACE("----------------------opengl-callback-end--------------");
	}
}

bool App::OnInit(const uint _SCREEN_WIDTH, const uint _SCREEN_HEIGHT)
{
	SCREEN_WIDTH = _SCREEN_WIDTH;
	SCREEN_HEIGHT = _SCREEN_HEIGHT;

	if (SDL_Init(SDL_INIT_VIDEO) < 0)
		return false;
	else
	{
		Window = SDL_CreateWindow(
			"Onyx",
			SDL_WINDOWPOS_UNDEFINED,
			SDL_WINDOWPOS_UNDEFINED,
			SCREEN_WIDTH,
			SCREEN_HEIGHT,
			SDL_WINDOW_OPENGL | SDL_WINDOW_RESIZABLE
		);
		if (Window == nullptr)
		{
			ONYX_CORE_FATAL("Window could not be initialized.");

			return false;
		}
		else
		{
			GLContext = SDL_GL_CreateContext(Window);
			SDL_GL_MakeCurrent(Window, GLContext);

			glewExperimental = GL_TRUE;
			if (glewInit() != GLEW_OK)
			{
				ONYX_CORE_ERROR("Could not initialize GLEW");

				return false;
			}
			else
			{
```

```
                                glEnable(GL_CULL_FACE);
                                glEnable(GL_DEPTH_TEST);
                                glEnable(GL_TEXTURE_2D);

                                SDL_GL_SetSwapInterval(1);
                                glEnable(GL_DEBUG_OUTPUT);
                                glDebugMessageCallback(MessageCallback, 0);

                                PushModules();
                        }
                }
        }

        return true;
}

void App::OnEvent(SDL_Event* Event)
{
        for (auto& layer : m_LayerStack)
                layer->OnEvent(Event);
}

void App::PushLayer(Layer* layer)
{
        m_LayerStack.PushLayer(layer);
        layer->OnAttach();
}

void App::PushOverlay(Layer* layer)
{
        m_LayerStack.PushOverlay(layer);
        layer->OnAttach();
}

void App::OnCleanup()
{
        ONYX_CORE_INFO("Exit program..");
        SDL_GL_DeleteContext(GLContext);
        SDL_DestroyWindow(Window);
        SDL_Quit();
}

void App::ResizeWindow(const uint width, const uint height)
{
        SCREEN_WIDTH = width;
        SCREEN_HEIGHT = height;
        for (Layer* layer : m_LayerStack)
                layer->OnResize();
        glViewport(0, 0, width, height);
}
```