Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп’ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

бакалавра
(назва освітньо-кваліфікаційного рівня)

студента *Харченко Дмитро Олександрович*
(ПІБ)

академічної групи *122-19-1*
(шифр)

спеціальності *122 Комп’ютерні науки*
(код і назва спеціальності)

освітньої програми *Комп’ютерні науки*
(назва освітньої програми)

на тему: *Розробка ігрового рушія для 2D ігор з використанням технології C++ SFML*

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтингово ю	інституційн ою	
кваліфікаційної роботи				
розділів:				
спеціальний	*доц. Кабак Л.В.*			
економічний	*проф. Вагонова О.Г.*			
Рецензент				
Нормоконтролер	*доц. Гуліна І.Г.*			

Дніпро
2023

Міністерство освіти і науки України
НТУ «Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
завідувач кафедри
програмного забезпечення комп’ютерних систем
(повна назва)

______________ М.О. Алексєєв
(підпис) (прізвище, ініціали)

« » ____________ 2023 року

ЗАВДАННЯ

на кваліфікаційну роботу

бакалавра

(назва освітньо-кваліфікаційного рівня)

студента *122-19-1* *Харченко Д.О.*
(група) (прізвище та ініціали)

тема кваліфікаційної роботи *Розробка ігрового рушія для 2D ігор з використанням технології С++ SFML*

затверджена наказом ректора НТУ «ДП» від *16.05.2023* № *350-с*

Розділ	Зміст виконання	Термін виконання
Спеціальний	***На основі матеріалів проєктно-технологічної практики та інших науково-технічних джерел провести аналіз стану рішення проблеми та постановку задачі.*** ***Обґрунтувати вибір та здійснити реалізацію методів вирішення проблеми***	***13.05.2023 р.***
Економічний	***Провести розрахунок трудомісткості розробки програмного забезпечення, витрат на створення ПЗ й тривалості його розробки***	***27.05.2023 р.***

Завдання видав ____________ *Доц. Кабак Л.В.*
(підпис) (посада, прізвище, ініціали)

Завдання прийняв до виконання ____________ *Харченко Д.О.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *14.01.2023 р.*

Термін подання кваліфікаційної роботи до ЕК: *12,06.2023 р.*

РЕФЕРАТ

Об’єкт розробки: ігровий рушій для 2D ігор з використанням технології C++ SFML.

Мета кваліфікаційної роботи: розробика ігрового рушія за допомогою технології SFML C++ з метою надання розробникам зручної системи для програмування ігрових застосунків.

У вступі описується тема та предметна галузь роботи, конкретизуються технології використані у процесі створення системи, конкретизується мета та описуються очікувані результати роботи.

У першому розділі сформульоване завдання та призначення роботи, встановлені вимоги до програмного виробу.

У другому розділі проаналізована створена система, виділені та описані ключові компоненти програми, алгоритми їх дії, роз’яснений порядок використання системи та типи вхідних даних, надані приклади роботи та використання ігрового рушія.

У третьому економічному розділі прораховано вартість роботи по створенню програми враховуючи трудомісткість та час витрачений на розробку системи.

Практичне значення роботи полягає у створенні системи для спрощення розробки комп’ютерних 2d ігор, надання інструментів та абстракцій для кращої організації проєктів, програм і систем розробників 2d ігор.

Актуальність інформаційної системи на пряму пов’язана зі збільшенням попиту на комп’ютерні ігри, що у свою чергу означає прискорення розвитку індустрії комп’ютерних ігор та попит на інструменти які пришвидшують та будь-яким чином спрощують, або організовують процес розробки ігрових застосунків.

ABSTRACT

Object of development: game engine for 2D games created using C++ and SFML

Purpose of work: development of the game engine using C++ and SFML technologies as a way of giving developers a convenient system for programming and creating games.

Introduction highlights the theme and subject field of the work, specifies technologies used in the system development process, the purpose and desired results are described.

First chapter formulates the task and the motive of work, sets clear requirements for the outcome product.

Second chapter analyzes the created system, highlights and describes key system components and abstractions, their algorithms, explains the minimal steps needed to use the system and types of input data needed, provides examples of usage and capabilities of the game engine.

Third chapter is concerned with economic value of the work, it provides the calculations of the price of developed system based on labor and time spent on the creation.

Practical purpose of the work is the creation of the system to ease the development process of 2d games, granting the instruments and abstractions for better organization of the projects and programs for 2d game creators.

Topicality of the system is directly connected with the increasing demand for computer games, which in turn means acceleration of game development industry and greater demand for instruments which enchance, speed up and in any way ease or organize the process of game development.

ЗМІСТ

РЕФЕРАТ……………………………………………………………………………..3

ABSTRACT…………………………………………………………………………..4

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ………………………………………………7

ВСТУП……………………………………………………………………………… 8

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1. Загальні відомості з предметної галузі………………………………… 10

1.2. Призначення розробки та галузь застосування……………………….. 10

1.3. Підстава для розробки……………………………………………….. 11

1.4. Постановка завдання……………………………………………………… 11

1.5. Вимоги до програми або програмного виробу…………………………. 13

1.5.1. Вимоги до функціональних характеристик……………………………… 13

1.5.2. Вимоги до інформаційної безпеки……………………………………… 14

1.5.3. Вимоги до складу та параметрів технічних засобів……………………… 14

1.5.4. Вимоги до інформаційної та програмної сумісності …………………… 14

РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ..

2.1. Функціональне призначення системи……………………………………….. 15

2.2. Опис застосованих математичних методів……………………………… 16

2.3. Опис використаних технологій та мов програмування………………… 17

2.4. Опис структури системи та алгоритмів її функціонування ……………… 18

2.5. Обґрунтування та організація вхідних та вихідних даних програми…. 41

2.6. Опис розробленої системи ....………………………………………… 41

2.6.1. Використані технічні засоби…….……………………………………… 41

2.6.2. Використані програмні засоби…………………………………………….. 41

2.6.3. Виклик та завантаження програми………………………………………… 42

2.6.4. Опис інтерфейсу користувача……………………………………………… 43

РОЗДІЛ 3. ЕКОНОМІЧНИЙ РОЗДІЛ………………………………………………

3.1. Розрахунок трудомісткості та вартості розробки програмного продукту…45

3.2. Рахунок витрат на створення програми…………………………………… 49
ВИСНОВКИ………………………………………………………………………… 51
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ………………………………………… 52
Додаток А. Код програми…………………………….……………………….53

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ІС – інформаційна система.

ПК – персональний комп’ютер.

SFML – Simple and Fast Multimedia Library.

ВСТУП

Темою даної дипломної роботи є розробка ігрового рушія для 2D ігор з використанням технології C++ SFML. Ігровий рушій це фреймворк здебільшого створений для розробки ігор і зазвичай включає в себе релевантні бібліотеки та допоміжні програми. Розробники використовують ігрові рушії з метою створення ігор для ігрових консолей та інших типів комп’ютерів.

Ігрові рушії зазвичай надають вівзуальні інструменти та програмні компоненти для розробки. Ключовий функціонал ігрових рушіїв може включати графічний рушій для 2D та 3D графіки, рушій фізики, менеджер колізій, інтерфейси роботи зі звуком, анімаціями, мережею, менеджери пам’яті, підтримка локалізації та кінематографії. Додатковий функціонал може включати платформну абстракцію, яка дозволяє без зайвих зусиль запускати ігри на різних платформах.

В цілому процес розробки ігор складається з великої кількості процесів: графічний дизайн, дизайн сюжету, аудіо розробка, дизайн ігрових рівнів, дизайн анімацій, та програмування. З технічної точки зору, процес розробки ігор може бути описаний як використання або адаптація можливостей ігрового рушія (або чистої мови програмування) для створення конкретної гри або портування її на множину платформ. Мета даної дипломної роботи саме полягає в створенні системи для спрощення технічного процесу створення ігор.

SFML – біблотека написана на C++ яка надає прості інтерфейси для взаємодії з мультимедійними компонентами системи, наприклад, класи для роботи з графікою, звуком та мережею та деякі допоміжні класи для роботою з системою. Використання SFML для створення проекту надає можливість сфокусуватись на програмному функціоналі та структурі рушія водночас маючи можливість розширення наданого бібліотекою низькорівневого функціоналу.

У процесі розробки ігрового рушія будуть використані принципи ООП (інкапсуляція, абстракція, наслідування, поліморфізм), та деякі принципи функціонального програмування.

Робота буде здійснюватись з використанням C++ стандарту ISO/IEC 14882:2020 (C++20), бібліотеки SFML для використання мультимедійних компонентів системи, інструметну “quom” для пакування коду рушія у один зручний файл та програми “Tiled” для створення та опису ігрових рівнів. Також буде створено масив прикладів використання ключових елементів ігрового рушія та кілька прикладів простих 2D ігор.

Основною метою розробки ігрового рушія є створення простої та інтуїтивно зрозумілої системи для створення просих 2D ігор.

Результатом цієї роботи буде бібліотека – ігровий рушій надаючий ключові ресурси, структуру та систему для створення 2D ігор.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1 Загальні відомості з предметної галузі

2D ігри засновані на двовимірній графіці, де об'єкти та персонажі представлені у вигляді плоских спрайтів (текстур). Існує багато жанрів 2d ігор, наприклад: платформери, аркади, головоломки, RPG, стратегії та інші.

Ігровий рушій (іноді також називають ігровим двигуном) є базовим інструментарієм для розробки ігор. Ігрові рушії надають розробникам набір інструментів і функціонал для програмування фізики, роботи з графікою, аудіо, управління введенням, роботи з рівнями та багато іншого.

Розробка ігрового рушія включає створення архітектури, яка дозволяє ефективно розробляти складові гри. Основною метою розробки є створення модульної та розширюваної системи, яка полегшує розробку ігор та сприяє повторному використанню коду.

SFML є потужною, кросплатформеною та легкою у використанні бібліотекою для розробки мультимедійних додатків. Вона надає простий доступ до основних мультимедійних функцій, таких як графіка, аудіо, мережа та введення користувача що є зручним інтерфейсом для побудови ігрового рушія з використанням мови програмування C++.

1.2 Призначення розробки та галузь застосування

Назва системи: «Ігровий рушій для 2d ігор з використанням технології SFML C++»

Термінологія:

1. Ігровий рушій — інструментарій для розробки ігор.
2. SFML — кросплатформена бібліотека для розробки мультимедійних додатків.

Причини виникнення необхідності розробки програмного забезпечення:

1. Зростаюча популярність ігрових застосунків і програм.

2. Потреба у створенні ефективної системи яка задовольняє умови розробників ігор.
Галузь застосування: ігрова індустрія.

1.3 Підстава для розробки

Відповідно до освітньої програми, згідно навчального плану та графіків навчального процесу, в кінці навчання студент виконує кваліфікаційну роботу.

Тема роботи узгоджується з керівником проекту, випускаючою кафедрою, та затверджується з наказом ректора.

Таким чином підставами для розробки (виконанням кваліфікаційної роботи) є:

– освітня програма спеціальності 122 «Комп'ютерні науки»;

– навчальний план та графік навчального процесу;

– наказ ректора Національного технічного університету «Дніпровська політехніка» № 317-с від 07.06.2021 р;

– завдання на кваліфікаційну роботу на тему «Розробка ігрового рушія для 2d ігор з використанням технології SFML C++».

1.4 Постановка завдання

Мета: Розробити ігровий рушій за допомогою технології SFML C++ з метою надання розробникам зручної системи для програмування ігрових застосунків.

Призначення: Розроблена ІС має забезпечити розробників інструментами та структурою для створення 2d ігор.

Техніко-економічна сутність: Розробка ігрового рушія за допомогою технології SFML є доцільним рішенням, оскільки вона надає можливість сфокусуватись на програмному функціоналі та структурі рушія водночас залишаючи за розробниками можливість розширення наданого бібліотекою низькорівневого функціоналу.

Об’єкти використання системи: Система буде використовуватись ентузіастами та професіоналами сфери розробки ігор.

Структура об’єктів інформаційної системи: Система складається з масиву абстракцій ключового функціоналу присутнього у більшості ігрових застосунків, а саме: менеджеру ресурсів (текстур, шрифтів, аудіо), менеджеру анімацій, менеджеру графічного інтерфейсу, менеджуру контролерів, менеджеру фізики, графіки та колізій, менеджеру об’єктів дебагу та деяких надбудов для визначення та спрощення роботи з ігровим рушієм.

Призначення вихідної інформації: Система за потреби може надавати визначенні дані для спрощення створення, тестування та дебагу ігрових застосунків (напириклад границі колізій об’єктів, візуальне відображенні фізичних векторів, цифрове відображення значень заданих змінних тощо.)

Вимоги до організації збору та передачі в обробку вхідної інформації: Система має надавати зручну абстракцію для роботи з засобами вводу (контролерами).

Умови припинення розв’язання завдання автоматизованим способом: Розв’язання завдання автоматизованим способом припиниться у випадку неможливості доступу до мультимедійних ресурсів (текстури, шрифти, аудіо), деякі критичні логічні програмні помилки допущені при використанні коду системи.

1.5 Вимоги до програми або програмного виробу

1.5.1 Вимоги до функціональних характеристик

Вимоги до функціональних характеристик ігрового рушія включають:

1. Менеджер медіа ресурсів (текстур, шрифтів та аудіо): система мє надавати інтерфейс завантаження та менеджменту в пам'яті заданих ігрових медіа ресурсів.

2. Менеджер анімацій текстур: система має надавати абстракції для створення та менеджменту анімацій текстур ігрових об'єктів.

3. Менеджер графічного інтерфейсу: система має надавати інструменти та абстракції для створення простих інтерфейсів користувача.

4. Менеджер графічних ігрових об'єктів: систкма має надавати зручний інтерфейс прорисовки графічних об'єктів гри.

5. Менеджер та абтракції фізики: система має реалізовувати рушій фізики та надавати інтерфейси для його використання та розширення.

6. Менеджер, алгоритми та абстракції колізій об'єктів: система має реалізовувати рушій обробки колізій ігрових об'єктів та інтерфейси для його використання та розширення а також алгоритми детекції колізій між об'єктами.

7. Менеджер та абстракції дебагу об'єктів: система має надавати функціонал візуального відображення інформації корисної для тестування, розробки та дебагу ігрових додатків (напириклад границі колізій об'єктів, візуальне відображенні фізичних векторів, цифрове відображення значень заданих змінних тощо.).

8. Менеджер засобів вводу (контролерів): система має надавати універсальний інтерфейс для написання скриптів ігрових контролерів.

9. Менеджер станів об’єктів: система має реалізовувати інтерфейс для роботи з моделлю «Multi state machine»

1.5.2 Вимоги до інформаційної безпеки

Система має належним чином контролювати та проводити маніпуляції з пам’яттю для запобіження витоків пам’яті та аварійних завершень застосунків.

1.5.3 Вимоги до складу та параметрів технічних засобів

Вимоги до складу та параметрів технічних засобів для ігрового рушія включають лише наявність в системі встановленої залежної бібліотеки SFML та достатній ресурсний потенціал ПК відповідно до об’єму навантаження та пам’яті потрібного для конкретної розроблюваної 2d гри.

1.5.4 Вимоги до інформаційної та програмної сумісності

Система має бути сумісна з різними операційними системам (Windows, Linux).

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1 Функціональне призначення системи

Функціональним призначенням розробленої системи є організація та менеджмент ключових об’єктів та структур, які можуть бути використані для розробки 2d ігор.

Функціональними блоками розробленого ігрового рушія є:

- **Assets** - Завантаження та менеджмент мультимедійних ресурсів (текстури, шрифти, аудіо)
- **Controller** - Опис та менеджмент засобів вводу (контролерів)
- **View** - Менеджмент та скриптування ігрових камер
- **Logic** - Ключова логіка системи
  - Опис та менеджмент ігрової фізики
  - Опис та менеджмент колізій та взаємодій між об’єктами
  - Опис та менеджмент станів об’єктів
  - Менеджмент графічних анімацій
  - Менеджмент ігрових об’єктів
  - Менеджмент „сцен“ (колекцій ігрових об’єктів)
- **Debug** - Базові інструменти для дебагу ключової логіки

З точки зору експлуатаційного призначення розроблена система надає розробникам систему менеджменту та масив програмних інструментів для створення 2d ігор.

2.2 Опис застосованих математичних методів

Теорія множин [5]

У менеджері колізій (CollisionManager) (див. Розділ 2.4) реалізовано концепт визначення „фази“ колізій з метою розширення можливостей розробників для написання скриптів взаємодії об’єктів.

Ігровий об’єкт, при колізії з іншим об’єктом, може перебувати у одній з трьох фаз: **start**(момент зіткнення), **continuous**(постійна взаємодія), **end**(момент завершення взаємодії).

Для визначення фази колізій використана теорія множин.

Припустимо існує 2 масиви об’єктів: **past**(об’єкти які перебували у стані колізії під час минулої ітерації) та **present**(об’єкти у стані колізії під час даної ітерації), тоді:

continuous = past ∩ present

start = present - continuous

end = past - continuous

Описана логіка може бути проілюстрована за допомогою наступної діаграми:

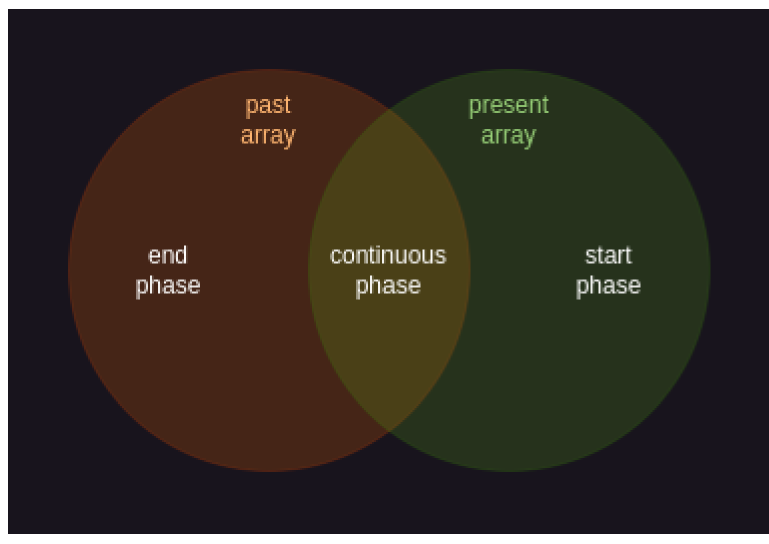

Рис.1 «Теорія множин в CollisionManager»

У CollisionManager описаний функціонал реалізований наступним чином:

```
// Determine collision phase
std::sort(presentCollisions.begin(), presentCollisions.end());
std::sort(pastCollisions.begin(), pastCollisions.end());


std::vector<sge::Collision> continuousPhaseCollisions;
std::set_intersection(
    pastCollisions.begin(),pastCollisions.end(),
    presentCollisions.begin(),presentCollisions.end(),
    std::back_inserter(continuousPhaseCollisions)
);

std::vector<sge::Collision> startPhaseCollisions;
std::set_difference(
    presentCollisions.begin(),presentCollisions.end(),
    continuousPhaseCollisions.begin(),continuousPhaseCollisions.end(),
    std::back_inserter(startPhaseCollisions)
);

std::vector<sge::Collision> endPhaseCollisions;
std::set_difference(
    pastCollisions.begin(),pastCollisions.end(),
    continuousPhaseCollisions.begin(),continuousPhaseCollisions.end(),
    std::back_inserter(endPhaseCollisions)
);
//
```

Рис.2 «Визначення фази колізій»

2.3 Опис викоританих технологій та мов програмування

У процесі створення системи були використані наступні технології:

Мова програмування C++ [1, 6]

- Універсальна мова програмування високого рівня розроблена Б'ярном Страустрпуом на основі мови C.

Обґрунтування вибору:

- C++ надає великі можливості управління пам'яттю, забезпечуючи більше контролю, гнучкості та оптимізації ігрових ресурсів
- C++ сумісний з низькорівневою C і мовою асемблера, що полегшує розробникам ігор взаємодію з компонентами апаратного рівня
- C++ це компільована мова, що забезпечує кращу продуктивність під час виконання програм

Бібліотека SFML [2]

- Крос платформна бібліотека для розробки програмного забезпечення, призначена для забезпечення простого програмного інтерфейсу для різних мультимедійних компонентів комп’ютерів.

Обґрунтування вибору:

- SFML написана на C++, що означає високу швидкість виконання та добру сумісність з C++
- Надані бібліотекою інтерфейси мультимедійних компонетів спрощують створення програмного функціоналу та структури рушія водночас залишаючи за розробниками можливість розширення низькорівневого функціоналу

2.4 Опис структури системи та алгоритмів її функціонування

Логічна структура:

Загальна логічна структура рушія може бути проілюстрована за допомогою наступної діаграми:

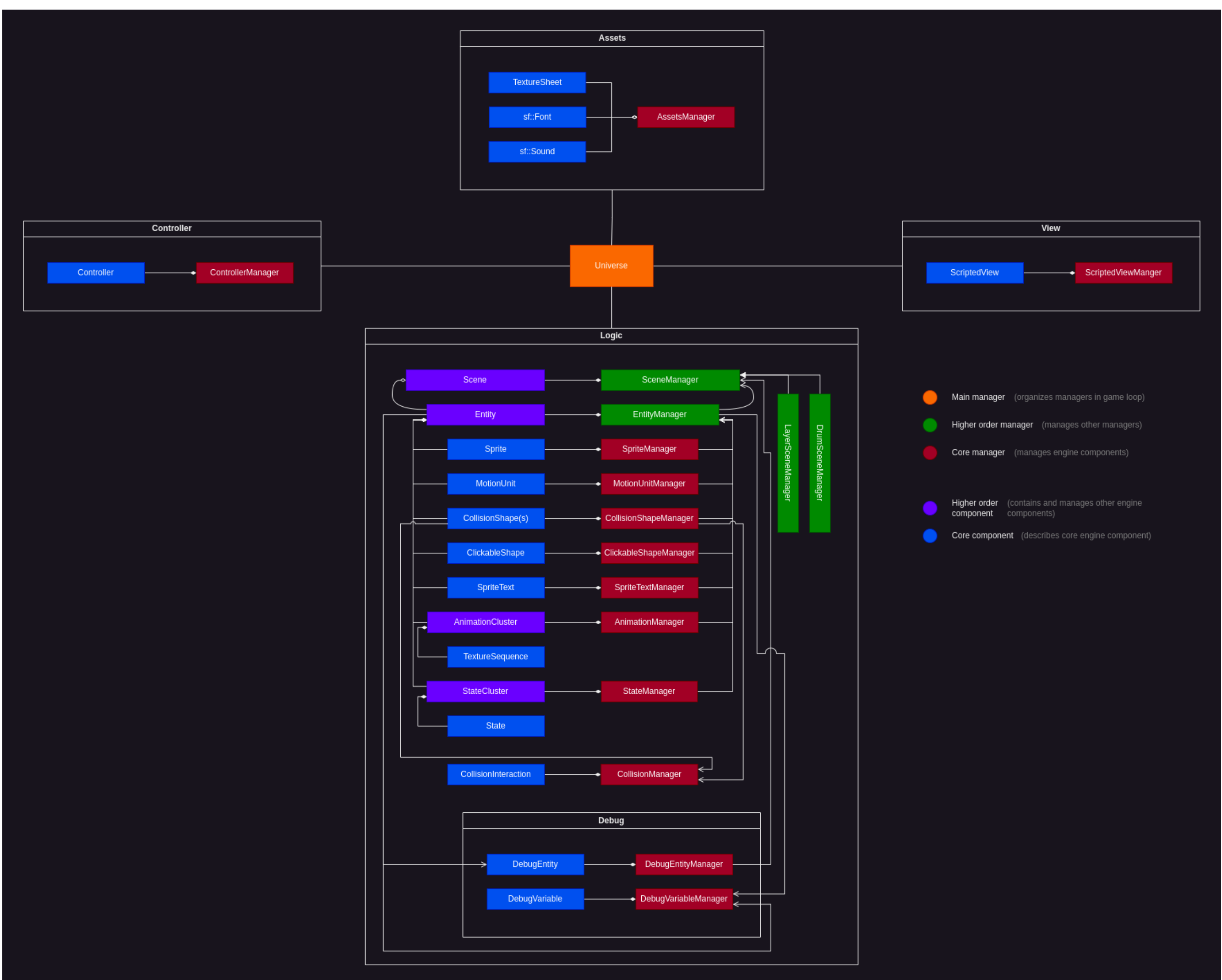

Рис.3 «Структура ігрового рушія»

Структуру, проілюстровану вище, можна назвати **«Component - Manager» (CM)** структурою. Як видно на Рис.2, кожен ключовий компонент системи має свого відповідного менеджера.

Базові поняття

Менеджер компонентів:

Мета менеджера компонентів: зберігання ключових компонентів та їх організація у головному циклі рушія.

Логічно менеджери можна поділити на наступні типи:

Universe

- головний менеджер та саме інтерфейс для використання рушія. Містить в собі головний цикл рушія у якому організовано порядок виконання інших менеджерів.

Менеджер вищого порядку

- складний менеджер мета якого організація відповідних об'єктів а також управління масивом ключових менеджерів

Ключовий менеджер

- простий менеджер мета якого лише організація відповідних об'єктів

Менеджери також поділяються за програмною структурою:

За типом зберігання компонентів:

- VectorManager
- ViewManager
- LabelManager

За секцією циклу яка оброблюється:

- EventManager
- UpdateManager
- DrawManager

Ієрархічний поділ менеджерів може бути проілюстрований за допомогою наступної діаграми:

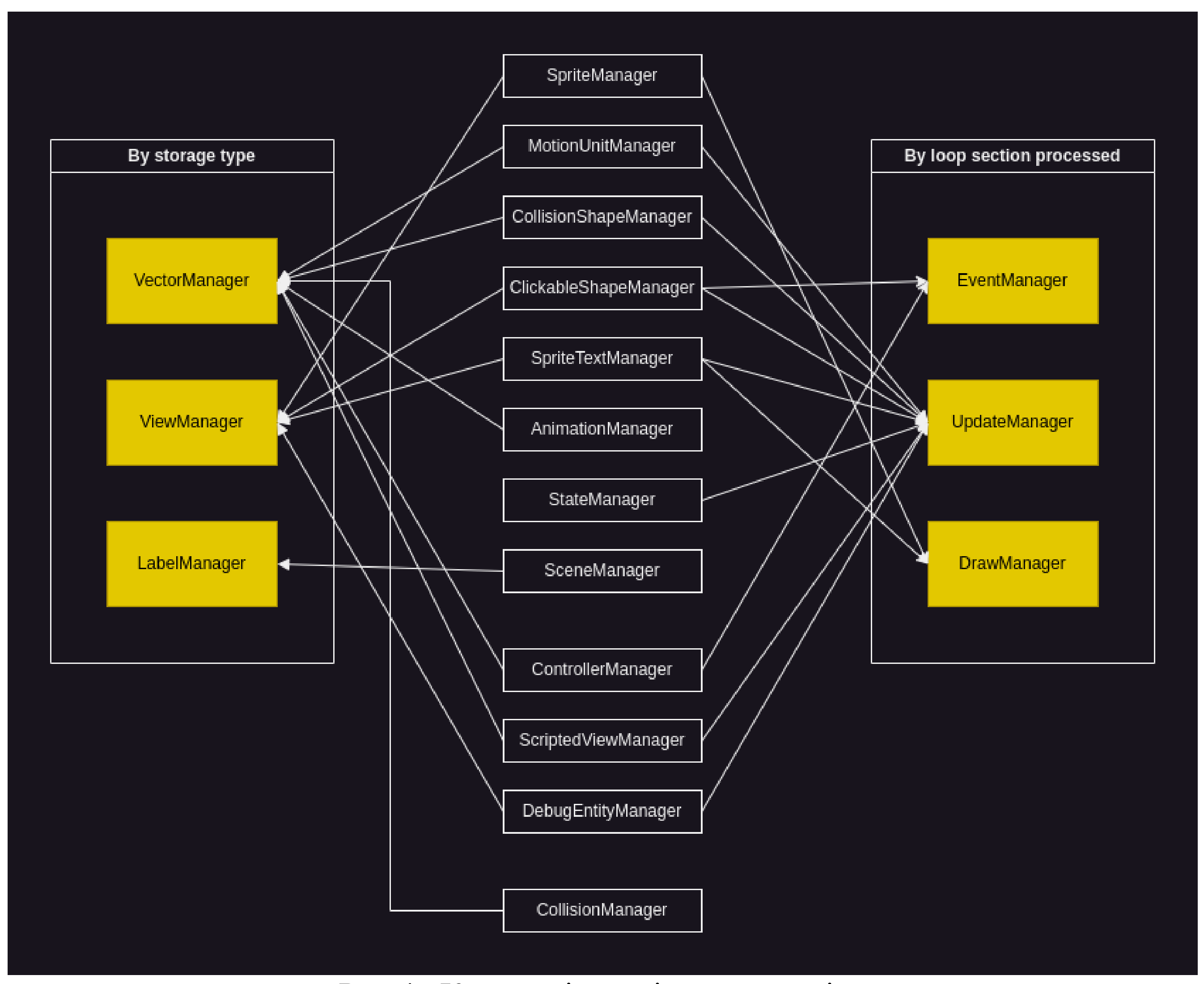

Рис.4 «Класова ієрархія менеджерів»

Ключовий компонент:

Мета ключового компонента: опис конкретної частини функціоналу системи.

Логічно ключові компоненти можна поділити на наступні типи:

Компонент вищого порядку: контейнер для ключових компонентів

Ключовий компонент

Стан компонента

Деякі компоненти (здебільшого у функціональному блоці Logic) реалізовують концепт «стану». Ключовий компонент може перебувати у одному з трьох станів: **active, paused, hidden**.

Стан, у якому перебуває компонент впливає на його обробку (або припинення обробки) менеджерами у головному циклі рушія.

Концепт станів може бути проілюстрований за допомогою наступної діаграми:

Рис.5 «Обробка компонентів в залежності від їх стану»

Головний цикл рушія

Клас Universe, реалізовує головний цикл ігрового рушія.

Головний цикл рушія поділяється на наступні секції:

Events:

- отримання та обробка подій ініційованих користувачем (наприклад обробка кнопок контролера або клавіатури)

Updates:

- оновлення ключових компонентів та об’єктів (наприклад зміна позиції об’єкта гравця у ігровому просторі)

Draws:

- прорисовка ігрових об’єктів

Відношення до менеджерів та компонетів:

Менеджери оброблюють одну або декілька секцій циклу (визначається наслідуванням одного або декількох з класів EventManager, UpdateManager або DrawManager) (див Рис. 3).

Стан компонента визначає чи оброблюватиметься він у заданій секції (див. Рис. 4).

Delta Time [3]

Також у циклі обчислюється delta час (час між ітераціями головного циклу рушія) для корректного встановлення частоти кадрів та для деяких обчислень менеджерів.

Секції циклу та порядок їх обробки менеджерами може бути проілюстрований за допомогою наступної діаграми:

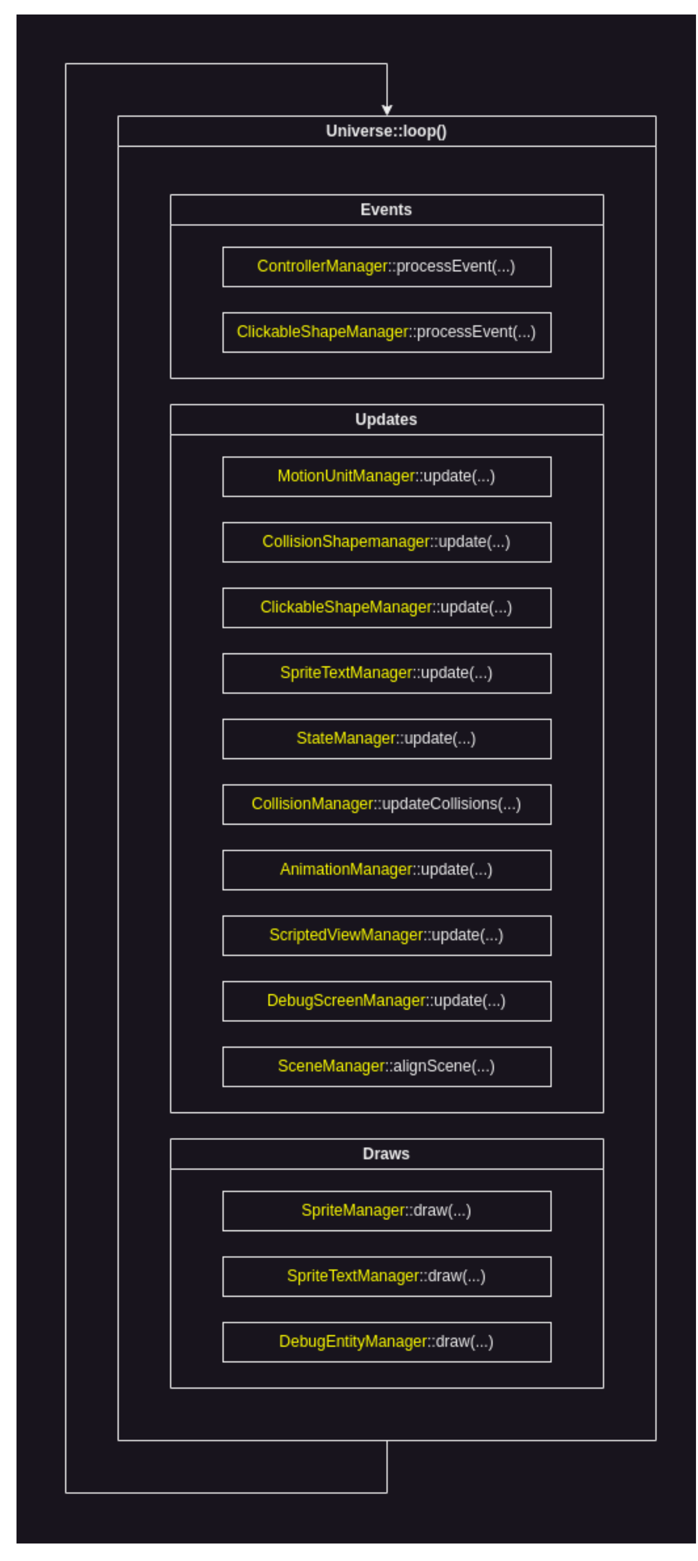

Рис.6 «Головний цикл рушія»

Детальний опис блоків та компонентів системи

Блок Assets:

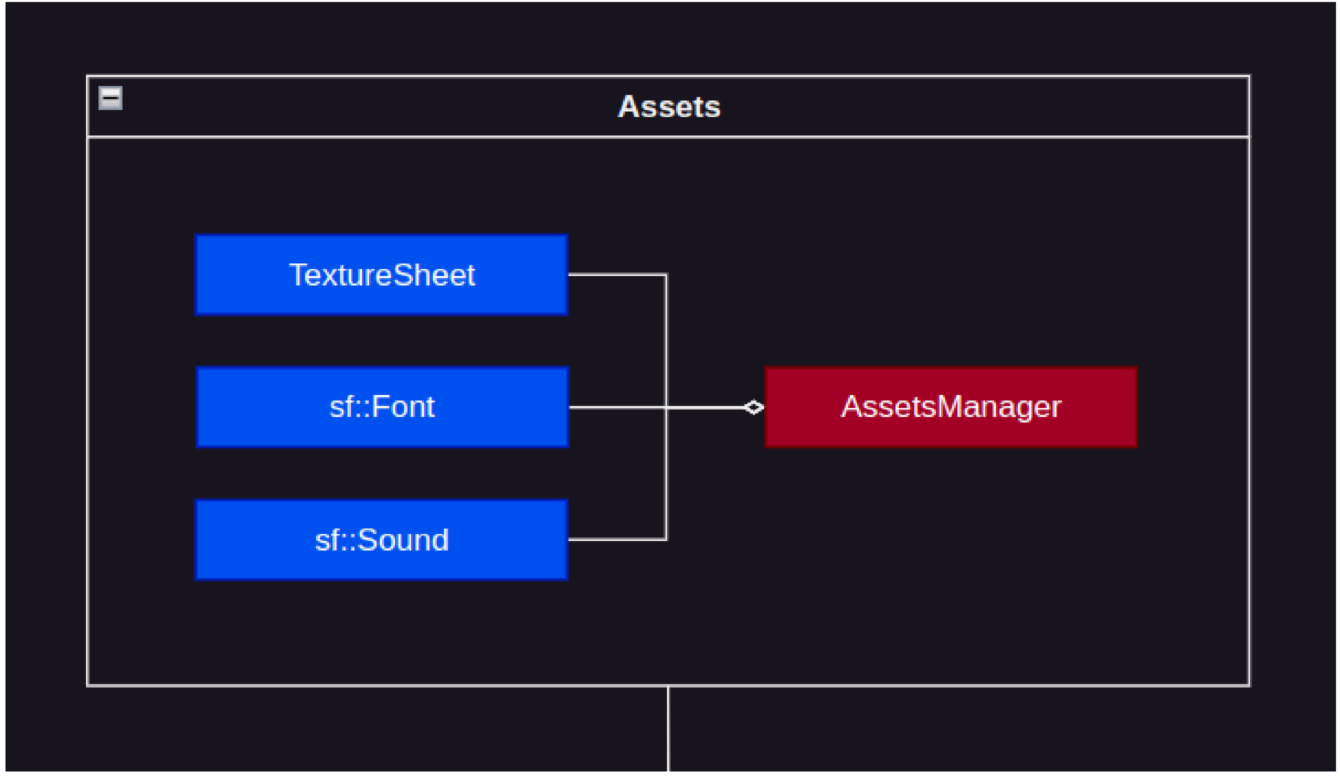

Рис.7 «Складові блоку Assets»

Призначення:

Складові даного блоку відповідають за завантаження, зберігання та менеджмент мультимедійних ресурсів гри

Складові блоку:

TextureSheet — компонент який описує текстурний лист

- Конструктор:

    ініціює масив квадратів текстур за заданими координатими

    зберігає текстуру та локацію текстури

- getLocation():

    повертає локацію текстури

- getTexture():

    повертає текстуру

- getTextureRect(n, …):

    повертає квадрат n — ої текстури

Font — компонент який описує шрифт

Sound — компонент який описує звуковий ефект

AssetsManager — менеджер який саме завантажує та зберігає зазначені ресурси

- loadTextureSheet(…):

    завантажує лист текстур

- loadFont(…)

    завантажує шрифт

- loadSFX(…)

    завантажує звокові ефекти

- specifyMusicLocation(…):

    завантажує локацію аудіофайлу

Приклад використання:

```
// Load assets
universe->assetsManager->loadTextureSheet(
    std::filesystem::current_path().string() + "/examples/dev/assets/tilemap.png",
    "tileset",
    sge::TextureSheetSizes{16, 16, 20, 20}
);
universe->assetsManager->loadFont(
    std::filesystem::current_path().string() + "/examples/dev/assets/m5x7.ttf",
    "m5x7"
);
//
```

Рис.8 «Приклад завантаження текстур»

Блок View:

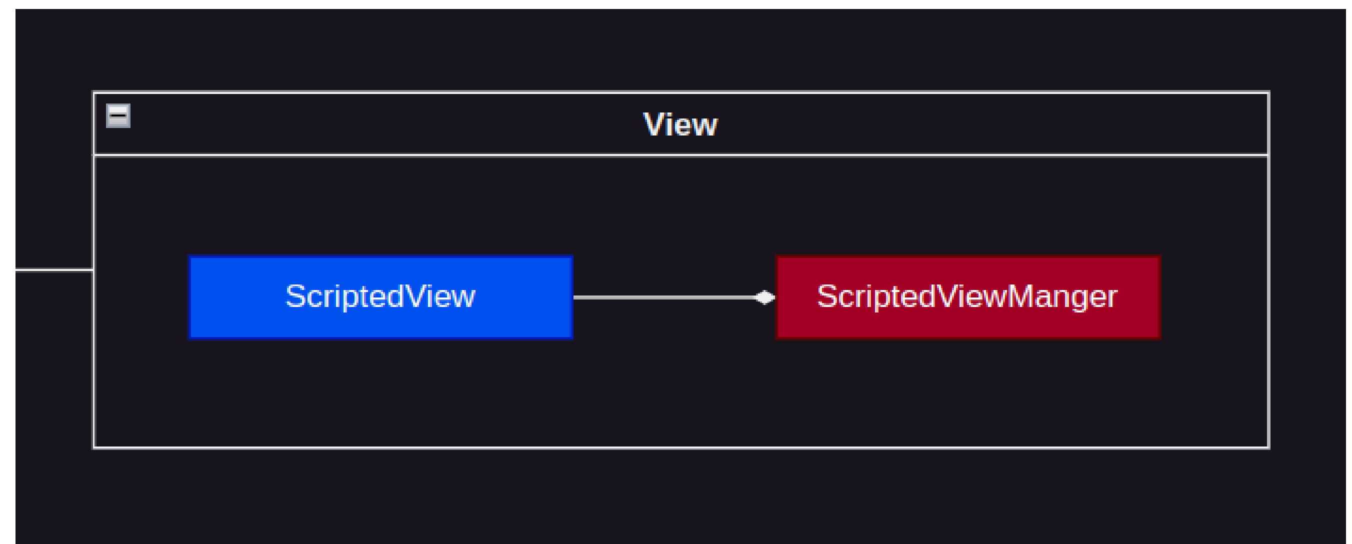

Рис.9 «Складові блоку View»

Призначення:

Складові блоку надають можливість створення скриптів для ігрових камер.

Складові блоку:

ScriptedView — компонент який описує скрипт екрану (камери)

- script():

  скрипт камери

ScriptedViewManager — менеджер який завантажує та включає ScriptedView у головний цикл рушія.

- update():

  ітерує скрипти зареєстрованих камер у циклі рушія

Приклад використання:

```cpp
class CameraView : public sge::ScriptedView{
    public:
        CameraView(sge::Entity* playerEntity) : m_playerEntityPtr(playerEntity){
            this->setCenter(sf::Vector2f(100, 100));
            this->setSize(sf::Vector2f(500, 300));
        };

        void script(){
            sf::Vector2f center = m_playerEntityPtr->sprite->getPosition();
            center.x += 4;
            center.y += 4;

            m_scroll.x = center.x - this->getCenter().x - m_scroll.x / 100;
            m_scroll.y = center.y - this->getCenter().y - m_scroll.y / 100;

            this->setCenter(center - m_scroll);
        }
    private:
        sf::Vector2f m_scroll = sf::Vector2f(0, 0);
        sge::Entity* m_playerEntityPtr;
};
```

Рис.10 «Приклад написання скрипту камери»

Блок Controller:

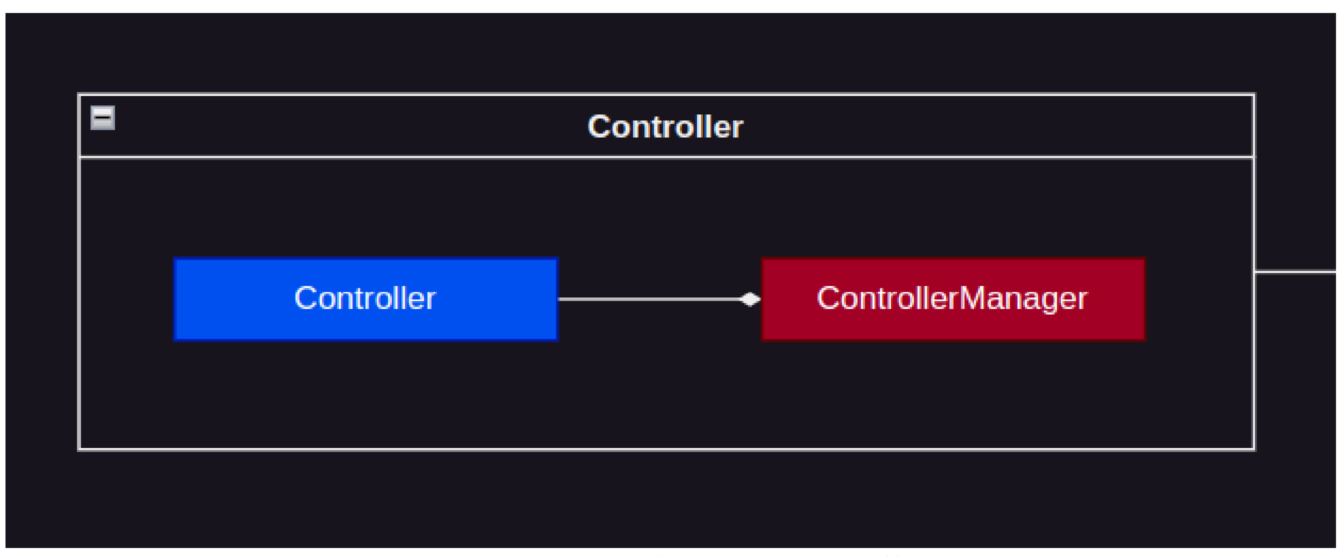

Рис.11 «Складові блоку Controller»

Призначення:

Складові блоку надають можливість створення скриптів для ігрових контролерів та засобів вводу.

Складові блоку:

Controller — компонент який описує контролер

- script():

скрипт контролера

ControllerManager — менеджер який завантажує та включає фунції - контролери у головний цикл рушія.

- processEvent(...):

ітерує скрипти зареєстрованих контролерів у циклі рушія

Приклад використання:

```
class KeyboardController : public sge::Controller{
    public:
        KeyboardController(sge::Entity* playerEntity) : m_playerEntityPtr(playerEntity){};

        void script(sf::Event event) override{
            if(event.type == sf::Event::KeyPressed){
                if(event.key.code == sf::Keyboard::A){
                    m_playerEntityPtr->stateCluster->deactivateState("moving_right");
                    m_playerEntityPtr->stateCluster->activateState("moving_left");
                }
                else if(event.key.code == sf::Keyboard::D){
                    m_playerEntityPtr->stateCluster->deactivateState("moving_left");
                    m_playerEntityPtr->stateCluster->activateState("moving_right");
                }

                if(event.key.code == sf::Keyboard::Space){
                    m_playerEntityPtr->stateCluster->deactivateState("on_ground");
                    m_playerEntityPtr->stateCluster->activateState("jump");
                    if(m_playerEntityPtr->stateCluster->isStateActive("jump")){
                        m_playerEntityPtr->motionUnit->velocity.y = - 200; // ! infinite jump for testing
                    }
                }
            }
            if(event.type == sf::Event::KeyReleased){
                if(event.key.code == sf::Keyboard::A){
                    m_playerEntityPtr->stateCluster->deactivateState("moving_left");
                }
                if(event.key.code == sf::Keyboard::D){
                    m_playerEntityPtr->stateCluster->deactivateState("moving_right");
                }
            }
        }

    private:
        sge::Entity* m_playerEntityPtr;
};
```

Рис.12 «Приклад написання скрипту руху гравця за допомогою клавіатури»

Блок Logic:

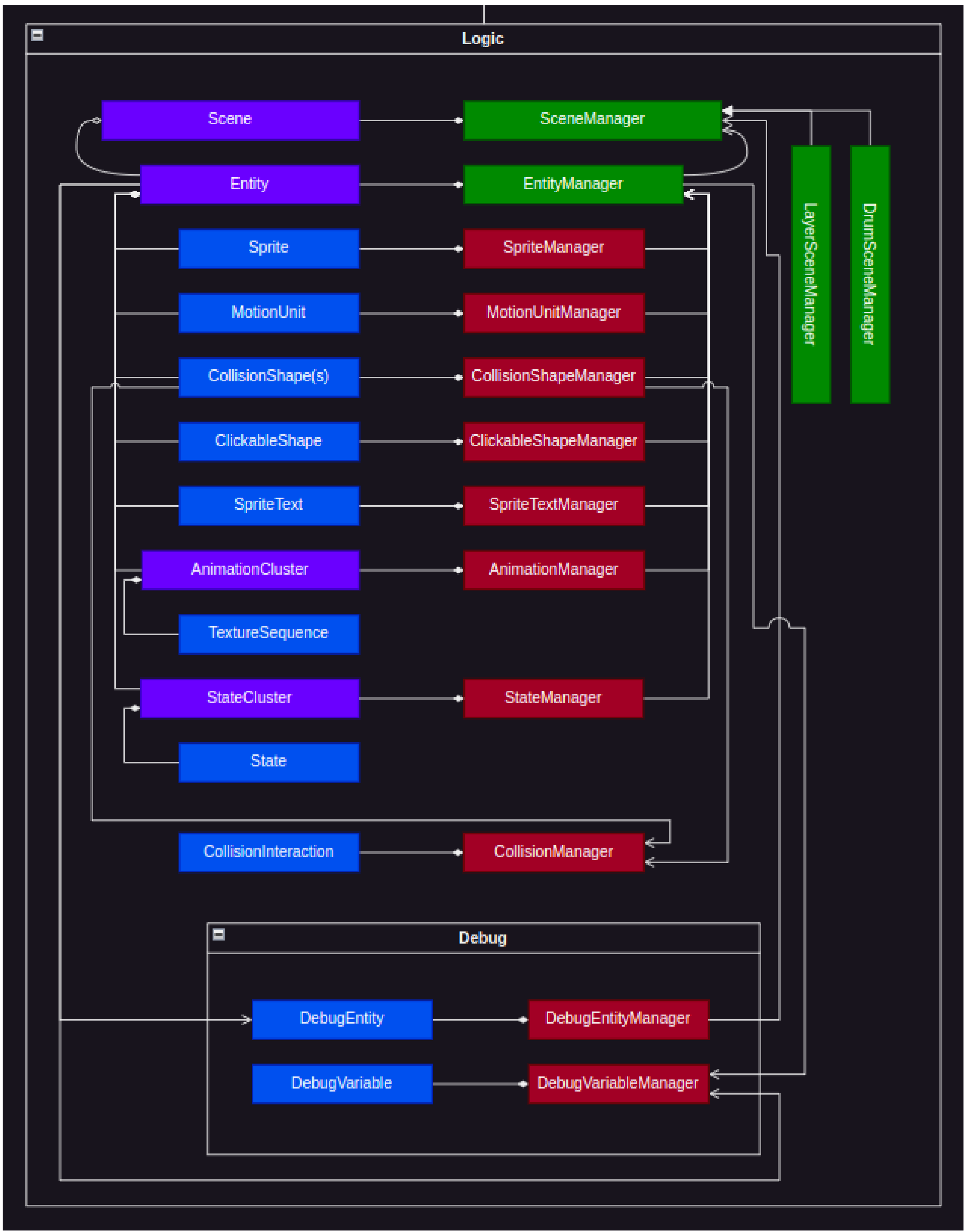

Рис.13 «Складові блоку Logic»

Призначення:

Складові блоку описують головний функціонал ігрового рушія.

Опис:

У даному рушії для створення ігрових об’єктів використовується лише один компонент — Entity. Тобто все, що відображається на екрані є Entity.

Залежно від типу Entity який потрібно створити, розробник має можливість додати до нього наступні складові (Sprite, PhysicalObject, CollisionShape, ClickableShape, SpriteText, Animation, StateCluster). Наприклад, кнопка — це Entity який включає в себе Sprite та ClickableShape, а об’єкт гравця — це Entity який включає в себе Sprite, MotionUnit та CollisionShape. Іншими словами, розробник, залежно від своїх потреб, має свободу створити будь-яку комбінацію об’єктів у компоненті Entity.

Складові блоку:

Sprite - ключова складова Entity. Складові описані нижче мають посилання на Sprite та можуть змінювати його атрибути або визначати свою позицію відносто нього. Sprite описує позицію та текстуру об’єкту.

SpriteManagaer

- draw(...):

малює зареєстровані Sprite на екрані

ClickableShape - складова Entity яка дозволяє створити форму для обробки натискання мишею. При натисканні мишею на форму ClickableShape буде виконана задана розробником функція.

-action():

функція обробки натискання

- offset:

значення зміщення форми відносно Sprite

- align():

вирівнює форму відносно Sprite враховуючи offset

ClickableShapeManager

- update(...):

вирівнює форми зареєстрованих ClickableShape

- processEvent(…):

  ітерує функції обробки натискання зареєстрованих ClickableShape у циклі рушія

SpriteText - складова Entity яка дозволяє додати текст відносно Sprite.

- offset:

  значення зміщення тексту відносно Sprite

- align():

  вирівнює текст відносно Sprite враховуючи offset

SpriteTextManager

- update(...):

  вирівнює текст зареєстрованих SpriteText

- draw(…):

  малює текст зареєстрованих SpriteText на екрані

TextureSequence — компонент який описує послідовність текстур

- sequenceRects:

  квадрати заданих текстур послудовності

- textureSheet:

  текстурний лист послідовності

AnimationCluster — компонент — контейнер TextureSequence. Використовується для створення та групування всіх анімацій деякого конкретного Sprite.

- animationDelayMilliseconds:

  час між кадрами анімації

- addTextureSequence(…):

  додає TextureSequence до контейнеру

- setCurrentTextureSequence(…):

  встановлює анімацію яка наразі ітерується

- getCurrentTextureSequence():

  повертає анімацію яка наразі ітерується

- run():

  ітерує текстури поточної анімації у циклі рушія

AnimationManager

- update(…):

  ітерує текстури всіх зареєстрованих AnimationCluster

State — описує конкретний стан Entity. Може використовуватись для спрощення написання алгоритмів гравця та інших об’єктів гри.

- enterScript():

  скрипт входу в стан

- exitScript():

  скрипт виходу зі стану

- updateScript():

  скрипт який постійно виконується при активному стані

StateCluster — компонент — контейнер State. Використовується для організації станів конкретного Entity.

- getActiveStates():

  повертає всі поточні активні стани

- getActiveStateNames():

  повертає назви всих поточних активних станів

- activateState(…):

  робить заданий стан активним

- deactivateState(…):

  деактивує заданий стан

- isStateActive(…):

  повертає інформацію про активність стану

- states:

  всі стани кластеру

StateManager

- update(…):

  ітерує updateScript всіх активних Stat

Приклад використання станів для опису руху гравця:

```
stateCluster = new sge::StateCluster();
stateCluster→states["on_ground"] = new PlayerOnGroundState(this);
stateCluster→states["jump"] = new PlayerJumpState(this);
stateCluster→states["moving_right"] = new PlayerMovingRightState(this);
stateCluster→states["moving_left"] = new PlayerMovingLeftState(this);
stateCluster→activateState("jump"); // jump because player is initially falling
```

```
class KeyboardController : public sge::Controller{
    public:
        KeyboardController(sge::Entity* playerEntity) : m_playerEntityPtr(playerEntity){};

        void script(sf::Event event) override{
            if(event.type == sf::Event::KeyPressed){
                if(event.key.code == sf::Keyboard::A){
                    m_playerEntityPtr→stateCluster→deactivateState("moving_right");
                    m_playerEntityPtr→stateCluster→activateState("moving_left");
                }
                else if(event.key.code == sf::Keyboard::D){
                    m_playerEntityPtr→stateCluster→deactivateState("moving_left");
                    m_playerEntityPtr→stateCluster→activateState("moving_right");
                }

                if(event.key.code == sf::Keyboard::Space){
                    m_playerEntityPtr→stateCluster→deactivateState("on_ground");
                    m_playerEntityPtr→stateCluster→activateState("jump");
                    if(m_playerEntityPtr→stateCluster→isStateActive("jump")){
                        m_playerEntityPtr→motionUnit→velocity.y = - 200; // ! infinite jump for testing
                    }
                }
            }
            if(event.type == sf::Event::KeyReleased){
                if(event.key.code == sf::Keyboard::A){
                    m_playerEntityPtr→stateCluster→deactivateState("moving_left");
                }
                if(event.key.code == sf::Keyboard::D){
                    m_playerEntityPtr→stateCluster→deactivateState("moving_right");
                }
            }
        }

    private:
        sge::Entity* m_playerEntityPtr;
};
```

Рис.14 «Стани руху гравця»

MotionUnit - складова Entity яка додає скрипти та змінні для обробки фізики. MotionUnit включає в себе набір змінних які описують фізичні властивості об’єкту (швидкість, прискорення) та надає можливість маніпулювати позицією Sprite в залежності від значення зазначених змінних. Дана комбінація змінних зі скриптом надає можливість створення таких фізичних явищ як гравітація, імпульс, сила тертя та ін.

- velocity:

значення бистроти

- acceleration:

значення прискорення

- extraProperties:

контейнер для додаткових фізичних властивостей

- constantForces:

контейнер постійних сил

- fieldForce:

контейнер сил фіз. Поля

- addComputationScript(…):

додає скрипт обчислення фізики

- update(…):

ітерує всі скрипти обчислення у заданому порядку

MotionUnitManager

-update(…):

ітерує скрипти обчислення всіх зареєстрованих MotionUnit

CollisionShape - компонент який описує форму для обчислення колізій.

- offset:

значення зміщення форми відносно Sprite

- collisionGroups:

масив груп до яких належить дана форма

- getMeasurements():

повертає дані про форму

- align():

вирівнює форму відносно Sprite враховуючи offset

CollisionShapeManager

- getComponentsByCollisionGroup(…):

повертає всі компоненти заданої групи

- update(…):

вирівнює всі зареєстровані CollisionShape

CollisionInteractions — компонент який описує взаємодію між групами CollisionShape

- initiatorGroups:

контейнер назв груп ініціаторів колізій

- recipientGroups:

контейнер назв груп сприймачів колізій

- collisionDetectionAlgorithm(...):

функція алгоритму детекції колізії між об’єктами груп CollisionShape

- startPhaseCollisionResponse(…):

скрипт початку колізії

- continuousPhaseCollisionResponse(…):

скрипт постійної колізії

- endPhaseCollisionResponse(…):

скрипт закінчення колізії

- pastCollisions

колізії попередньої ітерації

CollisionManager

- updateCollisions():

оброблює всі зареєстровані CollisionInteraction.

Entity — компонент який описує об’єкт гри

- sprite: див. Sprite
- motionUnit: див. MotionUnit
- collisionShapes: масив CollisionShape (див. CollisionShape)
- clickableShape: див. ClickableShape
- spriteText: див. SpriteText
- animationCluster: див. AnimationCluster
- stateCluster: див. StateCluster

- activateEntityParts():

  переводить всі складові Entity у стан active
- pauseEntityParts():

  переводить всі складові Entity у стан paused
- hideEntityParts():

  переводить всі складові Entity у стан hidden

EntityManager

- registerComponent(…):

  рекурсивно реєструє компоненти які включає Entity

- deregisterComponent(…):

  рекурсивно видаляє компоненти які включає Entity

Scene — компонент - контейнер для Entity. Може використовуватись для опису ігрових рівнів чи екранів інтерфейсу.

- addEntity(...), addEntities(…):

  додають Entity до контейнеру
- addDebugEntity(…), addDebugEntity(…):

  додають DebugEntity до контейнеру
- activateSceneParts():

  переводить всі складові Scene у стан active
- pauseSceneParts():

переводить всі складові Scene у стан paused

- hideSceneParts():

переводить всі складові Scene у стан hidden

SceneManager

- setupDebug(…):

ініціює функціонал дебагу

DrumSceneManager — менеджер у якому, у даний момент часу, лише одна Scene може бути активна

- setCurrentScene(…), alignScene():

встановлює активну Scene

LayerSceneManager — менеджер у якому кілька Scene можуть бути активними одночасно

Блок Debug:

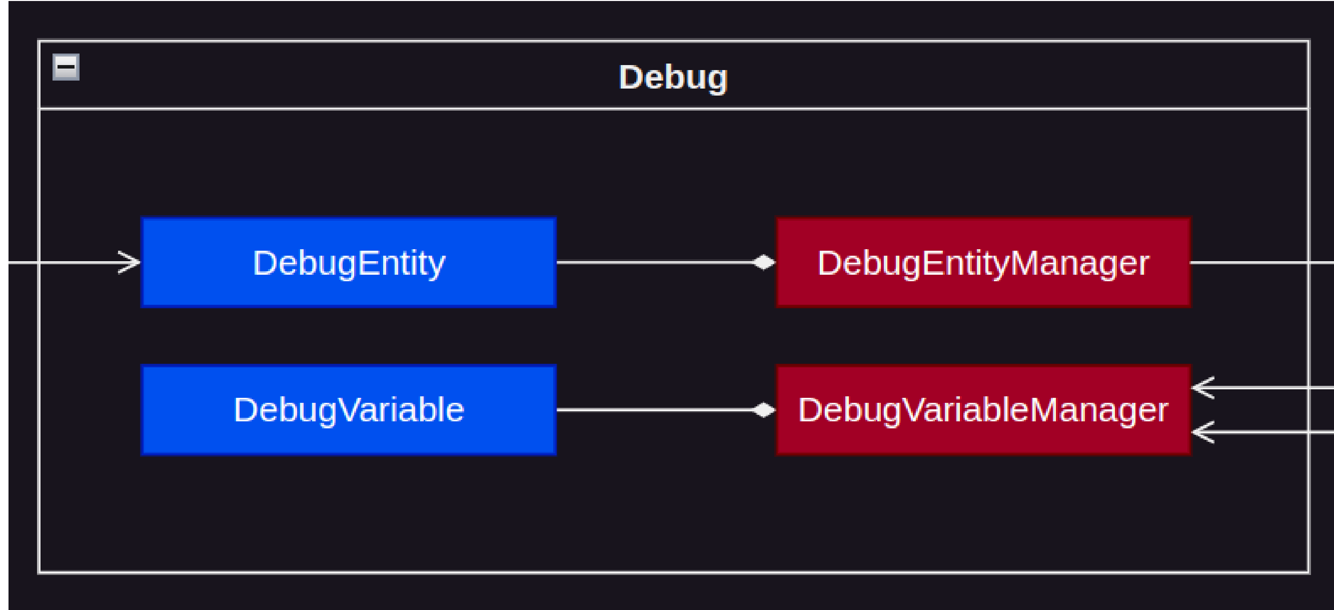

Рис.15 «Складові блоку Debug»

Призначення:

Складові блоку описують дебаг функціонал ігрового рушія.

Складові блоку:

DebugEntity — парний компонент для дебагу конкретних об’єктів Entity.

- costomCollisionShapeBorderSettings:

  контейнер налащтувань CollisionBorder

- generateCollisionBorders():

  генерує рамку навколо CollisionShape об’єкту

- addExtraDebugFunction(…):

  додає додаткові функції для дебагу

- getExtraDebugFunctions():

  повертає додаткові функції для дебагу

DebugEntityManager

- draw(…):

  ітерує додаткові функції та малює CollisionShapeBorder на екрані

DebugVariable — змінна, яка описує деяку дебаг інформацію.

- valueUpdateScript():

  скрипт оновлення значення змінної

- value

  значення змінної

DebugScreenManager

- addDebugVariable(…):

  додає DebugVariable до контейнеру

- getDebugVariables():

  повертає всі зареєстровані DebugVariable

- updateDebugVariables():

  оновлює всі зареєстровані змінні дебагу

2.5 Обґрунтування та організація вхідних та вихідних даних програми

Вхідні дані системи представляють собою будь-які події (натискання) ініційовані пристроями вводу (клавіатура, миша). Зазначені події кодуються бібліотекою SFML.

2.6 Опис розробленої системи

Система є набором інструментів та функціоналу призначеного для спрощення розробки 2d ігор.

2.6.1 Використані технічні засоби

Так як система є набором інструментів, для роботи системи може бути використана будь-яка машина з наявною в системі встановленою залежною бібліотекою SFML та достатнім ресурсним потенціалом машини відповідно до об'єму навантаження та пам'яті потрібного для конкретної розроблюваної 2d гри.

2.6.2 Використані програмні засоби

Робота системи протестована на машині з операційною системою Ubuntu 22.04.2 LTS.

Для компіляції програм написаних за допомогою системи використано компілятор gcc 11.3.0 та інструмент CMake.

2.6.3 Виклик та завантаження програми

Для використання системи розробник повинен виконати наступний мінімальний набір дій:

1. Додати файл з кодом рушія «SGE.hpp» до свого проєкту.

2. Включити файл рушія до свого проєкту:

```
#include "../../SGE.hpp"
```

Рис.16 «Приклад коду включення файлу рушія»

3. Ініціювати та налаштувати вікно програми. Наприклад:

```
sf::RenderWindow *window = new sf::RenderWindow(sf::VideoMode(1000, 600), "Test");
window->setKeyRepeatEnabled(false); // For proper keyboard events handling (e.g. jumping)
sf::View v = window->getDefaultView();
sf::View* debugScreenView = &v;
```

Рис.17 «Приклад коду ініціації вікна програми»

4. Ініціювати об'єкт Universe та, за потреби, дебаг менеджери:

```
sge::Universe* universe = new sge::Universe(window);
universe->setupDebugEntityManager();
```

```
universe->setupDebugScreenManager(debugScreenView, universe->assetsManager->getFont("m5x7"), 30);
```

Рис.18 «Приклади коду ініціації рушія»

5. Описати функціонал програми.

6. Запустити головний цикл рушія:

```
universe->loop();
```

Рис.18 «Код запуску циклу рушія»

2.6.4 Опис інтерфейсу користувача

Дана система є програмним ігровим рушієм, тому не надає інтерфейсу користувача у звичайному його розумінні. Система надає програмістам можливість створювати інтерфейси для потреб їх проєктів та ігор.

Система реалізовує абстракції DebugVariable та DebugScreenManager які дозволяють розробникам виводити релевантну дебаг інформацію у вікно програми.

Наприклад:

```
universe->debugScreenManager->addDebugVariable(
    sf::Vector2f(30, 60),
    new sge::DebugVariable{
        [playerEntity](){
            return
                "pos_x: " + std::to_string(playerEntity->sprite->getPosition().x) +
                "    pos_y: " + std::to_string(playerEntity->sprite->getPosition().y);
        }
    }
);
universe->debugScreenManager->addDebugVariable(
```

Рис.19 «Код створення змінної відображення позиції гравця»

Такі змінні можуть використовуватись для виводу інформації про наявні стани об’єкту, позицію, швидкість, прискорення, сили які діють на об’єкт тощо.

```
active_states:  on_ground                        [box] forces:  gravity -> (0.000000, 1000.000000)
pos_x: 377.804626    pos_y: 176.000000           [box] vel: (0.000000, 0.000000)
forces:  gravity -> (0.000000, 1000.000000)      [box] acc: (0.000000, 1000.000000)
vel: (0.000000, 0.000000)
acc: (0.000000, 1000.000000)
```

Рис.20 «Ігрове вікно з створеними розробником дебаг змінними»

Також приміром, використовуючи Entity з комбінацією компонентів Sprite та ClickableShape, розробники можуть створювати кнопки. Використовуючи Entity з комбінацією компонентів Sprite та SpriteText, розробники можуть виводити текст на екран.

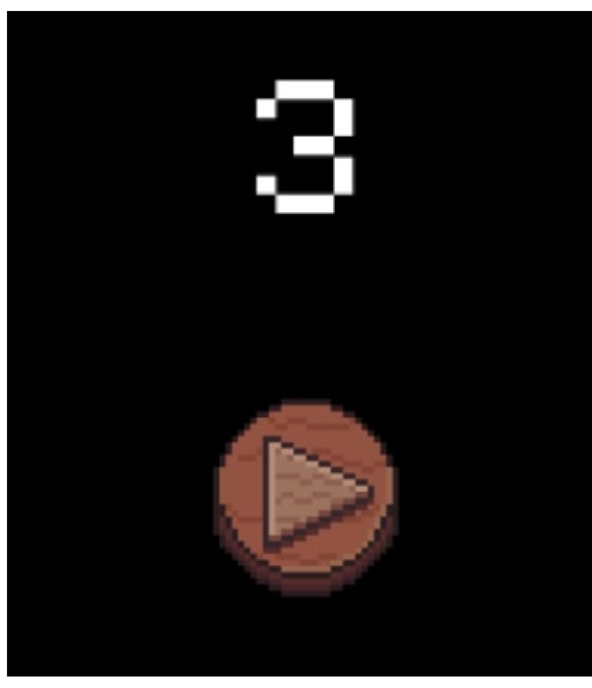

Рис.21 «Кнопка та текстовий об’єкт»

РОЗДІЛ 3
ЕКОНОМІЧНИЙ РОЗДІЛ

3.1 Розрахунок трудомісткості та вартості розробки інформаційної системи

Початкові дані:

1. передбачуване число операторів інформаційної системи — 2300
2. коефіцієнт корекції програми в ході її розробки — 0,1

3. коефіцієнт складності програми — 2

4. годинна заробітна плата програміста — 2000 грн/год [9]

5. коефіцієнт збільшення витрат праці в наслідок недостатнього опису задачі — 1,2

6. коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даною спеціальністю — 1,1

7. вартість машино-години ЕОМ — 20 грн/год

Нормування праці в процесі створення ПЗ істотно ускладнено в силу творчого характеру праці програміста. Тому трудомісткість розробки ПЗ може бути розрахована на основі системи моделей з різною точністю оцінки. Нормування праці в процесі створення ПЗ істотно ускладнено в силу творчого характеру праці програміста. Тому трудомісткість розробки ПЗ може бути розрахована на основі системи моделей з різною точністю оцінки.

Трудомісткість розробки інформаційної системи можна розрахувати за формулою:

$$t = t_o + t_и + t_а + t_п + t_{отл} + t_д, \text{ людино-годин}, \quad (3,1)$$

де t_o- витрати праці на підготовку й опис поставленої задачі (приймається 50 людино-годин);

$t_и$ - витрати праці на дослідження алгоритму рішення задачі;

$t_а$- витрати праці на розробку блок-схеми алгоритму;

$t_п$-витрати праці на програмування по готовій блок-схемі та монтажу приладу;

$t_{отл}$-витрати праці на налагодження інформаційної системи на ЕОМ;

$t_д$ - витрати праці на підготовку документації.

Складові витрати праці визначаються через умовне число операторів у інформаційної системи, яке розробляється.

Умовне число операторів (підпрограм):

$$Q = q \cdot C \cdot (1 + p),$$

де q - передбачуване число операторів (2300);

C - коефіцієнт складності інформаційної системи (2);

p - коефіцієнт корекції інформаційної системи в ході її розробки (0,1).

Звідси умовне число операторів інформаційної системи:

Q = 2300 * 2 * (1 + 0.1) = 5060

Витрати праці на вивчення опису задачі t_u визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_u = \frac{Q \cdot B}{(75..85) \cdot k}$$, людино-годин,

де B - коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі;

k - коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності. При стажі роботи від 1 до 2 роки він складає 1,1.

Приймемо збільшення витрат праці внаслідок недостатнього опису завдання не більше 50% (B = 1,2). З урахуванням коефіцієнта кваліфікації k = 1,1, отримуємо витрати праці на вивчення опису завдання:

t_u= (5060 * 1,2) / (75 * 1,1) = 73.6 людино - години.

Витрати праці на розробку алгоритму рішення задачі визначаються за формулою:

$$t_a = \frac{Q}{(20 \ldots 25) \cdot k}$$, людино-годин, (3.2)

де Q – умовне число операторів інформаційної системи;

k – коефіцієнт кваліфікації програміста.

Підставивши відповідні значення в формулу (3.2), отримаємо:

t_a= 5060 / (20 * 1,1) = 230 людино-годин.

Витрати на складання програми по готовій блок-схемі та монтажу інформаційної системи:

$t_{п} = \frac{Q}{(20...25)\cdot k}$, людино-годин.

t_n = 5060 / (25 * 1,1) = 184 людино-години.

Витрати праці на налагодження інформаційної системи на ЕОМ:

за умови автономного налагодження одного завдання:

$t_{отл} = \frac{Q}{(4..5)\cdot k}$, людино-годин.

$t_{отл}$= 5060 / (5 · 1,1) = 920 людино-годин.

за умови комплексного налагодження завдання:

$t^{k}_{отл} = 1,5 \cdot t_{отл}$, людино-годин.

$t^{k}_{отл}$= 1,5 · 920 = 1380 людино-годин.

Витрати праці на підготовку документації визначаються за формулою:

$t_{д} = t_{др} + t_{до}$, людино-годин,

де $t_{др}$-трудомісткість підготовки матеріалів і рукопису:

$$t_{др} = \frac{Q}{(15..20) \cdot k}$$, людино-годин,

$t_{до}$ - трудомісткість редагування, печатки й оформлення документації:

$$t_{до} = 0,75 \cdot t_{др}$$, людино-годин.

Підставляючи відповідні значення, отримаємо:

$t_{др}$ = 5060 / (20 · 1,1) = 230 людино-годин.

$t_{до}$ = 0,75 · 230 = 172,5 людино-годин.

$t_{д}$ = 230 + 172,5 = 402,55 людино-годин.

Повертаючись до формули (3.1), отримаємо повну оцінку трудомісткості розробки програмного забезпечення:

t = 50 + 23,6 + 230 + 184 + 1380 + 402,5 = 2296,6 людино-годин.

3.2 Розрахунок витрат на створення інфомаційної системи

Витрати на створення інформаційної системи $К_{ПО}$ включають витрати на заробітну плату виконавця програми $З_{ЗП}$ і витрат машинного часу, необхідного на налагодження інформаційної системи на ЕОМ:

$$К_{ПО} = З_{ЗП} + З_{МВ}$$, грн.

Заробітна плата виконавців визначається за формулою:

$З_{ЗП} = t \cdot С_{ПР}$, грн,

де: t - загальна трудомісткість, людино-годин;

$C_{ПР}$ - середня годинна заробітна плата програміста, грн/година

З урахуванням того, що середня годинна зарплата програміста становить 2000 грн / год, отримуємо:

$З_{ЗП} = 2296{,}6 * 2000 = 4539200$ грн

Вартість машинного часу, необхідного для налагодження інформаційної системи на ЕОМ, визначається за формулою:

$З_{мв} = t_{отл} \cdot C_{мч}$, грн, (3.3)

де $t_{отл}$ - трудомісткість налагодження інформаційної системи на ЕОМ, год;

$C_{мч}$ - вартість машино-години ЕОМ, грн/год (20 грн/год).

Підставивши в формулу (3.3) відповідні значення, визначимо вартість необхідного для налагодження машинного часу:

$З_{мв} = 920 * 20 = 18400$

Звідси витрати на створення інформаційної системи:

$K_{ПО} = 4539200 + 18400 = 4557600$ грн.

Очікуваний період створення інформаційної системи:

$T = \frac{t}{B_k \cdot F_p}$, м е міс.

де B_k- число виконавців (дорівнює 1);

F_p - місячний фонд робочого часу (при 40 годинному робочому тижні F_p=176 годин).

Звідси витрати на створення інформаційної системи:

$T = 2296,6 / 1 * 176 \approx 13$

Висновок: вартість розробки ігрового рушія становить 4557600 грн. Очікуваний час розробки становить 2296,6 годин, тобто 13 місяців. Цей термін пов'язаний зі числом операторів, і включає час на дослідження і розробку алгоритму вирішення поставленого завдання, програмування, налагодження, монтажу і підготовку документації.

ВИСНОВКИ

Результатом виконаної роботи є компактна система яка спрощує розробку ігрових застосунків та надає можливість створювати 2d ігри простого ступеня складності відносно швидко, порівняно з написанням всього коду застосунку самостійно.

Створена система надає можливість розробникам сфокусувати більшу кількість своїх сил та ресурсів на такі аспекти розробки як дизайн, написання цікавої історії, створення та покращення текстур, анімацій, аудіо, в той час як ігровий рушій спрощує створення програмної частини продукту.

З часу виходу оновлення стандарту C++ 11 [11] до мови була додана можливість створення lambda виразів [10], що, на мою думку значно спрощує та мотивує викорстання парадигм функціонального програмування. Концепт lambda виразів надав можливість створити досить цікаву та гнучку комбінацію парадигм об’єктно-орієнтованого програмування та функціонального програмування.

Також з часу стандарту C++ 11 до C++ були додані можливості асинхронного програмування[12], які, на мою думку спрощують роботу з потоками. Даний функціонал потенційно може бути використано для подальшої оптимізації даного ігрового рушія та саме ігор.

З подальшим розвитком мови програмування C++, технології SFML, та інструментів та концептів програмування в цілому, впевнений, що розробники матимуть ще більше можливостей виразити свою пристасть та креативність через створення ігор.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. C++ - мова програмування загального призначення. URL: https://uk.wikipedia.org/wiki/C%2B%2B

2. Simple and Fast Multimedia Library. Документація технології SFML. URL: https://www.sfml-dev.org/

3. Delta timing. URL: https://en.wikipedia.org/wiki/Delta_timing

4. Unified Modeling Language. URL: https://uk.wikipedia.org/wiki/Unified_Modeling_Language

5. Теорія множин. URL: https://en.wikipedia.org/wiki/Set_theory

6. Документація C++. URL: https://en.cppreference.com/w/

7. Стандарт C++ 20 URL: https://en.cppreference.com/w/cpp/20

8. Design patterns in C++ URL: https://refactoring.guru/design-patterns/cpp

9. How much do game developer jobs pay per hour? URL: https://www.ziprecruiter.com/Salaries/GAME-Developer-Salary-per-Hour

10. Lambda expressions URL: https://en.cppreference.com/w/cpp/language/lambda

11. Стандарт C++ 11 URL: https://en.cppreference.com/w/cpp/11

12. Async URL: https://en.cppreference.com/w/cpp/thread/async

ДОДАТОК А

ЛІСТИНГ ПРОГРАМИ

```
#ifndef SGE_HPP
#define SGE_HPP

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

#ifndef UNIVERSE_H
#define UNIVERSE_H

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

```
#include <vector>

namespace sge{
    class AssetsManager;
    class ControllerManager;
    class ScriptedViewManager;
    class DebugEntityManager;
    class DebugScreenManager;

    class SpriteManager;
    class MotionUnitManager;
    class CollisionShapeManager;
    class ClickableShapeManager;
    class SpriteTextManager;
    class AnimationManager;
    class StateManager;
    class CollisionManager;
    class EntityManager;
    class DrumSceneManager;
    class LayerSceneManager;


    class Universe{
        public:
            Universe(sf::RenderWindow* window);

            float dtCap = .15f;

            void setupDebugEntityManager();
            void setupDebugScreenManager(sf::View* debugScreenView, sf::Font* debugScreenFont, int fontSize);
            void loop();

            sge::AssetsManager* assetsManager = nullptr;
            sge::ControllerManager* controllerManager = nullptr;
            sge::ScriptedViewManager* scriptedViewManager = nullptr;
            sge::DebugEntityManager* debugEntityManager = nullptr;
            sge::DebugScreenManager* debugScreenManager = nullptr;

            sge::CollisionManager* collisionManager = nullptr;
            sge::EntityManager* entityManager = nullptr;
            sge::DrumSceneManager* drumSceneManager = nullptr;
            sge::LayerSceneManager* layerSceneManager = nullptr;
```

```
        private:
            sf::RenderWindow* m_windowPtr;

            sf::Clock m_deltaClock;

            sge::SpriteManager* m_spriteManager = nullptr;
            sge::MotionUnitManager* m_motionUnitManager = nullptr;
            sge::CollisionShapeManager* m_collisionShapeManager = nullptr;
            sge::AnimationManager* m_animationManager = nullptr;
            sge::StateManager* m_stateManager = nullptr;

            sge::ClickableShapeManager* m_clickableShapeManager = nullptr;
            sge::SpriteTextManager* m_spriteTextManager = nullptr;
    };
}

#endif

// Manager
#ifndef VECTOR_MANAGER_H
#define VECTOR_MANAGER_H

#include <algorithm>
#include <vector>

namespace sge{
    template<typename T>
    class VectorManager{
        public:
            void registerComponent(T component){ m_components.push_back(component); }
            void deregisterComponent(T component){ m_components.erase(std::remove(m_components.begin(),
m_components.end(), component), m_components.end()); }
            std::vector<T> getComponents(){ return m_components; };

        protected:
            std::vector<T> m_components;
    };
}

#endif
#ifndef VIEW_MANAGER_H
```

```
#define VIEW_MANAGER_H

#include <SFML/Graphics.hpp>
#include <algorithm>
#include <unordered_map>
#include <vector>

namespace sge{
    template<typename T>
    class ViewManager{
        public:
            void registerComponent(sf::View* view, T component){ m_components[view].push_back(component); }
            void deregisterComponent(sf::View* view, T
component){ m_components[view].erase(std::remove(m_components[view].begin(), m_components[view].end(),
component), m_components[view].end()); }
            std::vector<T> getViewComponents(sf::View* view){ return m_components[view]; }
            std::unordered_map<sf::View*, std::vector<T>> getComponentsUnorderedMap(){ return m_components; }

        protected:
            std::unordered_map<sf::View*, std::vector<T>> m_components;
    };
}

#endif
#ifndef LABEL_MANAGER_H
#define LABEL_MANAGER_H

#include <string>
#include <unordered_map>

namespace sge{
    template<typename T>
    class LabelManager{
        public:
            void registerComponent(std::string label, T component){ m_components[label] = component; }
            void deregisterComponent(std::string label){ m_components.erase(label); }
            T getComponent(std::string label){ return m_components[label]; }
            std::unordered_map<std::string, T> getComponentsMap(){ return m_components; }

        protected:
            std::unordered_map<std::string, T> m_components;
    };
```

```
}

#endif
//

// Component
#ifndef STATEFUL_COMPONENT_H
#define STATEFUL_COMPONENT_H

namespace sge{
    struct StatefulComponent{
        bool isActive = true;
        bool isPaused = false;
        bool isHidden = false;

        void activate(){
            isActive = true;
            isPaused = false;
            isHidden = false;
        }
        void pause(){
            isActive = false;
            isPaused = true;
            isHidden = false;
        }
        void hide(){
            isActive = false;
            isPaused = false;
            isHidden = true;
        }
    };
}

#endif
//

// AssetsManager
#ifndef ASSETS_MANAGER_H
#define ASSETS_MANAGER_H

#include <unordered_map>
#include <string>
```

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

#ifndef TEXTURE_SHEET_SIZES_H
#define TEXTURE_SHEET_SIZES_H

namespace sge{
    struct TextureSheetSizes{
        int textureSizeX;
        int textureSizeY;
        int numTexturesX;
        int numTexturesY;
        int gapX = 0;
        int gapY = 0;
    };
}

#endif

namespace sge{
    class TextureSheet;

    // TODO return nulls when asset does not exist
    class AssetsManager{
        public:
            void loadTextureSheet(std::string location, std::string name, sge::TextureSheetSizes textureSheetSizes);
            sge::TextureSheet* getTextureSheet(std::string name);

            void loadFont(std::string location, std::string name);
            sf::Font* getFont(std::string name);

            void loadSFX(std::string location, std::string name);
            sf::Sound* getSound(std::string name);

            void specifyMusicLocation(std::string location, std::string name);
            std::string getMusicLocation(std::string name);

        private:
            std::unordered_map<std::string, sge::TextureSheet*> m_textures;
            std::unordered_map<std::string, sf::Font*> m_fonts;
            std::unordered_map<std::string, std::pair<sf::SoundBuffer*, sf::Sound*>> m_sfx;
```

```
            std::unordered_map<std::string, std::string> m_musicLocations;
    };
}

#endif

#ifndef TEXTURESHEET_H
#define TEXTURESHEET_H

#include <SFML/Graphics.hpp>
#include <string>
#include <vector>

namespace sge{
    // TODO texturesheet with gaps between textures
    class TextureSheet{
        public:
            TextureSheet(sge::TextureSheetSizes textureSheetSizes, std::string location);

            std::string getLocation();
            sf::Texture* getTexture();
            sf::IntRect getTextureRect(int textureN, bool isFlippedHorizontally, bool isFlippedVertically);

        private:
            std::string m_location;
            sf::Texture m_textureSheet;
            std::vector<sf::IntRect> m_textureRects;
    };
}

#endif
//

// Debug
#ifndef DEBUG_ENTITY_H
#define DEBUG_ENTITY_H

#include <SFML/Graphics.hpp>
#include <unordered_map>
#include <vector>
#include <string>
#include <functional>
```

```
#ifndef COLLISION_SHAPE_BORDER_SETTINGS_H
#define COLLISION_SHAPE_BORDER_SETTINGS_H

#include <SFML/Graphics.hpp>

namespace sge{
    struct CollisionShapeBorderSettings{
        sf::Color color = sf::Color::Blue;
        float thickness = 1;
    };
}

#endif

namespace sge{
    class Entity;
    class CollisionShapeBorder;

    class DebugEntity : public sge::StatefulComponent{
        public:
            DebugEntity(sge::Entity* relatedEntity);

            sge::Entity* getRelatedEntity();

            bool drawCollisionShapeBorders = true;
            std::unordered_map<std::string, sge::CollisionShapeBorderSettings> customCollisionShapeBorderSettings;
            std::vector<sge::CollisionShapeBorder*> generateCollisionShapeBorders();

            void addExtraDebugFunction(std::function<void(sf::RenderWindow* windowPtr)> extraDebugFunction);
            std::vector<std::function<void(sf::RenderWindow* windowPtr)>> getExtraDebugFunctions();

        private:
            sge::Entity* m_relatedEntity;
            sge::CollisionShapeBorderSettings m_defaultCollisionShapeBorderSettings =
sge::CollisionShapeBorderSettings();
            std::vector<std::function<void(sf::RenderWindow* windowPtr)>> m_extraDebugFunctions;
    };
}

#endif
#ifndef DEBUG_ENTITY_MANAGER_H
```

```
#define DEBUG_ENTITY_MANAGER_H

#include <SFML/Graphics.hpp>
#include <unordered_map>
#include <vector>

#ifndef DRAW_MANAGER_H
#define DRAW_MANAGER_H

#include <SFML/Graphics.hpp>

namespace sge{
    struct DrawManager{
        virtual void draw(sf::RenderWindow* window){};
    };
}

#endif

namespace sge{
    class DebugEntity;

    class DebugEntityManager :
        public sge::ViewManager<sge::DebugEntity*>,
        public sge::DrawManager{

        public:
            void draw(sf::RenderWindow* window) override;
    };
}

#endif
#ifndef COLLISION_SHAPE_BORDER_H
#define COLLISION_SHAPE_BORDER_H

#include <SFML/Graphics.hpp>

namespace sge{
    class CollisionShape;
    struct CollisionShapeBorderSettings;

    class CollisionShapeBorder : public sf::RectangleShape{
```

```
        public:
            CollisionShapeBorder(sge::CollisionShape* owner, sge::CollisionShapeBorderSettings settings);
    };
}

#endif

#ifndef DEBUG_SCREEN_MANAGER_H
#define DEBUG_SCREEN_MANAGER_H

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

namespace sge{
    struct DebugVariable;
    class Entity;
    class EntityManager;

    class DebugScreenManager{
        public:
            DebugScreenManager(sge::EntityManager* entityManager, sf::View* debugScreenView, sf::Font*
debugScreenFont, int fontSize)
                : m_entityManagerPtr(entityManager), m_view(debugScreenView), m_font(debugScreenFont),
m_fontSize(fontSize){};

            void addDebugVariable(sf::Vector2f position, sge::DebugVariable* debugVariable);
            std::unordered_map<sge::DebugVariable*, sge::Entity*> getDebugVariables();

            void updateDebugVariables();

        private:
            std::unordered_map<sge::DebugVariable*, sge::Entity*> m_debugVariables;

            sf::View* m_view;
            sf::Font* m_font;
            int m_fontSize;
            sge::EntityManager* m_entityManagerPtr;
    };
}

#endif
```

```
#ifndef DEBUG_VARIABLE_H
#define DEBUG_VARIABLE_H

#include <functional>
#include <string>

namespace sge{
    class DebugVariable{
        public:
            std::function<std::string()> valueUpdateScript = [](){ return ""; };
            std::string value = "";

        friend class sge::DebugScreenManager;
        protected:
            void update(){ value = valueUpdateScript(); };
    };
}

#endif
//

// Controller
#ifndef CONTROLLER_H
#define CONTROLLER_H

#include <SFML/Graphics.hpp>

namespace sge{
    class Controller{
        public:
            virtual void script(sf::Event event){};
    };
}

#endif
#ifndef CONTROLLER_MANAGER_H
#define CONTROLLER_MANAGER_H

#include <functional>
#include <vector>

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

```
#ifndef EVENT_MANAGER_H
#define EVENT_MANAGER_H

#include <SFML/Graphics.hpp>

namespace sge{
    struct EventManager{
        virtual void processEvent(sf::Event event){};
    };
}

#endif

namespace sge{
    class Controller;

    class ControllerManager :
        public sge::VectorManager<sge::Controller*>,
        public sge::EventManager{

        public:
            void processEvent(sf::Event event) override;
    };
}

#endif
//

// View
#ifndef SCRIPTED_VIEW_H
#define SCRIPTED_VIEW_H

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

namespace sge{
    struct ScriptedView : public sf::View{
        virtual void script(){};
    };
}
```

```
#endif
#ifndef SCRIPTED_VIEW_MANAGER_H
#define SCRIPTED_VIEW_MANAGER_H

#include <SFML/Graphics.hpp>
#include <vector>
#include <string>

#ifndef UPDATE_MANAGER_H
#define UPDATE_MANAGER_H

namespace sge{
  struct UpdateManager{
    virtual void update(float dt){};
  };
}

#endif

namespace sge{
  struct ScriptedView;

  class ScriptedViewManager :
    public sge::VectorManager<sge::ScriptedView*>,
    public sge::UpdateManager{

    public:
      void update(float dt) override;
  };
}

#endif
//

// Logic
#ifndef SPRITE_MANAGER_H
#define SPRITE_MANAGER_H

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

```
namespace sge{
    class Sprite;

    class SpriteManager :
        public sge::ViewManager<sge::Sprite*>,
        public sge::DrawManager{

        public:
            void draw(sf::RenderWindow* window) override;
    };
}

#endif

#ifndef MOTION_UNIT_MANAGER_H
#define MOTION_UNIT_MANAGER_H

#include <vector>

namespace sge{
    class MotionUnit;

    class MotionUnitManager :
        public sge::VectorManager<sge::MotionUnit*>,
        public sge::UpdateManager{

        public:
            void update(float dt) override;
    };
}

#endif
#ifndef MOTION_UNIT_H
#define MOTION_UNIT_H

#include <SFML/Graphics.hpp>
#include <string>
#include <vector>
#include <unordered_map>
#include <functional>

namespace sge{
```

```
    class MotionUnit : public sge::StatefulComponent{
        public:
            MotionUnit(sf::Sprite* ownerSprite) : m_ownerSpritePtr(ownerSprite){};

            sf::Sprite* getOwnerSprite();

            sf::Vector2f velocity = sf::Vector2f(0, 0);
            sf::Vector2f acceleration = sf::Vector2f(0, 0);
            std::unordered_map<std::string, sf::Vector2f> extraProperties;

            std::unordered_map<std::string, sf::Vector2f> contactForces; // frictional forces, air resistance, tension, applied
forces, normal forces, forces in springs
            std::unordered_map<std::string, sf::Vector2f> fieldForces; // gravitational, electric, magnetic

            void addComputationScript(std::string name, std::function<void(sge::MotionUnit*, float)> computation);

            void update(float dt);

        private:
            sf::Sprite* m_ownerSpritePtr;

            std::unordered_map<std::string, std::function<void(sge::MotionUnit*, float)>> m_computationScripts;
            std::vector<std::string> m_computationScriptsOrder;
    };
}

#endif
#ifndef COMPUTATION_SCRIPTS_H
#define COMPUTATION_SCRIPTS_H

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

namespace sge{
    class MotionUnit;

    std::function<void(sge::MotionUnit*, float)> updatePositionBasedOnVelocity();

    std::function<void(sge::MotionUnit*, float)> updateVelocityBasedOnAcceleration(sf::Vector2f
speedLimit=sf::Vector2f(9999, 9999));

    std::function<void(sge::MotionUnit*, float)> calculateAcceleration();
```

```
}

#endif

#ifndef COLLISION_MANAGER_H
#define COLLISION_MANAGER_H

#include <string>
#include <vector>
#include <unordered_map>
#include <functional>

namespace sge{
   struct Collision;
   class CollisionShape;
   class CollisionInteraction;
   class CollisionShapeManager;

   class CollisionManager : public sge::VectorManager<CollisionInteraction*>{
      public:
         CollisionManager(CollisionShapeManager* collisionShapeManager):
m_collisionShapeManagerPtr(collisionShapeManager){};

         void updateCollisions();

      private:
         CollisionShapeManager* m_collisionShapeManagerPtr;
   };
}

#endif
#ifndef COLLISION_INTERACTION_H
#define COLLISION_INTERACTION_H

#include <unordered_map>
#include <vector>
#include <string>

namespace sge{
   struct Collision;
   class CollisionShape;
   class CollisionManager;
```

```
    class CollisionInteraction{
        friend class CollisionManager;

        public:
            CollisionInteraction(std::vector<std::string> initiatorGroups, std::vector<std::string> recipientGroups)
                : initiatorGroups(initiatorGroups), recipientGroups(recipientGroups){};

            std::vector<std::string> initiatorGroups = {};
            std::vector<std::string> recipientGroups = {};

            virtual bool collisionDetectionAlgorithm(sge::CollisionShape* initiator, sge::CollisionShape* recipient){ return
false; }

            virtual void startPhaseCollisionResponse(std::vector<sge::Collision> collisions){}
            virtual void continuousPhaseCollisionResponse(std::vector<sge::Collision> collisions){}
            virtual void endPhaseCollisionResponse(std::vector<sge::Collision> collisions){}

        protected:
            std::unordered_map<sge::CollisionShape*, std::vector<sge::Collision>> pastCollisions;
    };
}

#endif
#ifndef COLLISION_SIDE_H
#define COLLISION_SIDE_H

namespace sge{
    enum CollisionSide : int{ left, right, top, bottom };
}

#endif
#ifndef COLLISION_H
#define COLLISION_H

namespace sge{
    enum CollisionSide : int;
    class CollisionShape;

    struct Collision{
        sge::CollisionShape *initiator;
        sge::CollisionShape *recipient;
```

```
        sge::CollisionSide initiatorImpactSide;
        sge::CollisionSide recipientImpactSide;

        friend bool operator< (const sge::Collision a, const sge::Collision b){ return a.recipient < b.recipient; }
        friend bool operator> (const sge::Collision a, const sge::Collision b){ return a.recipient > b.recipient; }
        friend bool operator== (const sge::Collision a, const sge::Collision b){ return a.recipient == b.recipient; }
        friend bool operator!= (const sge::Collision a, const sge::Collision b){ return a.recipient != b.recipient; }
    };
}

#endif
#ifndef COLLISION_UTILS_H
#define COLLISION_UTILS_H

namespace sge{
    class CollisionShape;
    enum CollisionSide : int;

    float determineCollisionDepth(sge::CollisionSide initiatorImpactSide, sge::CollisionShape *initiator,
sge::CollisionShape *recipient);

    sge::CollisionSide determineInitiatorImpactSide(sge::CollisionShape *initiator, sge::CollisionShape *recipient);

    sge::CollisionSide flipInitiatorImpactSide(sge::CollisionSide initiatorImpactSide);
}

#endif
#ifndef COLLISION_SHAPE_H
#define COLLISION_SHAPE_H

#include <SFML/Graphics.hpp>
#include <vector>
#include <string>

namespace sge{
    class Entity;
    struct Measurements;

    class CollisionShape : public sge::StatefulComponent, public sf::RectangleShape{
        public:
            CollisionShape(sge::Entity* ownerEntity);
```

```
            sf::Vector2f offset = sf::Vector2f(0, 0);
            std::vector<std::string> collisionGroups;

            sge::Entity* getOwnerEntity();
            sge::Measurements getMeasurements();
            void align();

        private:
            sge::Entity* m_ownerEntityPtr;
    };
}

#endif
#ifndef COLLISION_SHAPE_MANAGER_H
#define COLLISION_SHAPE_MANAGER_H

#include <vector>

namespace sge{
    class CollisionShape;

    class CollisionShapeManager :
        public sge::VectorManager<sge::CollisionShape*>,
        public sge::UpdateManager{

        public:
            std::vector<sge::CollisionShape*> getComponentsByCollisionGroup(std::string groupName);

            void update(float dt) override;
    };
}

#endif
#ifndef COLLISION_RESPONSES_H
#define COLLISION_RESPONSES_H

#include <vector>

namespace sge{
    struct Collision;

    void resolveAABB(std::vector<sge::Collision> collisions);
```

```
    void initiatorStandOnTopOfRecipient(std::vector<sge::Collision> collisions);
}

#endif
#ifndef COLLISION_DETECTION_ALGORITHMS_H
#define COLLISION_DETECTION_ALGORITHMS_H

namespace sge{
    class CollisionShape;

    bool boundingBox(sge::CollisionShape* initiator, sge::CollisionShape* recipient);

    // TODO
    // bool rayRect(){}
}

#endif

#ifndef CLICKABLE_SHAPE_H
#define CLICKABLE_SHAPE_H

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

namespace sge{
    class Entity;

    class ClickableShape : public sge::StatefulComponent, public sf::RectangleShape{
        public:
            ClickableShape(sge::Entity* ownerEntity);

            std::function<void(sge::ClickableShape* thisClickableShape, sf::Event event)> action;

            sf::Vector2f offset = sf::Vector2f(0, 0);



            sge::Entity* getOwnerEntity();
            void align();

        private:
            sge::Entity* m_ownerEntityPtr;
```

```
    };
}

#endif
#ifndef CLICKABLE_SHAPE_MANAGER_H
#define CLICKABLE_SHAPE_MANAGER_H

#include <unordered_map>
#include <vector>
#include <SFML/Graphics.hpp>

namespace sge{
    struct ClickableShape;

    class ClickableShapeManager :
        public sge::ViewManager<ClickableShape*>,
        public sge::UpdateManager,
        public sge::EventManager{

        public:
            void update(float dt) override;
            void processEvent(sf::Event event) override;
    };
}

#endif
#ifndef CLICKABLE_SHAPE_HELPERS_H
#define CLICKABLE_SHAPE_HELPERS_H

#include <SFML/Graphics.hpp>

namespace sge{
    class ClickableShape;

    bool isMouseOverClickableShape(ClickableShape* clickableShape, sf::RenderWindow* window);
}

#endif

#ifndef SPRITE_TEXT_H
#define SPRITE_TEXT_H
```

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

namespace sge{
    class SpriteText : public sge::StatefulComponent, public sf::Text{
        public:
            SpriteText(sf::Sprite* ownerSprite);

            sf::Vector2f offset = sf::Vector2f(0, 0);

            void align();

        private:
            sf::Sprite* m_ownerSpritePtr;
    };
}

#endif
#ifndef SPRITE_TEXT_MANAGER_H
#define SPRITE_TEXT_MANAGER_H

#include <unordered_map>
#include <vector>
#include <SFML/Graphics.hpp>

namespace sge{
    class SpriteText;

    class SpriteTextManager :
        public sge::ViewManager<sge::SpriteText*>,
        public sge::UpdateManager,
        public sge::DrawManager{

        public:
            void update(float dt) override;
            void draw(sf::RenderWindow* window) override;
    };
}

#endif

#ifndef ANIMATION_MANAGER_H
#define ANIMATION_MANAGER_H
```

```
#include <vector>

namespace sge{
    class AnimationCluster;

    class AnimationManager :
        public sge::VectorManager<sge::AnimationCluster*>,
        public sge::UpdateManager{

        public:
            void update(float dt) override;
    };
}

#endif
#ifndef ANIMATION_CLUSTER_H
#define ANIMATION_CLUSTER_H

#include <SFML/Graphics.hpp>
#include <string>
#include <unordered_map>
#include <vector>

namespace sge{
    class TextureSequence;

    class AnimationCluster : public sge::StatefulComponent{
        public:
            AnimationCluster(sf::Sprite* ownerSprite) : m_ownerSpritePtr(ownerSprite){};

            int animationDelayMilliseconds = 100;

            void addTextureSequence(std::string name, TextureSequence* animationSequence);
            void setCurrentTextureSequence(std::string name);
            std::string getCurrentTextureSequence();

            void run();

        private:
            sf::Sprite* m_ownerSpritePtr;
```

```
            std::unordered_map<std::string, TextureSequence*> m_textureSequences;
            void m_updateTexture();

            sf::Clock m_clock;
            std::string m_currentTextureSequence;
            int m_currentTextureN = 0;
    };
}

#endif
#ifndef TEXTURE_SEQUENCE_H
#define TEXTURE_SEQUENCE_H

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

namespace sge{
    class TextureSheet;

    class TextureSequence{
        public:
            TextureSequence(std::vector<int> textureSequence, sge::TextureSheet* textureSheetPtr, bool
isFlippedHorizontally, bool isFlippedVertically);

            std::vector<sf::IntRect> sequenceRects;
            TextureSheet* textureSheet;
            // TODO create runForward / cycle flags and use them in 'Animation'
    };
}

#endif

#ifndef STATE_H
#define STATE_H

#include <functional>
#include <string>

namespace sge{
    class StateCluster;
    class Entity;
```

```
    class State{
        public:
            State(Entity* ownerEntity) : m_ownerEntityPtr(ownerEntity){};

            virtual void enterScript(){}
            virtual void exitScript(){}
            virtual void updateScript(float dt){}

        protected:
            Entity* m_ownerEntityPtr;
    };
}

#endif
#ifndef STATE_CLUSTER_H
#define STATE_CLUSTER_H

#include <unordered_map>
#include <functional>
#include <vector>
#include <string>

namespace sge{
    struct State;

    class StateCluster : public sge::StatefulComponent{
        public:
            std::vector<sge::State*> getActiveStates();
            std::vector<std::string> getActiveStateNames();
            void activateState(std::string name);
            void deactivateState(std::string name);

            bool isStateActive(std::string name);

            std::unordered_map<std::string, sge::State*> states;

        private:
            std::vector<std::string> m_activeStates;
    };
}

#endif
```

```
#ifndef STATE_MACHINE_H
#define STATE_MACHINE_H

#include <vector>

namespace sge{
    class StateCluster;

    class StateManager :
        public sge::VectorManager<sge::StateCluster*>,
        public sge::UpdateManager{

        public:
            void update(float dt) override;
    };
}

#endif

#ifndef ENTITY_H
#define ENTITY_H

#include <unordered_map>

namespace sge{
    class Sprite;
    class MotionUnit;
    class CollisionShape;
    class ClickableShape;
    class SpriteText;
    class AnimationCluster;
    class StateCluster;

    class Entity : public sge::StatefulComponent{
        public:
            sge::Sprite* sprite = nullptr;
            sge::MotionUnit* motionUnit = nullptr;
            std::unordered_map<std::string, sge::CollisionShape*> collisionShapes;
            sge::ClickableShape* clickableShape = nullptr;
            sge::SpriteText* spriteText = nullptr;
            sge::AnimationCluster* animationCluster = nullptr;
            sge::StateCluster* stateCluster = nullptr;
```

```
            void activateEntityParts();
            void pauseEntityParts();
            void hideEntityParts();

        private:
            // Hidden because extended using 'activateEntityParts', 'pauseEntityParts' and 'hideEntityParts'
            using sge::StatefulComponent::activate;
            using sge::StatefulComponent::pause;
            using sge::StatefulComponent::hide;
            //
    };
}

#endif
#ifndef ENTITY_MANAGER_H
#define ENTITY_MANAGER_H

#include <unordered_map>
#include <vector>
#include <SFML/Graphics.hpp>

namespace sge{
    class Entity;
    class SpriteManager;
    class MotionUnitManager;
    class CollisionShapeManager;
    class ClickableShapeManager;
    class SpriteTextManager;
    class AnimationManager;
    class CollisionManager;
    class StateManager;

    class EntityManager : public sge::ViewManager<sge::Entity*>{
        public:
            EntityManager(
                sge::SpriteManager* spriteManager,
                sge::MotionUnitManager* motionUnitManager,
                sge::CollisionShapeManager* collisionShapeManager,
                sge::ClickableShapeManager* clickableShapeManager,
                sge::SpriteTextManager* SpriteTextManager,
                sge::AnimationManager* animationManager,
```

```
            sge::StateManager* stateManager
        );

        void registerComponent(sf::View* view, sge::Entity* entity);
        void deregisterComponent(sf::View* view, sge::Entity* entity);

        private:
            // Hidden because extended
            using sge::ViewManager<sge::Entity*>::registerComponent;
            using sge::ViewManager<sge::Entity*>::deregisterComponent;
            //

            void m_registerEntityMembers(sf::View* view, sge::Entity* entity);
            void m_deregisterEntityMembers(sf::View* view, sge::Entity* entity);

            sge::SpriteManager* m_spriteManagerPtr;
            sge::MotionUnitManager* m_motionUnitManager;
            sge::CollisionShapeManager* m_collisionShapeManagerPtr;
            sge::ClickableShapeManager* m_clickableShapeManagerPtr;
            sge::SpriteTextManager* m_spriteTextManagerPtr;
            sge::AnimationManager* m_animationManagerPtr;
            sge::StateManager* m_stateManagerPtr;
    };
}

#endif
#ifndef PLAIN_ENTITY_H
#define PLAIN_ENTITY_H

#include <SFML/Graphics.hpp>

#ifndef SPRITE_H
#define SPRITE_H

#include <SFML/Graphics.hpp>

namespace sge{
    class Sprite : public sge::StatefulComponent, public sf::Sprite{
        public:
            Sprite() : sf::Sprite(){};
            Sprite(const sf::Texture& texture) : sf::Sprite(texture){};
            Sprite(const sf::Texture& texture, const sf::IntRect& rectangle) : sf::Sprite(texture, rectangle){};
```

```
    };
}

#endif

namespace sge{
    class PlainEntity : public sge::Entity{
        public:
            PlainEntity(sf::Texture* texture, sf::IntRect textureRect, sf::Vector2f position){
                sprite = new sge::Sprite(*texture, textureRect);
                sprite->setPosition(position);
            }
    };
}

#endif
#ifndef VOID_ENTITY_H
#define VOID_ENTITY_H

#include <SFML/Graphics.hpp>

namespace sge{
    class VoidEntity : public sge::Entity{
        public:
            VoidEntity(sf::Vector2f size, sf::Vector2f position, std::vector<std::string> collisionGroups){
                sprite = new sge::Sprite();
                sprite->setPosition(position);

                collisionShapes["global_bounds"] = new sge::CollisionShape(this);
                collisionShapes["global_bounds"]->setSize(size);
                collisionShapes["global_bounds"]->collisionGroups = collisionGroups;
            }
    };
}

#endif
#ifndef STATIC_ENTITY_H
#define STATIC_ENTITY_H

namespace sge{
    class StaticEntity : public sge::PlainEntity{
        public:
```

```
        StaticEntity(sf::Texture* texture, sf::IntRect textureRect, sf::Vector2f position, std::vector<std::string>
collisionGroups)
            : sge::PlainEntity(texture, textureRect, position){
                collisionShapes["global_bounds"] = new sge::CollisionShape(this);
                collisionShapes["global_bounds"]->collisionGroups = collisionGroups;
            }
    };
}

#endif
#ifndef MOBILE_ENTITY_H
#define MOBILE_ENTITY_H

#include <SFML/Graphics.hpp>

namespace sge{
    class MobileEntity : public sge::StaticEntity{
        public:
        MobileEntity(sf::Texture* texture, sf::IntRect textureRect, sf::Vector2f position, std::vector<std::string>
collisionGroups)
            : sge::StaticEntity(texture, textureRect, position, collisionGroups){
                motionUnit = new MotionUnit(sprite);
            }
    };
}

#endif
#ifndef SIMPLE_MOBILE_ENTITY_H
#define SIMPLE_MOBILE_ENTITY_H

#include <SFML/Graphics.hpp>

namespace sge{
    class SimpleMobileEntity : public sge::MobileEntity{
        public:
        SimpleMobileEntity(sf::Texture* texture, sf::IntRect textureRect, sf::Vector2f position, std::vector<std::string>
collisionGroups)
            : sge::MobileEntity(texture, textureRect, position, collisionGroups){
                motionUnit->addComputationScript("update_velocity", sge::updateVelocityBasedOnAcceleration());
                motionUnit->addComputationScript("update_position", sge::updatePositionBasedOnVelocity());
            }
    };
```

```
}

#endif
#ifndef COMPLEX_MOBILE_ENTITY_H
#define COMPLEX_MOBILE_ENTITY_H

#include <SFML/Graphics.hpp>

namespace sge{
    class ComplexMobileEntity : public sge::MobileEntity{
        public:
            ComplexMobileEntity(sf::Texture* texture, sf::IntRect textureRect, sf::Vector2f position,
std::vector<std::string> collisionGroups)
                : sge::MobileEntity(texture, textureRect, position, collisionGroups){
                    motionUnit->addComputationScript("update_acceleration", sge::calculateAcceleration());
                    motionUnit->addComputationScript("update_velocity", sge::updateVelocityBasedOnAcceleration());
                    motionUnit->addComputationScript("update_position", sge::updatePositionBasedOnVelocity());
                }
    };
}

#endif

#ifndef SCENE_H
#define SCENE_H

#include <unordered_map>
#include <vector>
#include <string>
#include <SFML/Graphics.hpp>

namespace sge{
    class Entity;
    class DebugEntity;
    class CollisionShape;

    class Scene : public sge::StatefulComponent{
        public:
            // rename to 'add' (parts are added)
            void addEntity(sf::View* view, sge::Entity* entity);
            void addEntities(sf::View* view, std::vector<sge::Entity*> entities);
            void addDebugEntity(sf::View* view, sge::DebugEntity* debugEntity);
```

```
            void addDebugEntities(sf::View* view, std::vector<sge::DebugEntity*> debugEntities);

            void activateSceneParts();
            void pauseSceneParts();
            void hideSceneParts();



            std::vector<sge::Entity*> getViewEntities(sf::View* view);
            std::unordered_map<sf::View*, std::vector<sge::Entity*>> getEntitiesMap();

            std::vector<sge::DebugEntity*> getViewDebugEntities(sf::View* view);
            std::unordered_map<sf::View*, std::vector<sge::DebugEntity*>> getDebugEntitiesMap();

        private:
            // Hidden because extended using 'activateScene', 'pauseScene' and 'hideScene'
            using sge::StatefulComponent::activate;
            using sge::StatefulComponent::pause;
            using sge::StatefulComponent::hide;
            //

            std::unordered_map<sf::View*, std::vector<sge::Entity*>> m_entities;
            std::unordered_map<sf::View*, std::vector<sge::DebugEntity*>> m_debugEntities;
    };
}

#endif
#ifndef SCENE_MANAGER_H
#define SCENE_MANAGER_H

#include <unordered_map>
#include <string>

namespace sge{
    class Scene;
    class EntityManager;
    class CollisionManager;
    class DebugEntityManager;

    class SceneManager : public sge::LabelManager<sge::Scene*>{
        public:
            SceneManager(sge::EntityManager* entityManager): m_entityManagerPtr(entityManager){};
            void setupDebug(sge::DebugEntityManager* debugEntityManager);
```

```
        protected:
            void m_registerSceneMembers(std::string label);
            void m_deregisterSceneMembers(std::string label);

            sge::EntityManager* m_entityManagerPtr = nullptr;
            sge::DebugEntityManager* m_debugEntityManagerPtr = nullptr;
    };
}

#endif
//

// utils
#ifndef APPROACH_H
#define APPROACH_H

namespace sge{
    float approach(float goal, float step, float current){
        float diff = goal - current;

        if(step > abs(goal - current)) return goal;
        if(diff > step) return current + step;
        if(diff < -step) return current - step;
        return goal;
    }
}

#endif
#ifndef MEASUREMENTS_H
#define MEASUREMENTS_H

namespace sge{
    struct Measurements{
        float x;
        float y;
        float height;
        float width;
    };
}

#endif
```

```
//

#endif

#ifndef SGE_MAIN
#define SGE_MAIN

#ifndef DRUM_SCENE_MANAGER_H
#define DRUM_SCENE_MANAGER_H

namespace sge{
    class Scene;

    class DrumSceneManager : public sge::SceneManager{
        public:
            DrumSceneManager(sge::EntityManager* entityManager)
                : sge::SceneManager(entityManager){};

            void setCurrentScene(std::string name);
            void alignScene();

        private:
            std::string m_currentScene = "";
            std::string m_loadedScene = "";
    };
}

#endif
#ifndef LAYER_SCENE_MANAGER_H
#define LAYER_SCENE_MANAGER_H

#include <string>

namespace sge{
    class Scene;

    class LayerSceneManager : public sge::SceneManager{
        public:
            LayerSceneManager(sge::EntityManager* entityManager)
                : sge::SceneManager(entityManager){};

            void registerComponent(std::string label, sge::Scene* scene);
```

```
        void deregisterComponent(std::string label);

      private:
        // Hidden because extended
        using sge::SceneManager::registerComponent;
        using sge::SceneManager::deregisterComponent;
        //
  };
}

#endif

sge::Universe::Universe(sf::RenderWindow* window){
  m_windowPtr = window;

  sge::AssetsManager* AsM = new sge::AssetsManager();
  sge::ControllerManager* CoM = new sge::ControllerManager();

  sge::ScriptedViewManager* VM = new sge::ScriptedViewManager();
  sge::SpriteManager* SpM = new sge::SpriteManager();
  sge::MotionUnitManager* MUM = new sge::MotionUnitManager();
  sge::CollisionShapeManager* CSM = new sge::CollisionShapeManager();
  sge::ClickableShapeManager* ClSM = new sge::ClickableShapeManager();
  sge::SpriteTextManager* STM = new SpriteTextManager();
  sge::AnimationManager* AnM = new sge::AnimationManager();
  sge::StateManager* StM = new sge::StateManager();
  sge::CollisionManager* CM = new sge::CollisionManager(CSM);
  sge::EntityManager* EM = new sge::EntityManager(SpM, MUM, CSM, ClSM, STM, AnM, StM);
  sge::DrumSceneManager* DrSM = new sge::DrumSceneManager(EM);
  sge::LayerSceneManager* LaSM = new sge::LayerSceneManager(EM);

  assetsManager = AsM;
  controllerManager = CoM;
  scriptedViewManager = VM;

  m_spriteManager = SpM;
  m_motionUnitManager = MUM;
  m_collisionShapeManager = CSM;
  m_clickableShapeManager = ClSM;
  m_spriteTextManager = STM;
  m_animationManager = AnM;
  m_stateManager = StM;
```

```
    collisionManager = CM;
    entityManager = EM;
    drumSceneManager = DrSM;
    layerSceneManager = LaSM;
}

void sge::Universe::setupDebugEntityManager(){
    sge::DebugEntityManager* debugEntityManagerPtr = new sge::DebugEntityManager();
    debugEntityManager = debugEntityManagerPtr;

    drumSceneManager->setupDebug(debugEntityManagerPtr);
    layerSceneManager->setupDebug(debugEntityManagerPtr);
}
void sge::Universe::setupDebugScreenManager(sf::View* debugScreenView, sf::Font* debugScreenFont, int fontSize = 
20){
    debugScreenManager = new sge::DebugScreenManager(entityManager, debugScreenView, debugScreenFont, 
fontSize);
}

void sge::Universe::loop(){
    m_deltaClock.restart();

    while(m_windowPtr->isOpen()){
        // dt
        sf::Time deltaTime = m_deltaClock.restart();
        float dt = deltaTime.asSeconds();
        if(dt > dtCap) dt = dtCap;
        //

        // Event
        sf::Event event;
        while(m_windowPtr->pollEvent(event)){
            if(event.type == sf::Event::Closed) m_windowPtr->close();

            controllerManager->processEvent(event);
            m_clickableShapeManager->processEvent(event);
        }
        //

        // Update
        m_motionUnitManager->update(dt);
        m_collisionShapeManager->update(dt);
```

```
        m_clickableShapeManager->update(dt);
        m_spriteTextManager->update(dt);
        m_stateManager->update(dt);
        collisionManager->updateCollisions();

        m_animationManager->update(dt);
        scriptedViewManager->update(dt);

        debugScreenManager->updateDebugVariables();

        drumSceneManager->alignScene(); // Scene can be reset only after all managers finished their updates to prevent
segfaults in loops
        //


        // Draw
        m_windowPtr->clear();

        m_spriteManager->draw(m_windowPtr);
        m_spriteTextManager->draw(m_windowPtr);
        if(debugEntityManager) debugEntityManager->draw(m_windowPtr);

        m_windowPtr->display();
        //
    }
}


void sge::AssetsManager::loadTextureSheet(std::string location, std::string name, sge::TextureSheetSizes
textureSheetSizes){ m_textures[name] = new sge::TextureSheet(textureSheetSizes, location); }
sge::TextureSheet* sge::AssetsManager::getTextureSheet(std::string name){ return m_textures[name]; }

void sge::AssetsManager::loadFont(std::string location, std::string name){
    sf::Font* font = new sf::Font;
    font->loadFromFile(location);

    m_fonts[name] = font;
}
sf::Font* sge::AssetsManager::getFont(std::string name){ return m_fonts[name]; }

void sge::AssetsManager::loadSFX(std::string location, std::string name){
    sf::SoundBuffer* buffer = new sf::SoundBuffer();
```

```
    buffer->loadFromFile(location);

    sf::Sound* sound = new sf::Sound(*buffer);
    m_sfx[name] = std::make_pair(buffer, sound);
}
sf::Sound* sge::AssetsManager::getSound(std::string name){ return m_sfx[name].second; }

void sge::AssetsManager::specifyMusicLocation(std::string location, std::string name){ m_musicLocations[name] =
location; }
std::string sge::AssetsManager::getMusicLocation(std::string name){ return m_musicLocations[name]; }


sge::TextureSheet::TextureSheet(sge::TextureSheetSizes textureSheetSizes, std::string location){
    m_location = location;
    m_textureSheet.loadFromFile(location);

    for(int i = 0; i < textureSheetSizes.numTexturesY*textureSheetSizes.textureSizeY; i +=
textureSheetSizes.textureSizeY+textureSheetSizes.gapY){
        for(int j = 0; j < textureSheetSizes.numTexturesX*textureSheetSizes.textureSizeX; j +=
textureSheetSizes.textureSizeX+textureSheetSizes.gapX){
            m_textureRects.push_back(sf::IntRect(j, i, textureSheetSizes.textureSizeX, textureSheetSizes.textureSizeY));
        }
    }
}

std::string sge::TextureSheet::getLocation(){ return m_location; }
sf::Texture* sge::TextureSheet::getTexture(){ return &m_textureSheet; }
sf::IntRect sge::TextureSheet::getTextureRect(int textureN, bool isFlippedHorizontally = false, bool isFlippedVertically
= false){
    sf::IntRect rect = m_textureRects[textureN];

    if(isFlippedHorizontally){
        rect.left += rect.width;
        rect.width *= -1;
    }
    if(isFlippedVertically) rect.height *= -1;

    return rect;
}


sge::DebugEntity::DebugEntity(sge::Entity* relatedEntity){ m_relatedEntity = relatedEntity; }
```

```
sge::Entity* sge::DebugEntity::getRelatedEntity(){ return m_relatedEntity; }

std::vector<sge::CollisionShapeBorder*> sge::DebugEntity::generateCollisionShapeBorders(){
    std::vector<sge::CollisionShapeBorder*> collisionShapeBorders;
    for(auto &[name, collisionShape] : m_relatedEntity->collisionShapes){
        if(customCollisionShapeBorderSettings.count(name)){
            collisionShapeBorders.push_back(new sge::CollisionShapeBorder(collisionShape,
customCollisionShapeBorderSettings[name]));
        }
        else{
            collisionShapeBorders.push_back(new sge::CollisionShapeBorder(collisionShape,
m_defaultCollisionShapeBorderSettings));
        }
    }

    return collisionShapeBorders;
}

void sge::DebugEntity::addExtraDebugFunction(std::function<void(sf::RenderWindow* windowPtr)>
extraDebugFunction){ m_extraDebugFunctions.push_back(extraDebugFunction); }
std::vector<std::function<void(sf::RenderWindow* windowPtr)>>
sge::DebugEntity::getExtraDebugFunctions(){ return m_extraDebugFunctions; }


void sge::DebugEntityManager::draw(sf::RenderWindow* window){
    for(auto& [view, debugEntities]: m_components){
        window->setView(*view);

        for(sge::DebugEntity* debugEntity : debugEntities){
            if(debugEntity->isActive || debugEntity->isPaused){
                // Run extraDebugFunctions
                // !
                // ! rewrite into update or just remove
                // !
                for(std::function<void(sf::RenderWindow* renderWindow)> extraDebugFunction :
debugEntity->getExtraDebugFunctions()){
                    extraDebugFunction(window);
                }
                //

                // Draw collision shape borders
```

```
            if(debugEntity->drawCollisionShapeBorders){
                for(sge::CollisionShapeBorder* collisionShapeBorder : debugEntity->generateCollisionShapeBorders()){
                    window->draw(*collisionShapeBorder);
                }
            }
            //
        }
    }
  }
}


sge::CollisionShapeBorder::CollisionShapeBorder(sge::CollisionShape* owner, sge::CollisionShapeBorderSettings
settings){
    this->setFillColor(sf::Color(0,0,0,0));

    this->setOutlineColor(settings.color);
    this->setOutlineThickness(settings.thickness);

    this->setPosition(sf::Vector2f(owner->getPosition().x + settings.thickness, owner->getPosition().y +
settings.thickness));
    this->setSize(sf::Vector2f(owner->getSize().x - settings.thickness*2, owner->getSize().y - settings.thickness*2));
}


void sge::DebugScreenManager::addDebugVariable(sf::Vector2f position, sge::DebugVariable* debugVariable){
    sge::Entity* textEntity = new sge::Entity();
    textEntity->sprite = new sge::Sprite();
    textEntity->sprite->setPosition(position);

    textEntity->spriteText = new sge::SpriteText(textEntity->sprite);
    textEntity->spriteText->setCharacterSize(m_fontSize);
    textEntity->spriteText->setFont(*m_font);
    textEntity->spriteText->setString(debugVariable->value);

    m_entityManagerPtr->registerComponent(m_view, textEntity);
    m_debugVariables[debugVariable] = textEntity;
}

std::unordered_map<sge::DebugVariable*, sge::Entity*> sge::DebugScreenManager::getDebugVariables(){ return
m_debugVariables; }
```

```
void sge::DebugScreenManager::updateDebugVariables(){
    for(auto& [debugVariable, textEntity] : m_debugVariables){
        debugVariable->update();
        textEntity->spriteText->setString(debugVariable->value);
    }
}


void sge::ControllerManager::processEvent(sf::Event event){
    for(sge::Controller* controller : m_components){
        controller->script(event);
    }
}


void sge::ScriptedViewManager::update(float dt){
    for(sge::ScriptedView* scriptedView : m_components){
        scriptedView->script();
    }
}


void sge::SpriteManager::draw(sf::RenderWindow* window){
    for(auto [view, sprites] : m_components){
        window->setView(*view);

        for(sge::Sprite* sprite : sprites){
            if(sprite->isActive || sprite->isPaused) window->draw(*sprite);
        }
    }
}


void sge::MotionUnitManager::update(float dt){
    for(sge::MotionUnit* motionUnit : m_components){
        if(motionUnit->isActive) motionUnit->update(dt);
    }
}


sf::Sprite* sge::MotionUnit::getOwnerSprite(){ return m_ownerSpritePtr; }
```

```
void sge::MotionUnit::addComputationScript(std::string name, std::function<void(sge::MotionUnit*, float)>
computation){
    m_computationScripts[name] = computation;
    m_computationScriptsOrder.push_back(name);
}

void sge::MotionUnit::update(float dt){
    for(std::string computation : m_computationScriptsOrder){
        m_computationScripts[computation](this, dt);
    }
};


// displacement = v * Δt
// new position = old position + displacement
std::function<void(sge::MotionUnit*, float)> sge::updatePositionBasedOnVelocity(){
    return [](sge::MotionUnit* thisMotionUnit, float dt){
        thisMotionUnit->getOwnerSprite()->setPosition(thisMotionUnit->getOwnerSprite()->getPosition() +
thisMotionUnit->velocity * dt);
    };
}

// Δv = a / Δt
std::function<void(sge::MotionUnit*, float)> sge::updateVelocityBasedOnAcceleration(sf::Vector2f speedLimit){
    return [speedLimit](sge::MotionUnit* thisMotionUnit, float dt){
        thisMotionUnit->velocity.x = sge::approach(speedLimit.x, thisMotionUnit->acceleration.x*dt,
thisMotionUnit->velocity.x);
        thisMotionUnit->velocity.y = sge::approach(speedLimit.y, thisMotionUnit->acceleration.y*dt,
thisMotionUnit->velocity.y);
    };
}

// a = Σ contactForces + Σ fieldForces
std::function<void(sge::MotionUnit*, float)> sge::calculateAcceleration(){
    return [](sge::MotionUnit* thisMotionUnit, float dt){
        sf::Vector2f netForce = sf::Vector2f(0, 0);
        for(auto& [_, force] : thisMotionUnit->contactForces){
            netForce += force;
        }
        for(auto& [_, force] : thisMotionUnit->fieldForces){
            netForce += force;
        }
```

```
        thisMotionUnit->acceleration = netForce;
    };
}
#include <algorithm>

void sge::CollisionManager::updateCollisions(){
    std::vector<sge::Collision> presentCollisions;

    for(CollisionInteraction* collisionInteraction : m_components){
        for(std::string initiatorGroup : collisionInteraction->initiatorGroups){
            for(sge::CollisionShape* initiator :
m_collisionShapeManagerPtr->getComponentsByCollisionGroup(initiatorGroup)){
                if(!initiator->isActive) continue;

                // Collect all present collisions
                for(std::string recipientGroup : collisionInteraction->recipientGroups){
                    for(sge::CollisionShape* recipient :
m_collisionShapeManagerPtr->getComponentsByCollisionGroup(recipientGroup)){
                        if(!recipient->isActive) continue;

                        if(collisionInteraction->collisionDetectionAlgorithm(initiator, recipient)){
                            CollisionSide initiatorImpactSide = determineInitiatorImpactSide(initiator, recipient);

                            presentCollisions.push_back(sge::Collision{
                                initiator,
                                recipient,
                                initiatorImpactSide,
                                flipInitiatorImpactSide(initiatorImpactSide)
                            });
                        }
                    }
                }
                //

                std::vector<sge::Collision> pastCollisions = collisionInteraction->pastCollisions[initiator];

                // Determine collision phase
                std::sort(presentCollisions.begin(), presentCollisions.end());
                std::sort(pastCollisions.begin(), pastCollisions.end());

                std::vector<sge::Collision> continuousPhaseCollisions;
```

```
            std::set_intersection(pastCollisions.begin(),pastCollisions.end(),
presentCollisions.begin(),presentCollisions.end(), std::back_inserter(continuousPhaseCollisions));

            std::vector<sge::Collision> startPhaseCollisions;
            std::set_difference(presentCollisions.begin(),presentCollisions.end(),
continuousPhaseCollisions.begin(),continuousPhaseCollisions.end(), std::back_inserter(startPhaseCollisions));

            std::vector<sge::Collision> endPhaseCollisions;
            std::set_difference(pastCollisions.begin(),pastCollisions.end(),
continuousPhaseCollisions.begin(),continuousPhaseCollisions.end(), std::back_inserter(endPhaseCollisions));
            //

            // Run collision responses based on collision phase
            if(startPhaseCollisions.size())
                collisionInteraction->startPhaseCollisionResponse(startPhaseCollisions);

            if(continuousPhaseCollisions.size())
                collisionInteraction->continuousPhaseCollisionResponse(continuousPhaseCollisions);

            if(endPhaseCollisions.size())
                collisionInteraction->endPhaseCollisionResponse(endPhaseCollisions);
            //

            // Reset
            collisionInteraction->pastCollisions[initiator] = presentCollisions;
            presentCollisions.clear();
            //
        }
    }
  }
}
#include <limits>

float sge::determineCollisionDepth(sge::CollisionSide initiatorImpactSide, sge::CollisionShape *initiator,
sge::CollisionShape *recipient){
    auto [x1, y1, height1, width1] = initiator->getMeasurements();
    auto [x2, y2, height2, width2] = recipient->getMeasurements();

    if(initiatorImpactSide == sge::CollisionSide::left) return x2 + width2 - x1;
    if(initiatorImpactSide == sge::CollisionSide::right) return x1 + width1 - x2;
    if(initiatorImpactSide == sge::CollisionSide::top) return y2 + height2 - y1;
    if(initiatorImpactSide == sge::CollisionSide::bottom) return y1 + height1 - y2;
```

```
    return 0;
}

sge::CollisionSide sge::determineInitiatorImpactSide(sge::CollisionShape *initiator, sge::CollisionShape *recipient){
    std::vector<sge::CollisionSide> allImpactSides;

    if(initiator->getPosition().x > recipient->getPosition().x) allImpactSides.push_back(sge::CollisionSide::left);
    if(initiator->getPosition().x < recipient->getPosition().x) allImpactSides.push_back(sge::CollisionSide::right);
    if(initiator->getPosition().y > recipient->getPosition().y) allImpactSides.push_back(sge::CollisionSide::top);
    if(initiator->getPosition().y < recipient->getPosition().y) allImpactSides.push_back(sge::CollisionSide::bottom);

    sge::CollisionSide lowestDepthSide;
    float lowestDepth = std::numeric_limits<float>::infinity();

    for(sge::CollisionSide collisionSide : allImpactSides){
        float depth = sge::determineCollisionDepth(collisionSide, initiator, recipient);
        if(depth <= lowestDepth){
            lowestDepthSide = collisionSide;
            lowestDepth = depth;
        }
    }

    return lowestDepthSide;
}

sge::CollisionSide sge::flipInitiatorImpactSide(sge::CollisionSide initiatorImpactSide){
    if(initiatorImpactSide == sge::CollisionSide::top) return sge::CollisionSide::bottom;
    if(initiatorImpactSide == sge::CollisionSide::bottom) return sge::CollisionSide::top;
    if(initiatorImpactSide == sge::CollisionSide::right) return sge::CollisionSide::left;
    if(initiatorImpactSide == sge::CollisionSide::left) return sge::CollisionSide::right;

    return sge::CollisionSide::bottom;
}


sge::CollisionShape::CollisionShape(sge::Entity* ownerEntity){
    m_ownerEntityPtr = ownerEntity;

    this->setFillColor(sf::Color(0,0,0,0));
    this->setSize(sf::Vector2f(ownerEntity->sprite->getGlobalBounds().width,
ownerEntity->sprite->getGlobalBounds().height));
```

```
}

sge::Entity* sge::CollisionShape::getOwnerEntity(){ return m_ownerEntityPtr; }
sge::Measurements sge::CollisionShape::getMeasurements(){
   return { this->getPosition().x, this->getPosition().y, this->getGlobalBounds().height,
this->getGlobalBounds().width };
}
void sge::CollisionShape::align(){
   this->setPosition(m_ownerEntityPtr->sprite->getPosition() + offset);
}
#include <algorithm>

std::vector<sge::CollisionShape*> sge::CollisionShapeManager::getComponentsByCollisionGroup(std::string
groupName){
   std::vector<sge::CollisionShape*> collisionGroupMembers;
   for(sge::CollisionShape* collisionShape : m_components){
      // if CollisionShape is a member of 'groupName' collision group
      if(std::find(collisionShape->collisionGroups.begin(), collisionShape->collisionGroups.end(), groupName) !=
collisionShape->collisionGroups.end()){
         collisionGroupMembers.push_back(collisionShape);
      }
   }

   return collisionGroupMembers;
}

void sge::CollisionShapeManager::update(float dt){
   for(sge::CollisionShape* collisionShape : m_components){
      if(collisionShape->isActive) collisionShape->align();
   }
}



void sge::resolveAABB(std::vector<sge::Collision> collisions){
   for(sge::Collision collision : collisions){
      sge::CollisionShape* initiatorCollisionShape = collision.initiator;
      sge::CollisionShape* recipientCollisionShape = collision.recipient;
      sge::Sprite *initiatorSprite = collision.initiator->getOwnerEntity()->sprite;
      sge::Sprite *recipientSprite = collision.recipient->getOwnerEntity()->sprite;

      // Align initiator based on impact side
      if(collision.initiatorImpactSide == sge::CollisionSide::left){
```

```
            initiatorSprite->setPosition(
                recipientSprite->getPosition().x + recipientSprite->getGlobalBounds().width - collision.initiator->offset.x +
collision.recipient->offset.x,
                initiatorSprite->getPosition().y
            );
        }
        else if(collision.initiatorImpactSide == sge::CollisionSide::right){
            initiatorSprite->setPosition(
                recipientSprite->getPosition().x - collision.initiator->getGlobalBounds().width - collision.initiator->offset.x +
collision.recipient->offset.x,
                initiatorSprite->getPosition().y
            );
        }
        else if(collision.initiatorImpactSide == sge::CollisionSide::top){
            initiatorSprite->setPosition(
                initiatorSprite->getPosition().x,
                recipientSprite->getPosition().y + recipientSprite->getGlobalBounds().height - collision.initiator->offset.y +
collision.recipient->offset.y
            );
        }
        else if(collision.initiatorImpactSide == sge::CollisionSide::bottom){
            initiatorSprite->setPosition(
                initiatorSprite->getPosition().x,
                recipientSprite->getPosition().y - collision.initiator->getGlobalBounds().height - collision.initiator->offset.y +
collision.recipient->offset.y
            );
        }
        //
    }
}

void sge::initiatorStandOnTopOfRecipient(std::vector<sge::Collision> collisions){
    for(sge::Collision collision : collisions){
        if(collision.initiatorImpactSide == sge::CollisionSide::bottom){
            collision.initiator->getOwnerEntity()->motionUnit->velocity.y = 0;
        }
    }
}


bool sge::boundingBox(sge::CollisionShape* initiator, sge::CollisionShape* recipient){
    return initiator->getGlobalBounds().intersects(recipient->getGlobalBounds());
```

```
}


sge::ClickableShape::ClickableShape(sge::Entity* ownerUIEntity){ m_ownerEntityPtr = ownerUIEntity; }

sge::Entity* sge::ClickableShape::getOwnerEntity(){ return m_ownerEntityPtr; }
void sge::ClickableShape::align(){ this->setPosition(m_ownerEntityPtr->sprite->getPosition() + offset); }
#include <algorithm>
#include <SFML/Graphics.hpp>

void sge::ClickableShapeManager::update(float dt){
   for(auto& [_, clickableShapes] : m_components){
      for(sge::ClickableShape* clickableShape : clickableShapes){
         if(clickableShape->isActive) clickableShape->align();
      }
   }
}

void sge::ClickableShapeManager::processEvent(sf::Event event){
   for(auto& [_, clickableShapes] : m_components){
      for(ClickableShape* clickableShape : clickableShapes){
         if(clickableShape->isActive) clickableShape->action(clickableShape, event);
      }
   }
}


bool sge::isMouseOverClickableShape(ClickableShape* clickableShape, sf::RenderWindow* window){
   return
clickableShape->getOwnerEntity()->sprite->getGlobalBounds().contains(window->mapPixelToCoords(sf::Mouse::getP
osition(*window)));
}


sge::SpriteText::SpriteText(sf::Sprite* ownerSprite){ m_ownerSpritePtr = ownerSprite; }
void sge::SpriteText::align(){
   sf::Vector2f pos = m_ownerSpritePtr->getPosition() + offset;

   this->setPosition((int)pos.x, (int)pos.y);
}
#include <algorithm>
```

```
void sge::SpriteTextManager::update(float dt){
    for(auto& [_, spriteTextObjects] : m_components){
        for(sge::SpriteText* spriteText : spriteTextObjects){
            if(spriteText->isActive) spriteText->align();
        }
    }
}
void sge::SpriteTextManager::draw(sf::RenderWindow* window){
    for(auto& [view, spriteTextObjects] : m_components){
        window->setView(*view);

        for(SpriteText* spriteText : spriteTextObjects){
            if(spriteText->isActive || spriteText->isPaused) window->draw(*spriteText);
        }
    }
}


void sge::AnimationManager::update(float dt){
    for(sge::AnimationCluster* animation : m_components){
        if(animation->isActive) animation->run();
    }
}


void sge::AnimationCluster::addTextureSequence(std::string name, TextureSequence*
textureSequence){ m_textureSequences[name] = textureSequence; }
void sge::AnimationCluster::setCurrentTextureSequence(std::string name){
    m_clock.restart();
    m_currentTextureSequence = name;
    m_currentTextureN = 0;

    m_updateTexture();
}
std::string sge::AnimationCluster::getCurrentTextureSequence(){ return m_currentTextureSequence; }

void sge::AnimationCluster::run(){
    if(!m_textureSequences.size()){
        printf("No texture sequences initialized.\n");
        exit(1);
    }
    if(!m_currentTextureSequence.length()){
```

```
        printf("Can not run AnimationCluster if no current texture sequence is set.\n"); // ? Default to first added ?
        exit(1);
    }

    if(m_clock.getElapsedTime().asMilliseconds() > animationDelayMilliseconds){
        m_updateTexture();

        if(m_currentTextureN+1 == m_textureSequences[m_currentTextureSequence]->sequenceRects.size()){
            m_currentTextureN = 0;
        }
        else m_currentTextureN++;

        m_clock.restart();
    }
}

void sge::AnimationCluster::m_updateTexture(){
    m_ownerSpritePtr->setTexture(*m_textureSequences[m_currentTextureSequence]->textureSheet->getTexture());

m_ownerSpritePtr->setTextureRect(m_textureSequences[m_currentTextureSequence]->sequenceRects[m_currentTextu
reN]);
}


sge::TextureSequence::TextureSequence(std::vector<int> textureSequence, sge::TextureSheet* textureSheetPtr, bool
isFlippedHorizontally = false, bool isFlippedVertically = false){
    textureSheet = textureSheetPtr;

    for(int n : textureSequence){
        sf::IntRect rect = textureSheet->getTextureRect(n, isFlippedHorizontally, isFlippedVertically);
        sequenceRects.push_back(rect);
    }
}
#include <algorithm>

std::vector<sge::State*> sge::StateCluster::getActiveStates(){
    std::vector<sge::State*> activeStates;

    for(std::string activeStateName : m_activeStates){
        activeStates.push_back(states[activeStateName]);
    }
```

```
    return activeStates;
}
std::vector<std::string> sge::StateCluster::getActiveStateNames(){ return m_activeStates; }

void sge::StateCluster::activateState(std::string name){
    if(!isStateActive(name)){
        states[name]->enterScript();
        m_activeStates.push_back(name);
    }
}
void sge::StateCluster::deactivateState(std::string name){
    if(isStateActive(name)){
        states[name]->exitScript();
        m_activeStates.erase(std::remove(m_activeStates.begin(), m_activeStates.end(), name), m_activeStates.end());
    }
}

bool sge::StateCluster::isStateActive(std::string name){
    return std::count(m_activeStates.begin(), m_activeStates.end(), name);
}
#include <algorithm>

void sge::StateManager::update(float dt){
    for(StateCluster* stateCluster : m_components){
        if(stateCluster->isActive){
            for(sge::State* state : stateCluster->getActiveStates()){
                state->updateScript(dt);
            }
        }
    }
}



void sge::Entity::activateEntityParts(){
    sprite->activate();
    if(motionUnit) motionUnit->activate();
    if(!collisionShapes.empty()){
        for(auto& [_, collisionShape] : collisionShapes) collisionShape->activate();
    }
    if(clickableShape) clickableShape->activate();
    if(spriteText) spriteText->activate();
    if(animationCluster) animationCluster->activate();
```

```
    if(stateCluster) stateCluster->activate();

    sge::StatefulComponent::activate();
}

void sge::Entity::pauseEntityParts(){
    sprite->pause();
    if(motionUnit) motionUnit->pause();
    if(!collisionShapes.empty()){
        for(auto& [_, collisionShape] : collisionShapes) collisionShape->pause();
    }
    if(clickableShape) clickableShape->pause();
    if(spriteText) spriteText->pause();
    if(animationCluster) animationCluster->pause();
    if(stateCluster) stateCluster->pause();

    sge::StatefulComponent::pause();
}

void sge::Entity::hideEntityParts(){
    sprite->hide();
    if(motionUnit) motionUnit->hide();
    if(!collisionShapes.empty()){
        for(auto& [_, collisionShape] : collisionShapes) collisionShape->hide();
    }
    if(clickableShape) clickableShape->hide();
    if(spriteText) spriteText->hide();
    if(animationCluster) animationCluster->hide();
    if(stateCluster) stateCluster->hide();

    sge::StatefulComponent::hide();
}


sge::EntityManager::EntityManager(
        sge::SpriteManager* spriteManager,
        sge::MotionUnitManager* motionUnitManager,
        sge::CollisionShapeManager* collisionShapeManager,
        sge::ClickableShapeManager* clickableShapeManager,
        sge::SpriteTextManager* spriteTextManager,
        sge::AnimationManager* animationManager,
        sge::StateManager* stateManager
```

```
    ){

    m_spriteManagerPtr = spriteManager;
    m_motionUnitManager = motionUnitManager;
    m_collisionShapeManagerPtr = collisionShapeManager;
    m_clickableShapeManagerPtr = clickableShapeManager;
    m_spriteTextManagerPtr = spriteTextManager;
    m_animationManagerPtr = animationManager;
    m_stateManagerPtr = stateManager;
}

void sge::EntityManager::registerComponent(sf::View* view, sge::Entity* entity){
    m_registerEntityMembers(view, entity);
    sge::ViewManager<sge::Entity*>::registerComponent(view, entity);
}
void sge::EntityManager::deregisterComponent(sf::View* view, sge::Entity* entity){
    m_deregisterEntityMembers(view, entity);
    sge::ViewManager<sge::Entity*>::deregisterComponent(view, entity);
}

void sge::EntityManager::m_registerEntityMembers(sf::View* view, sge::Entity* entity){
    m_spriteManagerPtr->registerComponent(view, entity->sprite);

    if(entity->motionUnit){
        m_motionUnitManager->registerComponent(entity->motionUnit);
    }

    if(entity->collisionShapes.size()){
        for(auto& [_, collisionShape] : entity->collisionShapes){
            m_collisionShapeManagerPtr->registerComponent(collisionShape);
        }
    }

    if(entity->clickableShape){
        m_clickableShapeManagerPtr->registerComponent(view, entity->clickableShape);
    }

    if(entity->spriteText){
        m_spriteTextManagerPtr->registerComponent(view, entity->spriteText);
    }

    if(entity->animationCluster){
```

```
        m_animationManagerPtr->registerComponent(entity->animationCluster);
    }

    if(entity->stateCluster){
        m_stateManagerPtr->registerComponent(entity->stateCluster);
    }
}

void sge::EntityManager::m_deregisterEntityMembers(sf::View* view, sge::Entity* entity){
    m_spriteManagerPtr->deregisterComponent(view, entity->sprite);

    if(entity->motionUnit){
        m_motionUnitManager->deregisterComponent(entity->motionUnit);
    }

    if(entity->collisionShapes.size()){
        for(auto[_, collisionShape] : entity->collisionShapes){
            m_collisionShapeManagerPtr->deregisterComponent(collisionShape);
        }
    }

    if(entity->clickableShape){
        m_clickableShapeManagerPtr->deregisterComponent(view, entity->clickableShape);
    }

    if(entity->spriteText){
        m_spriteTextManagerPtr->deregisterComponent(view, entity->spriteText);
    }

    if(entity->animationCluster){
        m_animationManagerPtr->deregisterComponent(entity->animationCluster);
    }

    if(entity->stateCluster){
        m_stateManagerPtr->deregisterComponent(entity->stateCluster);
    }
}


void sge::Scene::addEntity(sf::View* view, sge::Entity* entity){ m_entities[view].push_back(entity); }
void sge::Scene::addEntities(sf::View* view, std::vector<sge::Entity*> entities){
    for(Entity* entity : entities){
```

```
        addEntity(view, entity);
    }
}
void sge::Scene::addDebugEntity(sf::View* view, sge::DebugEntity*
debugEntity){ m_debugEntities[view].push_back(debugEntity); }
void sge::Scene::addDebugEntities(sf::View* view,std::vector<DebugEntity*> debugEntities){
    for(DebugEntity* debugEntity : debugEntities){
        addDebugEntity(view, debugEntity);
    }
}

void sge::Scene::activateSceneParts(){
    for(auto& [_, entities] : m_entities){
        for(sge::Entity* entity : entities){
            entity->activateEntityParts();
        }
    }
    for(auto& [_, debugEntities] : m_debugEntities){
        for(sge::DebugEntity* debugEntity : debugEntities){
            debugEntity->activate();
        }
    }

    sge::StatefulComponent::activate();
}
void sge::Scene::pauseSceneParts(){
    for(auto& [_, entities] : m_entities){
        for(sge::Entity* entity : entities){
            entity->pauseEntityParts();
        }
    }
    for(auto& [_, debugEntities] : m_debugEntities){
        for(sge::DebugEntity* debugEntity : debugEntities){
            debugEntity->pause();
        }
    }

    sge::StatefulComponent::pause();
}
void sge::Scene::hideSceneParts(){
    for(auto& [_, entities] : m_entities){
        for(sge::Entity* entity : entities){
```

```
            entity->hideEntityParts();
        }
    }
    for(auto& [_, debugEntities] : m_debugEntities){
        for(sge::DebugEntity* debugEntity : debugEntities){
            debugEntity->hide();
        }
    }

    sge::StatefulComponent::hide();
}

std::vector<sge::Entity*> sge::Scene::getViewEntities(sf::View* view){ return m_entities[view]; }
std::unordered_map<sf::View*, std::vector<sge::Entity*>> sge::Scene::getEntitiesMap(){ return m_entities; };

std::vector<sge::DebugEntity*> sge::Scene::getViewDebugEntities(sf::View* view){ return m_debugEntities[view]; }
std::unordered_map<sf::View*, std::vector<sge::DebugEntity*>> sge::Scene::getDebugEntitiesMap(){ return
m_debugEntities; }


void sge::SceneManager::setupDebug(sge::DebugEntityManager* debugEntityManager){ m_debugEntityManagerPtr =
debugEntityManager; }

void sge::SceneManager::m_registerSceneMembers(std::string label){
    for(auto& [view, entities] : m_components[label]->getEntitiesMap()){
        for(sge::Entity* entity : entities){
            m_entityManagerPtr->registerComponent(view, entity);
        }
    }

    if(m_debugEntityManagerPtr){
        for(auto& [view, debugEntities] : m_components[label]->getDebugEntitiesMap()){
            for(sge::DebugEntity* debugEntity : debugEntities){
                m_debugEntityManagerPtr->registerComponent(view, debugEntity);
            }
        }
    }
}

void sge::SceneManager::m_deregisterSceneMembers(std::string label){
    if(m_debugEntityManagerPtr){
        for(auto& [view, debugEntities] : m_components[label]->getDebugEntitiesMap()){
```

```
            for(sge::DebugEntity* debugEntity : debugEntities){
                m_debugEntityManagerPtr->deregisterComponent(view, debugEntity);
            }
        }
    }

    for(auto& [view, entities] : m_components[label]->getEntitiesMap()){
        for(sge::Entity* entity : entities){
            m_entityManagerPtr->deregisterComponent(view, entity);
        }
    }
}


void sge::DrumSceneManager::setCurrentScene(std::string name){ m_currentScene = name; }

void sge::DrumSceneManager::alignScene(){
    if(m_currentScene.length()){
        if(m_loadedScene != m_currentScene){
            if(m_loadedScene.length()) m_deregisterSceneMembers(m_loadedScene);

            m_registerSceneMembers(m_currentScene);

            m_loadedScene = m_currentScene;
        }
    }
}


void sge::LayerSceneManager::registerComponent(std::string label, sge::Scene* scene){
    sge::SceneManager::registerComponent(label, scene);
    m_registerSceneMembers(label);
}
void sge::LayerSceneManager::deregisterComponent(std::string label){
    m_deregisterSceneMembers(label);
    sge::SceneManager::deregisterComponent(label);
}
#endif
```