МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
КАФЕДРА «ПРИКЛАДНА МАТЕМАТИКА ТА ІНФОРМАТИКА»

НАВЧАЛЬНИЙ МОДУЛЬ

«Розробка кросплатформених ігрових додатків на HTML5» дисципліни «Інструментальна підтримка розробки комп'ютерних ігрових додатків» підготовки студентів освітнього рівня «магістр» за спеціалізацією «Програмне забезпечення мультимедійних систем для ігрових додатків» спеціальності 121 «Інженерія програмного забезпечення»

GAMEHUB: UNIVERSITY-ENTERPRISES COOPERATION IN GAME INDUSTRY IN UKRAINE"

GAMEHUB: СПІВРОБІТНИЦТВО МІЖ УНІВЕРСИТЕТАМИ ТА ПІДПРИЄМСТВАМИ В СФЕРІ ГРАЛЬНОЇ ІНДУСТРІЇ В УКРАЇНІ»

561728-EPP-1-2015-1- ES-EPPKA2-CBHE-JP

The handbooks were performed with support of the Erasmus+ Programme of the European Union (561728-EPP-1-2015-1-ES-EPPKA2-CBHE-JP). The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Покровськ, 2018

УДК 004.032.6 : 004.92

Навчальний модуль «Розробка кросплатформених ігрових додатків на HTML5» дисципліни «Інструментальна підтримка розробки комп'ютерних ігрових додатків» підготовки магістрів за спеціалізацією «Програмне забезпечення мультимедійних систем для ігрових додатків» спеціальності 121 «Інженерія програмного забезпечення» / Укладачі: С.О. Цололо, Ю.Л. Дікова. – Покровськ: ДонНТУ, 2018. – 108 с.

Навчальний модуль «Розробка кросплатформених ігрових додатків на HTML5» дисципліни «Інструментальна підтримка розробки комп'ютерних ігрових додатків» входять до дисциплін підготовки студентів освітнього ступеня «магістр» спеціальності 121 «Програмна інженерія» за спеціалізацією «Програмне забезпечення мультимедійних систем для ігрових додатків», яка впроваджена в рамках виконання гранту Еразмус+ 561728-EPP-1-2015-1- ES-EPPKA2-CBHE-JP «GameHub: Співробітництво між університетами та підприємствами в сфері грального індустрії в Україні»

Укладачі:
С.О. Цололо, доцент кафедри КІ, к.т.н., доцент;
Ю.Л. Дікова, старший викладач кафедри КІ.

Рецензенти:
Святний В.А., завідувач кафедри КІ, д.т.н., професор;
Вовна О.В., завідувач кафедри ЕТ, д.т.н., доцент.

Відповідальний за випуск: **Дмитрієва О.А.**, завідувач кафедри ПМІ.

Розглянуто на засіданні кафедри «Прикладної математики і інформатики»
протокол № 11 від 15 травня 2018 року

Затверджено навчально-видавничим відділом ДонНТУ,
протокол № 14 від 12 червня 2018 року

Рекомендовано до друку Вченою радою ДонНТУ,
протокол № 10 від 21 червня 2018 року

ЗМІСТ

ЗАГАЛЬНИЙ ВСТУП

В рамках виконання гранту Еразмус+ 561728-EPP-1-2015-1- ES-EPPKA2-CBHE-JP «GameHub: Співробітництво між університетами та підприємствами в сфері гральної індустрії в Україні» в Донецькому національному технічному університеті впроваджена підготовка студентів спеціальності 121 «Програмна інженерія» за спеціалізацією «Програмне забезпечення мультимедійних систем для ігрових додатків».

Навчальний план підготовки студентів на рівні «магістр» передбачає опрацювання наступних навчальних модулів:

- «Методи теорії ігор в ігрових додатках» (Theory of Games in Game Applications), дисципліна «Математична теорія ігор»;
- «Розробка ігрових додатків на базі движка Unity» (Game Applications Development Based on Unity Engine), дисципліна «Інструментальна підтримка розробки комп'ютерних ігрових додатків»;
- «Розробка ігрових додатків для OS Android» (Game Applications Development for OS Android Platform), дисципліна «Інструментальна підтримка розробки комп'ютерних ігрових додатків»;
- «Розробка ігрових додатків на базі HTML-5 для WEB» (Game Applications development for WEB based on HTML-5) , дисципліна «Інструментальна підтримка розробки комп'ютерних ігрових додатків»;
- «3D графіка в ігрових додатках (на базі графічного редактора Blender)» (3D Graphics in Game Applications (Based on Blender Game Engine), дисципліна «Комп'ютерний синтез та обробка зображень»;
- «Місце ігрових додатків на ринку програмного забезпечення» (Place of Game Applications in Software Market), дисципліна «Сучасні засоби інформатики та комп'ютерний ринок».

Навчальний план підготовки студентів на рівні «бакалавр» передбачає опрацювання наступних навчальних модулів:

- «Архітектура ігрових додатків» (Game Applications Architecture), дисципліна «Архітектура та проектування програмного забезпечення»;

- «Основи створення ігрових додатків (на базі GameMaker)» (Basics of Game Applications Development with GameMaker Studio), дисципліна «Комп'ютерна графіка»;
- «Особливості тестування ігрових додатків» (Features of Game Applications Testing), дисципліна «Якість ПЗ та тестування»;
- «Командна розробка ігрових додатків» (Team Development of Game Applications), дисципліна «Групова динаміка і комунікації».

Для кожного модуля розроблено: довідник модуля, опорні конспекти (презентації) лекцій та методичні вказівки виконання лабораторних робіт і практичних занять за модулем. Методичні матеріали всіх модулів, доступні за посиланням *http://89.185.3.253:9090/login (login: guest, pass: gamehub)*.

Дане видання містить опис навчального модуля «Розробка кросплатформених ігрових додатків на HTML5» дисципліни «Інструментальна підтримка розробки комп'ютерних ігрових додатків» входять до дисциплін підготовки студентів освітнього ступеня «магістр» спеціальності 121 «Програмна інженерія».

ДОВІДНИК МОДУЛЯ

«Розробка кросплатформених ігрових додатків на HTML5»
дисципліни «Інструментальна підтримка розробки комп'ютерних ігрових додатків»
підготовки студентів освітнього рівня «магістр» за спеціалізацією «Програмне забезпечення мультимедійних систем для ігрових додатків» спеціальності 121 «Інженерія програмного забезпечення»

1 Вступ

Метою викладання навчальної дисципліни «Інструментальна підтримка розробки комп'ютерних ігрових додатків» є формування практичних знань та вмінь студента в області розробки ігрових додатків для сучасних платформ.

Основними завданнями вивчення дисципліни "Інструментальна підтримка розробки комп'ютерних ігрових додатків" є:

- знайомство із основними інструментальними засобами створення сучасних ігрових додатків для різних апаратних та програмних платформ, порівняльний аналіз особливостей, переваг і недоліків цих платформ, оцінка перспектив їх розвитку;
- огляд сучасних движків створення ігрових додатків, що дозволяють оптимізувати та прискорити процес розробки за допомогою ефективної реалізації модульної розробки додатку та ефективної підтримки кросплатформеності;
- оволодіння повним циклом розробки типового ігрового додатку, організація та контроль процесів планування, розробки, просування та підтримки ігрового додатку, вміння спланувати роботу команди розробників відповідно до спеціалізації кожного з них;
- реалізація типових елементів ігрових додатків у відповідності до платформи реалізації, оцінка ефективності організації ігрової механіки з урахуванням типових сценаріїв використання інтерфейсу користувача та різних способів взаємодії з платформою;
- урахування особливостей платформи реалізації ігрового додатку, вміння використати ці особливості належним чином, оптимізація програмного коду відповідно до можливостей апаратних платформ.

Змістовний модуль «Розробка кросплатформених ігрових додатків на HTML5» дисципліни «Інструментальна підтримка розробки комп'ютерних ігрових додатків» орієнтовно на оволодіння студентами практичних навичок створення кросплатформених ігрових додатків засобами платформи HTML5.

2 Опис модуля

Галузь знань:	12 «Інформаційні технології».
Напрям підготовки:	121 «Інженерія програмного забезпечення»
Рівень:	магістр
Назва дисципліни:	Інструментальна підтримка розробки комп'ютерних ігрових додатків
Назва змістовного модуля:	Розробка кросплатформених ігрових додатків на HTML5
Семестр:	1
Кількість кредитних одиниць:	дисципліна – 6,0, модуль – 2,0
Орієнтовна кількість годин:	дисципліна – 180, модуль – 60
Викладачі:	доцент, к.т.н. Цололо С.О. ст. викл. Дікова Ю.Л.

3 Мета та передбачувані результати вивчання модуля

3.1 Мета модуля

Мета модуля – ознайомити студентів з можливостями сучасних кросплаформенних фреймворків, навчити основами розробки ігрових додатків з використанням фреймворку Phaser, з'ясувати особливості організації роботи над мобільним ігровим додатком, проаналізувати реалізацію багатокористувацьких ігор і способи залучення користувачів.

3.1.1 Знання та їх використання

У разі успішного оволодіння матеріалами модуля студент буде:

- виконувати класифікацію ігрових мобільних додатків за типами проходження ігрового процесу, вміти виділити особливості розробки додатків відповідно до типу гри і платформи реалізації;
- здатен проаналізувати сучасні фреймворки для створення кросплатформенних ігрових додатків та обрати найоптимальніший з них відповідно до характеристик конкретної гри;
- реалізовувати інтерфейс користувача і елементи управління мобільною грою у фреймворку Phaser з урахуванням особливостей

керування процесом гри, які відповідають різним способам взаємодії користувачів з мобільною платформою;

- створювати анімацію і програмування руху персонажів за допомогою засобів Phaser, при цьому підкреслюючи особливості ігрової механіки поведінкою персонажів гри;
- виконувати налагодження, комплексне тестування і публікацію гри з оцінкою можливих шляхів залучення користувачів завдяки організації багатокористувацького ігрового процесу.

3.1.2 **Дослідницький навички**

За підсумками вивчення модулю студент повинен вміти аналізувати можливості сучасних мобільних платформ для реалізації ігрових додатків та обирати найбільш прийнятні інструменти та фреймворки для створення кросплатформенних мобільних ігрових додатків.

3.1.3 **Спеціальні вміння**

У разі успішного вивчення модуля студент буде вміти:

- оцінювати сучасні ігрові движки та фреймворки, що дозволять найбільш ефективно, просто та швидко реалізувати кросплатформенні мобільні ігрові додатки;
- використовувати сучасний універсальний фреймворк Phaser для розробки кросплатформенних мобільних ігрових додатків різних типів та рівнів складності;
- знати повний цикл розробки мобільних ігрових додатків та мати змогу підключитися до розробки додатку на будь-якому етапі реалізації.

3.1.4 **Соціальні вміння**

У разі успішного вивчення модуля студент буде вміти:

- працювати у команді для досягнення загального результату;
- критично ставитися до результатів групової та особистої роботи, вміти проводити самооцінку та групове оцінювання виконаних робіт;
- дотримуватися регламентних рамок, виконувати строки проведення та виконання робіт;
- застосовувати раніше отримані знання для засвоєння нових знань;

- обґрунтувати той чи інший спосіб діяльності при виконанні практичних завдань;
- презентувати ідеї, проміжні та підсумкові результати розробки, виділяти основні недоліки у розробці;
- своєчасно та самостійно виконувати пошук можливих помилок, знаходити способи їх усунення та попередження;
- обирати моделі та стилі спілкування;
- реалізовувати вербальні та невербальні способи спілкування;
- аргументовано критикувати дії інших членів команди та доводити доцільність свої рішень.

3.1.5 **Особисті якості**

У разі успішного вивчення модуля студент буде вміти:

- самостійно вирішувати питання, що відносяться усіх етапів розробки кросплатформенного ігрового додатку;
- вивчати та аналізувати онлайн-джерела та сучасну науково-технічну літературу з питань створення кросплатформенних ігрових додатків;
- оцінювати сучасні тенденції у створенні кромплатформенних ігрових додатків, аналізувати перспективність реалізації того чи іншого ігрового додатку.

4 Місце модуля в структурі дисципліни

Номер	*Змістовний модуль*	*Тиждень вивчання*
1	Розробка ігрових додатків на базі движка Unity	1-6
2	Розробка ігрових додатків для ОС Android	7-11
3	**Розробка кросплатформених ігрових додатків на HTML5**	12-16

5 Інформаційне наповнення змістовного модуля 3

Номер тижня	*Зміст*
12	Особливості платформи HTML5 та кроспларформені ігрові фреймворки.

Номер тижня	*Зміст*
	Основні принципи розробки ігрових HTML5-додатків на прикладі простої гри на Phaser.
13	Архітектура та базові компоненти ігрового HTML5-додатку на фреймворку Phaser. Створення шаблону додатку на Phaser із реалізацією типових ігрових станів
14	Огляд основних ігрових об’єктів фреймворку Phaser. Особливості організації ігрового процесу та основи роботи із фізичними движками в Phaser
15	Моделювання фізичних процесів та інші інструментальні засоби фреймворку Phaser. Тестування, відлагодження та шляхи прискорення додатку на Phaser. Підготовка залікового звіту і захист проекту.

6 Форми навчання

Навчальне навантаження модуля складається з аудиторної та самостійної роботи. Аудиторна робота включає 4 лекції, 3 лабораторні роботи та заключний практичний семінар із обговоренням розробленої гри.

Самостійна робота студентів передбачає підготовку до аудиторних занять і лабораторних робіт, а також підготовку до підсумкового тесту та оформлення залікового звіту. Підготовка до поточних аудиторних занять є аналіз літератури, електронних матеріалів з інтернету по темам лекцій і лабораторних робот, підготовка до проміжних та підсумкового тестів.

Лабораторні роботи 1-3 модуля орієнтовані на послідовне виконання наскрізного завдання та мають на меті відпрацювання навичок і вмінь розробки кросплатформенних ігрових HTML5-додатків за допомогою фреймворку Phaser.

Практичний семінар – заняття, на якому студент (група студентів) презентує гру, що була розроблена. Інші студенти оцінюють результат, виступаючи у ролі тестувальників та перших гравців. За результатами обговорення на семінарі до ігрової механіки вносяться зміни, що дозволять грі стати більш привабливою для широкої аудиторії.

Заліковий звіт – докладний опис та презентація графічного інтерфейсу кросплатформенного ігрового додатку, розробленого за допомогою фреймворка Phaser при виконанні лабораторних робіт 1-3.

7 Порядок проведення атестації

Загальний принцип оцінювання підсумкових знань студента з курсу «Інструментальна підтримка розробки комп'ютерних ігрових додатків» полягає в тестуванні студентів на лекціях, оцінці поточної практичної роботи студента у навчальному семестрі на лабораторних роботах та оцінки контрольного заходу у формі іспиту, у результаті котрих студент має сумарну оцінку в балах.

За результатами вивчання кожного змістовного модуля передбачено оцінка виконання залікового завдання. Оцінка включає: результати тестування студентів на лекціях, результати поточного опитування при виконання лабораторних робіт модуля, оцінку за виконання залікового завдання. Сумарно оцінка кожного змістовного модуля не може бути більш ніж 20 балів. Максимальна оцінка іспиту 40 балів.

Оцінка результатів вивчення дисципліни в цілому:

Поточне тестування	*Балів*
Змістовний модуль 1	до 20
Змістовний модуль 2	до 20
Змістовний модуль 3	**до 20**
Іспит	до 40
Разом	до 100

Графік проведення поточного оцінювання модуля:

Номер тижня	*Оцінювання*
13	Оцінка виконання лабораторної роботи 1
14	Оцінка виконання лабораторної роботи 2
15	Оцінка виконання лабораторної роботи 3
16	Оцінка залікового звіт і захисту проекту

Представлення звіту щодо виконання лабораторних робіт.

При представленні звіту з лабораторних робіт необхідно виконувати наступне. При умові виконання кожної окремої лабораторної роботи єдиний звіт надається студентом на 16 тижні семестру. Продовження

терміну можливе лише при наявності поважної причини, передбаченої порядком навчання студентів у вищій школі. При цьому:

- електронна версія єдиного звіту надається викладачеві через інтернет на початку 16 тижня;
- під час участі у практичному семінарі на 16 тижні студент демонструє закінчений варіант гри;
- за кожний день прострочки представлення та здачі єдиного звіту по модулю 2 знімається 1 бал (не більш ніж 5 днів – 5 балів)

Метод оцінки змістовного модуля

Кількість балів в загальній оцінці змістовного модулю відповідає наступному:

Виконання лабораторної роботи 1	максимально 4 бали.
Виконання лабораторної роботи 2	максимально 4 бали.
Виконання лабораторної роботи 3	максимально 4 бали.
Участь у практичному семінарі	максимально 4 бали.
Оцінка залікового звіту і захисту проекту	максимально 4 бали.

Усі набрані бали підсумовуються (максимально 20 балів), штрафні бали за запізнення в представленні єдиного звіту (максимально мінус 5 балів) віднімаються. Сумарна оцінка (від 0 до 20 балів) є індивідуальною оцінкою студента із засвоєння змістовного модуля 2.

Метод оцінки дисципліни в цілому

Оцінки студентів за результатами вивчення змістовних модулів 1-3 підсумовуються. Далі додається оцінка іспиту (максимально 40 балів) та розраховується сумарна оцінка студента в балах за дисципліною. Сумарна оцінка в балах, переводиться за нижченаведеною шкалою оцінювання в національну:

Сума балів за всі види навчальної діяльності	*Оцінка за національною шкалою*
90-100	Відмінно
82-89	Добре
74-81	Добре
64-73	Задовільно
60-63	Задовільно

35-59	не зараховано з можливістю повторного складання
0-34	не зараховано з обов’язковим повторним вивченням дисципліни

8 Зворотній зв’язок

Інформація щодо результатів тестування, виконання лабораторних робіт та загальна оцінка змістовного модулю надається кожному студенту як індивідуально, так і всієї групи в цілому.

Інформація щодо результатів тестування надається студентам на 16 тижні навчання. Інформація щодо оцінки виконання лабораторної роботи надається студентові під час заняття. Інформація щодо оцінки змістовного модуля в цілому надається студентам на 16 тижні навчання.

Контактні дані для online-допомоги та консультування:

Викладачі:

- доцент, к.т.н. Цололо С.О., sergii.tsololo@donntu.edu.ua,
- ст. викл. Дікова Ю.Л., yuliia.dikova@donntu.edu.ua

9 Викладацький склад та допоміжні джерела

Обов'язки викладачів:

- подача матеріалів модуля згідно з програмою;
- оцінка результатів тестування та виконання лабораторних робіт.

Обов'язки координатора дисципліни:

- планування та внесення змін до модуль;
- координація і управління професорсько-викладацьким складом;
- координація проведення тестування, лабораторних робіт та іспиту.

Обов'язки допоміжного персоналу:

Допоміжний персонал здійснює підготовку комп’ютерної техніки до виконання лабораторних робіт студентами та надає технічну підтримку студентів під час виконання лабораторних робіт.

Контактні дані викладачів:

доцент, к.т.н. Цололо С.О., sergii.tsololo@donntu.edu.ua,

ст. викл. Дікова Ю.Л., yuliia.dikova@donntu.edu.ua

Контактні дані куратора:

професор, д.т.н. Башков Є.О., eabashkov@i.ua

10 Навчальна програма і матеріали

10.1 Тема 1. Особливості платформи HTML5 та кросплаформені ігрові фреймворки. Основні принципи розробки ігрових HTML5-додатків на прикладі простої гри на Phaser.

10.1.1 Мета та очікувані результати

Ознайомити студентів з ключовими можливостями та особливостями використання сучасних кросплаформенних ігрових фреймворків,а також розглянути базові принципи розробки HTML5-додатків у фреймворку Phaser та створити простий тестовий додаток.

Очікуваними результатами є:

- огляд ключових сучасних ігрових фрейморків, що орієнтовані швидке та ефективне створення ігрових додатків різного рівню складності;
- розробка простого тестового додатку у фреймворку Phaser.

10.1.2 Лекція

Лекція знайомить із особливостями платформи HTML-5 для створення додатків та використанням фреймворків при розробці ігрових додатків, також розглядаються та аналізуються основні актуальні фрейворки, їх розподіл за призначенням та складністю, основні можливості та особливості використання.

Мета лекції:

- ознайомитися із основними складовими платформи HTML5, що використовуються при створення додатків;
- дізнатися про існуючі інструментальні засоби перетворення HTML5-додатків в нативні додатки мобільних операційних систем;
- ознайомитися із підходом використання фреймворків для розробки кросплатформенних ігрових додатків;
- классифікувати ігрові фреймворки за типами ігрових додатків, до яких підходять конкретні екземпляри фреймворків;

- проаналізувати переваги, особливості та можливості різних фреймворків (Construct 2, ImpactJS, EaselJS, Cocos2d-x, libGDX), виділити з низ найбільш підходящі для мобільної розробки;
- розглянути інструменти та середовища розробки ігрових кросплатформенних додатків.

Основні результати лекції відповідають вище передбаченим цілям.

10.1.3 Лабораторна робота 1. Основні принципи розробки ігрових HTML5-додатків на прикладі простої гри на Phaser.

Лабораторна робота орієнтована на ознайомлення студентів із архітектурою HTML5-додатків та основними принципами організації проекту на прикладі класичної гри «Змійка», реалізованої засобами фреймворку Phaser.

Мета лабораторної роботи:

- ознайомити студентів із архітектурою та основними складовими типового HTML5-додатку;
- розглянути варіанти організації файлової структури проекту для розміщення усіх необхідних скриптів та ресурсів (зображення, аудіофайли тощо);
- ознайомитися із основними складовими головного ігрового об’єкту Phaser.Game;
- ознайомитися із основними ігровими станами, способами їх ініціалізації та переходу між ними;
- розглянути особливості процесу завантаження та зберігання ігрових ресурсів;
- навчитися створювати ігрові об’єкти із ресурсів та використовувати масиви об’єктів, створених на основі єдиного ресурсу;
- розглянути питання організації керування персонажем за допомогою кнопок клавіатури;
- ознайомитися із способами фіксації та обробки зіткнень між ігровими об’єктами;
- розглянути особливості організації підрахунку та виводу на екран ігрових очок та інших параметрів гри;
- навчитися керувати швидкістю гри шляхом контролю основного ігрового циклу.

У разі успішного виконання лабораторної роботи студент отримає простий і, але повністю закінчений ігровий HTML5-додаток із класичним ігровим процесом, готовий до вдосконалення та розширення у рамках самостійної роботи.

Успішне засвоєння матеріалів та виконання лабораторної роботи сприяє формування у студента наступних соціальних навичок та вмінь:

– реалізувати свою діяльність творчо, проявляти пізнавальну активність;
– дотримуватися регламентних рамок, виконувати строки проведення та виконання робіт;
– застосовувати раніше отримані знання як основу для засвоєння нових знань;

що мають такі форми прояву:

– прагнення вирішувати поточні проблеми самостійно, або із залученням членів команди;
– прийняття оголошених цілей й бачення шляхів їх досягнення;
– прагнення пропонувати нові ідеї і втілювати їх;
– глибокий аналіз проблем і пошук нових можливостей;
– націленість на успіх, прагнення отримати найкращі результати;
– вміння планувати свій час та діяльність групи;
– дотримання регламентних рамок;
– дисциплінованість та зібраність;
– робити обґрунтоване пояснення до рішення навчальних завдань;
– застосовувати навички з планування послідовності виконання завдань.

10.1.4 **Методичні матеріали та вказівки**

Методичні матеріали та вказівки доступні за посиланням http://89.185.3.253:9090/login (login: *guest*, pass: *gamehub*), директорія *04_M_Game Applications development for WEB based on HTML-5*.

Опорний конспект лекції № 1 дивись

04_M_Lec_1 Game Applications development for WEB based on HTML-5.pdf

Методичні вказівки щодо виконання лабораторної роботи дивись

04_M_Lab Game Applications development for WEB based on HTML-5.pdf

10.1.5 **Питання, що виносяться на іспит.**

1. Основні елементи платформи HTML5.
2. Основні переваги HTML5 із точки зору створення ігрових додатків.
3. Способи розміщення HTML5-додатків.
4. Засоби перетворення HTML5-додатків в нативні додатки мобільних ОС.
5. Які популярні інструменти HTML5-to-app ви знаєте, вкажіть їх переваги та недоліки.
6. Загальна архітектура ігрових HTML5-додатків.
7. Фреймворки для розробки кросплатформенних додатків.
8. Классифікація ігрових фреймворків.
9. Переваги, особливості та можливості різних фреймворків (Construct 2, ImpactJS, EaselJS, Cocos2d-x, libGDX).

10.2 **Тема 2. Архітектура та базові компоненти ігрового HTML5-додатку на фреймворці Phaser. Створення шаблону додатку на Phaser із реалізацією типових ігрових станів.**

10.2.1 **Мета та очікувані результати**

Ознайомити студентів з особливостями архітектури та базовими компонентами ігрового HTML5-додатку на фреймворці Phaser, а також більш докладно розглянути типові режими ігрового додатку, організацію переходів між ними.

Очікуваними результатами є:

- огляд архітектури та основних елементів ігрового HTML5-додатку на фреймворці Phaser;
- розробка шаблону додатку у фреймворку Phaser із реалізацію переходів між різними ігровими станами.

10.2.2 **Лекція**

Лекція знайомить студентів з архітектурою кросплатформенного ігрового додатка на Phaser та основними компонентами, з яких будується додаток. Виконується огляд процесу проектування та розробки гри з точки зору гнучкості та ефективності повторного використання вже існуючих елементів.

Метою лекції є розгляд таких питань:

- архітектура додатку на Phaser;

- приклад типового додатку на Phaser;
- основні базові класи (Game, World, Camera);
- типові стани, в яких може знаходитися додаток Phaser (клас Stage);
- особливості завантаження та кешування елементів гри (класи Loader та Cache);
- система подій на основі сигналів (клас Signal);
- створення та групування ігрових об’єетів (класи GameObjectFactor, GameObjectCreator, Group);
- масштабування елементів (клас ScaleManager);
- робота із менеджером введення (клас InputHandler);
- робота із доповненнями (клас PluginManager).

Основні результати лекції відповідають вище передбаченим цілям.

10.2.3 **Лабораторна робота 2. Створення шаблону додатку на Phaser із реалізацією типових ігрових станів**

Лабораторна робота орієнтована на оволодіння студентами навичок із організації в ігровому додатку системи станів, зв’язаних між собою. В роботі докладно розглянутий процес створення базового шаблону ігрового додатку, який в подальшому часі може бути використаний для побудови HTML5-додатку любого жанру.

Мета лабораторної роботи:

- докладно ознайомитися із системою типових ігрових станів, які є основою будь-якого додатку на Phaser;
- ознайомитися із реалізацією попереднього завантаження ресурсів із підтримкою прогрес-бару;
- розглянути механізм та особливості використання в ігровому додатку користувацьких шрифтів;
- ознайомитися із завантаженням та керування звуковими файлами і ігровому додатку;
- розглянути особливості використання часових затримок та їх налаштувань відповідно до особливостей гри;
- розглянути способи організацій головного ігрового меню та створення універсальних засобів для розміщення меню в будь-якому стані додатку;
- ознайомитися із можливими варіантами повторного використання загальної таблиці стилів для однотипних ігрових елементів;

- розглянути особливості використання механізму домішок (mixins) для запобігання багаторазового дублювання коду;
- ознайомитися із створенням глобальних ігрових параметрів для використання в будь-якому стані додатку.

У разі успішного виконання лабораторної роботи студент отримає як навички у розбудові ігрових проектів та реалізації типових ігрових станів, так й готовий універсальний шаблон ігрового додатку на Phaser, який може використати як основу для роботи над грою будь-якого жанру.

Успішне засвоєння матеріалів та виконання лабораторної роботи сприяє формуванню у студента наступних соціальних навичок та вмінь:

- реалізувати свою діяльність творчо, проявляти пізнавальну активність;
- обґрунтувати той чи інший спосіб діяльності при виконанні практичних завдань;
- приймати та підтримувати правила ділової поведінки, усної комунікації, партнерських відносин;

що мають такі форми прояву:

- прагнення до самореалізації та нових форм діяльності;
- висловлення нестандартних ідей;
- прояви винахідливості, кмітливості, допитливості;
- бажання йти на розумний ризик при прийнятті рішень;
- уміння створювати творчу атмосферу і позитивний мікроклімат;
- застосовувати навички з планування послідовності виконання завдань;
- використовувати аналіз помилок, допущених у особистій чи груповій роботі, для знаходження правильного вирішення завдання;
- вміти побудувати довірче, взаємовигідне спілкування, підтримувати реалізацію групових планів та дій.

10.2.4 **Методичні матеріали та вказівки**

Методичні матеріали та вказівки доступні за посиланням http://89.185.3.253:9090/login (login: *guest*, pass: *gamehub*), директорія *04_M_Game Applications development for WEB based on HTML-5*.

Опорний конспект лекції № 2 дивись

04_M_Lec_2 Game Applications development for WEB based on HTML-5.pdf

Методичні вказівки щодо виконання лабораторної роботи дивись

04_M_Lab Game Applications development for WEB based on HTML-5.pdf

10.2.5 Питання, що виносяться на іспит.

1. Базові класи фреймворку Phaser.
2. Основні режими (стани) гри.
3. Завантаження ресурсів.
4. Сигнали та подієва модель.
5. Способи створення елементів.
6. Масштабування елементів.
7. Робота із доповненнями.
8. Організація користувацького введення.
9. Групування елементів.

10.3 Тема 3. Огляд основних ігрових об’єктів фреймворку Phaser. Особливості організації ігрового процесу та основи роботи із фізичними движками в Phaser.

10.3.1 Мета та очікувані результати

Ознайомити студентів із основними класами фреймворку Phaser, що використовуються для створення основних та допоміжних ігрових об’єктів. Розглянути особливості Phaser з точки зору реалізації ігрових додатків з підтримкою фізичних движків.

Очікуваними результатами є:

- огляд основних ігрових об’єктів та їх можливостей для організації ігрового процесу на Phaser;
- розробка додатку у фреймворку Phaser із підтримкою основних об’єктів і механізмів фізичного движка P2.

10.3.2 Лекція

Лекція знайомить студентів з основними ігровими об’єктами, що можуть бути додані до гри, особливостями їх створення та налаштувань, а також взаємодії між собою.

Метою лекції є розгляд таких питань**:**

- зображення, звичайні та тайлові спрайти (класи Sprite, TileSprite та Image);
- пакети спрайтів (клас SpriteBatch);

- графіка та бітові карти(класи Graphics та BitmapData);
- робота із текстурами (клас RenderTexture);
- звичайний та текстурний текстові елементи (класи Text та BitmapText);
- робота із анімаціями (класи Animation, AnimationParser, FrameData, Frame);
- типи часових затримок та їх використання (класи Time, Timer та TimerEvent);
- особливості використання тайлових карт (класи Tilemap, TilemapLayer, Tileset, Tile та TilemapParser);
- використання системи частинок (класи Particles, Emitter та Particle).

Основні результати лекції відповідають вищепередбаченим цілям.

10.3.3 Лабораторна робота 3. Особливості організації ігрового процесу та основи роботи із фізичними движками в Phaser.

Лабораторна робота орієнтована на отримання студентами навичок із організації ігрового процесу, створення ігрових додатків із підтримкою фізичних движків фреймворку Phaser та випадковою генерацією ігрових елементів.

Мета лабораторної роботи:

- навчитися готувати ресурси, необхідні для реалізації ігрових додатків із підтримкою фізичних движків (моделі об’єктів, фізичні карти тощо);
- ознайомитися із особливостями створення і налаштування об’єктів для використання в рамках фізичних процесів ігрового движка P2;
- навчитися налаштовувати різноманітні матеріали, що є основною взаємодії в фізичному движку P2, а також створювати контактні матеріали на основі базових із налаштуванням параметрів взаємодії;
- ознайомитися із відмінностями візуальних і фізичних об’єктів движку P2 та засобами перетворення координат між різними представленнями;
- навчитися прив’язувати об’єкти один до одного із налаштуванням різноманітних типів обмежень (constraints) та вказувати точки прив’язки;

– навчитися організувати користувацьке керування ігровим об’єктом із урахуванням особливостей фізичного движка та поточного положення об’єкту;
– ознайомитися із способами динамічної генерації контенту (поверхонь або ландшафтів) на основі масивів вершин, що були згенеровані шляхом наповнення випадковими значеннями.

У разі успішного виконання лабораторної роботи студент оволодіє знаннями щодо особливостей застосування фізичного движка P2, використання фізичних карт об’єктів, базових і контактних матеріалів, прив’язування об’єктів та створить алгоритм автоматичної генерації ігрової поверхні на основі випадкових значень.

Успішне засвоєння матеріалів та виконання лабораторної роботи сприяє формуванню у студента наступних соціальних навичок та вмінь:
– вміння роботи у команді, на досягнення загального результату;
– своєчасно та самостійно виконувати пошук можливих помилок, знаходити способи їх усунення та попередження;
– дотримуватися регламентних рамок, виконувати строки проведення та виконання робіт;

що мають такі форми прояву:
– приймати участь у груповому проектуванні, обговореннях, оцінюванні результатів виконаних робіт;
– бути здатним до прийняття та узгодження колективного рішення;
– вміння створювати творчу атмосферу;
– застосовувати навички з планування послідовності виконання завдань;
– використовувати аналіз помилок, допущених у особистій чи груповій роботі, для знаходження правильного вирішення поставленого завдання.

10.3.4 **Методичні матеріали та вказівки**

Методичні матеріали та вказівки доступні за посиланням http://89.185.3.253:9090/login (login: *guest*, pass: *gamehub*), директорія *04_M_Game Applications development for WEB based on HTML-5*.

Опорний конспект лекції № 3 дивись

04_M_Lec_3 Game Applications development for WEB based on HTML-5.pdf

Методичні вказівки щодо виконання лабораторної роботи дивись

04_M_Lab Game Applications development for WEB based on HTML-5.pdf

10.3.5 **Питання, що виносяться на іспит.**

1. Типи спрайтів та особливості їх використання.
2. Використання пакетних спрайтів.
3. Робота із графічними примітивами.
4. Особливості використнання текстур.
5. Використання тайлових карт.
6. Інструменти роботи із текстом.
7. Класи для роботи із анімацією.
8. Робота із часом і затримками.
9. Система частинок і сфери її використання.

10.4 **Тема 4 Моделювання фізичних процесів та інші інструментальні засоби фреймворку Phaser. Тестування, відлагодження та шляхи прискорення додатку на Phaser. Підготовка залікового звіту і захист проекту.**

10.4.1 **Мета та очікувані результати**

Проаналізувати типи фізичних движків, що підтримуються в Phaser, виділити їх переваги, недоліки та сфери використання, розглянути додаткові інструментальні засоби, що надаються фреймворком. Ознайомитися із процесом тестування та налагодженням ігрового балансу в грі, а також з особливостям роботи HTML5-додатків в режимах Canvas та WebGL.

Очікуваними результатами є:

- огляд фізичних движків та рекомендації щодо їх вибору відповідно до жанру та специфіки ігрового додатку;
- приклади використання додаткових засобів фреймворку Phaser, орієнтованих на відлагодження та тестування додатків;
- підготовка заключного звіту з докладним описом ігрового додатку, розробленого протягом виконання лабораторних робіт.

10.4.2 Л**екція**

Лекція знайомить з основними видами фізичних движків, що доступні у Phaser. Розглядаються особливості та відмінності, на яких грунтуються відповідні ігрові жанри. Наведений опис основних додаткових

інструментів для тестування та налагодження додатку. Також розглядаються варіанти інтеграції соціальних елементів у ігровий додаток та використання соціальних мереж для реклами додатку та залучення нових гравців.

Метою лекції є розгляд таких питань:

- аркадний ігровий движок (Arcade Phisics), його особливості, переваги і недоліки;
- fullbody-движок P2, сфери та особливості його використання;
- фізичний движок Ninja Physics та його особливості;
- використання системи проміжних кадрів (tweens) для створення анімації та підтримки ігрової механіки (класи TweenManager, Tween, TweenData та Easing);
- особливості різних способів уведення (класи Input, Pointer, DeviceButton, Keyboard ,Key,KeyCode, Mouse та Touch);
- математичний модуль та функції переходів (класи Angle, Distance, Easing, Fuzzy, Interpolation, Pow2, Snap, Random Data Generator);
- підтримка мережи та веб-доступу (клас Net);
- робота із звуковими файлами та звукові спрайти (класи SoundManager, Sound та AudioSprite);
- контроль характеристик пристрою, на якому працює додаток (клас Device);
- модуль обробки даних та рефлексії (класи ArraySet, ArrayUtils, Color, Debug, LinkedList та Utils).

10.4.3 **Практичний семінар. Тестування, відлагодження та шляхи прискорення додатку на основі фреймворку Phaser.**

Заключний практичний семінар орієнтовано на обговорення та тестування створеного студентом при виконанні лабораторних робіт 2-3 кросплатформеного ігрового HTML5-додатку за допомогою фреймворку Phaser.

Мета практичного семінару:

- оцінити правильність визначення концепції ігрового додатку та коректності відображення основних ігрових об’єктів;

- оцінити правильність визначення поведінки та взаємодії ігрових об’єктів відповідно до основних принципів використаної ігрової механіки і фізичного движка;
- оцінити як саме були враховані обмеження та особливості платформи HTML5 та порівняти поведінку додатку при запуску в режимах Canvas та WebGL;
- оцінити можливі вдосконалення та зміни, які треба внести до ігрового додатку задля його коректного перетворення в мобільний ігровий додаток за допомогою HTML-to-app;
- оцінити привабливість проекту та надати рекомендації щодо способів подальшого розвитку розробленого мобільного ігрового додатку.

На заключному семінарі кожен студент виступає з презентацією своєї розробки, в якій повинен:

- надати відомості щодо головної мети та завдань розробки кросплатформеного ігрового додатку;
- обґрунтувати основі рішення, прийняті при виконанні лабораторних робіт;
- продемонструвати розроблений ігровий додаток як у браузері, так і на мобільному пристрої (завдяки технології HTML-to-app);
- визначити основні вади і недоліки розробленого додатку;
- запропонувати щонайменше п’ять основних покращень, що можуть бути внесені до проекту.

Крім того доповідач повинен надати можливість тестування розробленого додатку. Викладач довільно призначає студента, який тестує розроблений ігровий додаток і виступає як опонент доповідача, висвітлюючи вади та недоліки розробки.

В ході обговорення студенти групи оцінюють правильність визначення основних графічних елементів ігрового додатку, правильність визначення основних подій та дій об’єктів (ігрової механіки) та визначають рекомендації щодо подальшого розвитку розробленого ігрового додатку.

За результатами обговорення на практичному семінарі студенти навчаються оцінювати ігрові додатки, створені на основі фреймворку Phaser, виявляти недоліки та помилки, що були допущені на етапах

розробки графічного інтерфейсу, ігрової механіки та тестування ігрового додатку, визначати подальші шаги розвитку додатку з урахуванням результатів обговорення.

Успішне засвоєння матеріалів практичного семінару сприяє формуванню у студента наступних соціальних навичок та вмінь:

- приймати та підтримувати правила ділової поведінки, усної комунікації, партнерських відносин;
- своєчасно та самостійно виконувати пошук можливих помилок, знаходити способи їх усунення та попередження;
- критично ставитися до результатів групової та особистої роботи, вміти проводити самооцінку та групове оцінювання виконаних робіт;

що мають такі форми прояву:

- вміння формулювати та доносити до співрозмовника свої думки;
- здатність узгоджувати внутрішні бажання з колективними потребами;
- використовувати прийоми ділового спілкування (публічного мовлення, презентаційних виступів, доповідей) раніше отримані знання як основу для засвоєння нових знань;
- дотримуватися мотиваційних принципів в роботі та навчанні;
- дотримуватись етичних норм ділового та партнерського спілкування;
- критичне ставлення до результатів групової та особистої роботи;
- мати навички в усуненні та запобіганні конфліктів;
- демонструвати доброзичливе та поважливе відношення до пропозицій учасників спільного проекту;
- формування елементів самоконтролю і самооцінки при виконанні діяльності.

10.4.4 **Методичні матеріали та вказівки**

Методичні матеріали та вказівки доступні за посиланням http://89.185.3.253:9090/login (login: *guest*, pass: *gamehub*), директорія *04_M_Game Applications development for WEB based on HTML-5*.

Опорний конспект лекції № 4 дивись

04_M_Lec_4 Game Applications development for WEB based on HTML-5.pdf

Методичні вказівки щодо підготовки до практичного семінару дивись

04_M_Lab Game Applications development for WEB based on HTML-5.pdf

10.4.5 Питання, що виносяться на іспит

1. Типи фізичних движків в Phaser та їх особливості.
2. Вибір фізичного движка відповідно до жанру та особливостей ігрової механіки додатку.
3. Особливості використання системи проміжних кадрів.
4. Реалізація користувацького введення різними способами.
5. Що таке функції переходів і де вони використовуються?
6. Отримання інформації про пристрій та мережеві задачі.
7. Робота із звуковими файлами.
8. Особливості використання аудіо-спрайтів
9. Додаткові класи роботи із даними.

11 Рекомендована література та інтернет-посилання

1. K. Dyrr, P.F. Navarro, R. Oliveira, B. Sparks. Game Development for Human Beings. Build Cross-Platform Games with Phaser – Zenva Pty Ltd, 2016. – 472 p.
2. Phaser – A fast, fun and free open source HTML5 game framework. – https://phaser.io/.
3. Learn How To Make HTML5 Games.– https://www.discoverphaser.com
4. Phaser Community – https://phaser.io/community.
5. Phaser Editor – A friendly IDE for HTML5 games creation. – http://phasereditor.boniatillo.com/.
6. How to Learn the Phaser HTML5 Game Engine. – https://gamedevelopment.tutsplus.com/articles/how-to-learn-the-phaser-html5-game-engine--gamedev-13643.

Discussion about the Phaser game framework. – http://www.html5gamedevs.com/forum/14-phaser/

МЕТОДИЧНІ ВКАЗІВКИ
ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ

Дисципліна: Інструментальна підтримка розробки комп'ютерних ігрових додатків

Модуль: Розробка кросплатформених ігрових додатків на HTML5

1 Вступ

Комплекс лабораторних робот модуля «Розробка кросплатформених ігрових додатків на HTML5» виконується відповідно до [1], передбачує виконання трьох лабораторних робіт та має на меті відпрацювання навичок розробки кросплатформених ігрових HTML5-додатків на основі фреймворку Phaser.

До комплексу лабораторних робіт входить заключний практичний семінар – заняття, на якому студент презентує ігровий кросплатформений HTML5-додаток, що було розроблено протягом виконання лабораторних робіт. При цьому інші студенти групи оцінюють результат, виконуючи роль тестувальників ігрового додатку. За результатами обговорення на семінарі сформулюються зміни, що мають бути внесені до остаточної версії ігрового додатку.

Заліковий звіт – надання докладного письмового звіту та презентація в групі ігрового HTML5-додатку, що був розроблений при послідовному виконанні завдання лабораторних робіт 2-3.

Самостійна робота студентів передбачає вивчення додаткової літератури [2-7], підготовку до лабораторних робіт, а також підготовку та оформлення залікового звіту та презентації за результатами виконання лабораторних робіт.

1.1 Графік виконання лабораторних робіт

Номер тижня	Виконання	Оцінювання (максимальний бал)
9	Лабораторна робота №1	3
10	Лабораторна робота №2	3
11	Лабораторна робота №3	3
12	Презентація та обговорення отриманих результатів на заключному практичному семінарі	4
	Заліковий звіт	2
Максимальний сумарний бал		15

1.2 Оцінка виконання комплексу лабораторних робіт.

Критерії оцінювання виконання кожної лабораторної роботи наведені в додатку А.

Критерії оцінювання обговорення результатів на заключному практичному семінарі наведені в додатку Б.

Критерії оцінювання заключного звіту наведені в додатку В.

Інформація щодо оцінки змістовного модуля в цілому надається студентам на 13 тижні навчання.

1.3 Контактні дані для online допомоги та консультування:

Викладачі :

- доцент, к.т.н. Цололо С.О., sergii.tsololo@donntu.edu.ua.
- ст. викл. Дікова Ю.Л., yuliia.dikova@donntu.edu.ua

2 Завдання до лабораторних робіт

Лабораторні роботи за модулем присвячені розробці декількох кросплатформених ігрових HTML5-додатків на основі засобів фреймворку Phaser. Підбір завдань та організація роботи над ними відповідає усім ключовим етапам проектування, розробки, тестування та підготовки до публікації ігрового додатку.

Отже, модуль складається із трьох лабораторних робіт:

1. *Знайомство із основними принципами розробки HTML5-додатків на прикладі простої гри на Phaser* – студентам пропонується огляд базових принципів побудови і розгортання ігрового додатку на Phaser. У результаті роботи студент отримують простий і повнофункціональний HTML5-додаток.
2. *Створення шаблону додатку на Phaser із реалізацією типових ігрових станів* – студентам пропонується розробити базовий шаблон ігрового додатку, який в ЛР №3 буде використаний у якості шаблону для розміщення розробленого ігрового процесу.
3. *Особливості організації ігрового процесу та основи роботи із фізичними движками в Phaser* – студенти знайомляться із принципами використання фізичного движка P2 на прикладі гри автомобільної тематики. Підсумковий додаток, за яким складається заключний звіт, формується шляхом інтеграції розробленого ігрового процесу до шаблону, який було створено в ЛР №2.

Таким чином, наприкінці виконання ЛР №3 студенти отримують повнофункційний ігровий HTML5-додаток із підтримкою обробки фізичних

процесів та типових ігрових станів. Саме цей додаток студент представляє на підсумковому практичному семінарі.

Таким чином, головною метою і результатом виконання даного модуля є реалізація повного циклу розробки кросплатформеного ігрового додатку, а також створення шаблону ігрового додатку із реалізацією типових станів гри. Цей шаблон може бути використаний у майбутньому у якості основи для інтеграції ігрового процесу любого жанр та рівня складності, що продемонстровано в ЛР №3.

3 Обладнання

Виконання лабораторних робіт передбачає створення кросплатформених ігрових додатків на основі фрейворку Phaser, що що використовує можливості стеку технологій HTML5 + CSS3 + JavaScript. Тому мінімальні вимоги до характеристик платформи є дуже гнучкими:

- сучасна операційна система (Windows, Linux, MacOS, тощо);
- браузер (Chrome, Firefox, Edge, Opera тощо);
- локальний веб-сервер (OSPanel, Denwer, NodeJS, LAMP тощо) для розміщення та тестування додатків;
- текстовий редактор із підсвічуванням синтаксису (Notepad++, Sublime Text, AkelPad тощо);
- фреймворк Phaser (https://phaser.io/).

4 Попередні підготовчі кроки до виконання лабораторних робіт

Для виконання лабораторних робіт необхідно оснастити робоче місце відповідним набором інструментальних засобів розробки програмного забезпечення.

Розробка додатків в рамках стеку технологій HTML5 не вимагає встановлення складних інструментальних засобів, редагування може відбуватися в будь-якому текстовому редакторі. Але краще все ж обрати редактор із підтримкою підсвічування синтаксису. Наприклад, це може бути NotePad++ для Windows, який можна встановити за посиланням https://notepad-plus-plus.org/download/ (рис. 4.1). Звичайно, для інших операційних систем можна обрати будь-який інший редактор, бо це не має принципового значення.

HTML5-додаток являє собою набір JS-скриптів у купі із необхідними ресурсами, тобто для його запуску достатньо лише браузера – бажано найновішого, але це не має суттєвого значення. При чому запускати скрипти можна прямо із папки на локальному диску. Але сучасні браузери мають неприємну особливість – вони не завжди коректно обробляють усе те, що завантажується із локальних папок. Тоді як робота із скриптами, що були завантажені з веб-серверів, не викликає жодних нарікань. Це пов’язане із необхідністю гарантування браузерами певного рівня безпеки.

Рисунок 4.1 – Завантаження редактора NotePad++

Рішенням для веб-розробників є встановлення компактних веб-серверів на свою власну робочу станцію. Такий веб-сервер є повноцінним рішенням за аналогією із віддаленими серверами, лише з тією відмінністю, що він запускається вручну та працює локально.

Очевидним шляхом отримання веб-серверу у себе на робочому місці буде завантаження та встановлення Apache, NGINX або NodeJS. Але із урахуванням того, що налаштування цих засобів займає деякий час і не є головною нашою метою, то більш ефективними та швидкими в розгортанні будуть рішення «все-в-одному». Вони містять усі необхідні компоненти веб-серверів, причому ці компоненти автоматично пов’язуються між собою. Зазвичай вони називаються xAMP-рішеннями за першими літерами прогам, що входять до комплексу (A – Apache, M – MySQL, P – PHP). Причому «x» відповідає операційній системі (W – Windows, M – MacOS, L – Linux).

На час створення методичних вказівок найбільш популярними рішеннями такого типу є (рис. 4.2):

– OSPanel для Windows (https://ospanel.io/);
– MAMP для MacOS (https://www.mamp.info/en/);
– Lamp-server для Ubuntu (http://help.ubuntu.ru/wiki/lamp).

Рисунок 4.2 – Завантаження xAMP-рішення

Отже, необхідно обрати одне з цих рішень відповідно до операційної системи, завантажити його та встановити. На цьому процес налаштування оточення є закінченим.

5 Лабораторна робота № 1. Основні принципи розробки ігрових HTML5-додатків на прикладі простої гри на Phaser.

Перша лабораторна робота орієнтована на ознайомлення студентів із архітектурою HTML5-додатків та основними принципами організації проекту на прикладі класичної гри «Змійка», реалізованої засобами фреймворку Phaser. Вибір класичної гри для ознайомлення із фреймворком обумовлений прагненням розглянути саме особливості фреймворку, коли організація ігрового процесу є нескладною і відомою.

Мета лабораторної роботи:

– ознайомити студентів із архітектурою та основними складовими типового HTML5-додатку;
– розглянути варіанти організації файлової структури проекту для розміщення усіх необхідних скриптів та ресурсів (зображення, аудіофайли тощо);
– ознайомитися із основними складовими головного ігрового об’єкту Phaser.Game;

- ознайомитися із основними ігровими станами, способами їх ініціалізації та переходу між ними;
- розглянути особливості процесу завантаження та зберігання ігрових ресурсів;
- навчитися створювати ігрові об’єкти із ресурсів та використовувати масиви об’єктів, створених на основі єдиного ресурсу;
- розглянути питання організації керування персонажем за допомогою кнопок клавіатури;
- ознайомитися із способами фіксації та обробки зіткнень між ігровими об’єктами;
- розглянути особливості організації підрахунку та виводу на екран ігрових очок та інших параметрів гри;
- навчитися керувати швидкістю гри шляхом контролю основного ігрового циклу.

Відповідно до завдання лабораторної роботи студент створює ігровий додаток «Змійка» із трьома основними станами та основним ігровим процесом, що включає організацію переміщення персонажа, збирання та підрахунок предметів, а також поступове збільшення швидкості гри.

У разі успішного виконання лабораторної роботи студент отримає простий і, але повністю закінчений ігровий HTML5-додаток із класичним ігровим процесом, готовий до вдосконалення та розширення у рамках самостійної роботи.

Успішне засвоєння матеріалів та виконання лабораторної роботи сприяє формування у студента наступних соціальних навичок та вмінь: вміння дотримуватися регламентних рамок, бажання постійно вчитися, оцінювати строки проведення та виконання роботи. Форми прояву: бажання пробувати щось нове, прагнення вирішувати поточні проблеми самостійно, йти на розумний ризик при прийнятті рішень, елементи самоконтролю і самооцінки при виконанні та оцінки роботи, вміння планувати свій час.

5.1 **Завдання до лабораторної роботи**

1. Обрати папку для проекту на веб-сервері та створити необхідну файлову структуру.
2. Розмістити усі необхідні ресурси у відповідних папках проекту.
3. Створити файл *index.html*, що являється точкою входу до ігрового додатку, та підключити в ньому фреймворк Phaser.

4. Створити головний JS-файл *main.js* та окремі файли ігрових станів «Меню», «Гра» та «Кінець гри».
5. В файлі main.js створити основний ігровий об’єкт та організувати перехід до першого ігрового стану («Меню»).
6. В стані «Меню»:
   - виконати завантаження усіх необхідних ресурсів;
   - відобразити стартовий екран із заставкою;
   - організувати перехід до стану «Гра».
7. В стані «Гра»:
   - створити головного персонажа та розробити код для його переміщення по ігровому полю за допомогою клавіатури;
   - забезпечити виведення на екран поточних значень ігрових параметрів (кількості предметів та швидкості);
   - забезпечити контроль зіткнення персонажу із краями ігрового поля («стінами») та ігровими предметами;
   - реалізувати процес нарощування швидкості відповідно до кількості предметів, що були зібрані;
   - реалізувати контроль умов закінчення гри (зіткнення із стінами або із собою) та перехід до стану «Кінець гри».
8. В стані «Кінець гри»:
   - забезпечити відображення підсумкового рахунку гри;
   - перехід до стану «Гра» для повторного запуску ігрового процесу.

5.2 Підготовка до лабораторної роботи №1

Підготовка до виконання лабораторної робити № 1 включає етапи:

1. Ознайомитися із головними принципами організації ігрового процесу в класичній грі «Змійка».
2. Ознайомитись з методичними вказівками до виконання лабораторної роботи (дивись 5.4).

5.3 Контрольні питання і попередні матеріали для допуску до лабораторної роботи № 1

5.3.1 Контрольні питання

1. З яких елементів складається стек технологій HTML5?
2. Наведіть основні загальні переваги і недоліки HTML5.

3. Наведіть основні переваги HTML5 із точки зору створення ігрових додатків.
4. Вкажіть основні недоліки HTML5 із точки зору створення ігрових додатків.
5. Опишіть способи розміщення ігрових HTML5-додатків.
6. Якими способами можна перетворити HTML5-додаток на додаток для мобільних операційних систем?
7. Які інструменти HTML5-to-app ви знаєте, вкажіть їх переваги та недоліки.
8. Наведіть архітектуру типового HTML5-додатку.
9. Які основні елементи входять до типового HTML5-додатку?
10. Яку роль виконують фреймворки в процесі розробки HTML5-додатків?
11. Назвіть найпопулярніші фреймворки для створення HTML5-додатків, порівняйте їх між собою.

5.3.2 **Попередні матеріали**

1. Спрайт головного персонажу (один сегмент тіла змійки).
2. Спрайт предмету для збирання.
3. Зображення для стартового екрану.
4. Зображення для кінцевого екрану.

5.4 **Методичні вказівки до виконання лабораторної роботи № 1**

5.4.1 **Створення проекту**

Розглянемо процес розробки ігрового додатку в Phaser на прикладі класичної гри «змійка» (snake). Це дозволить нам ознайомитися із типовими складовими гри – спрайтами, ігровими станами, попереднім завантаженням ресурсів, методами ініціалізації гри та основним ігровим циклом.

Гра на Phaser представляє собою набор файлів, орієнтовна організація яких наведена на рис. 5.1.

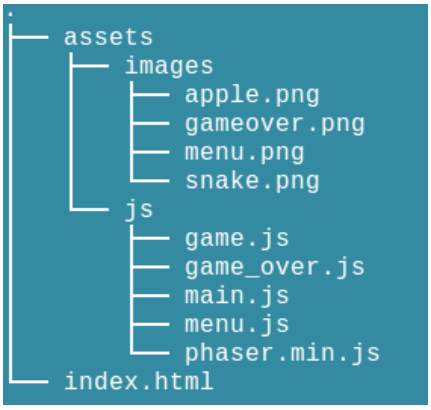

Рисунок 5.1 – Орієнтована файлова структура проекту на Phaser

Тобто для запуска гри достатньо скопіювати файли проекту до папки на веб-сервері та відкрити файл *index.html*. Він є точкою входу та має такий зміст:

```
<!doctype html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>Snake</title>
    <script src="assets/js/phaser.min.js"></script>
    <script src="assets/js/menu.js"></script>
    <script src="assets/js/game.js"></script>
    <script src="assets/js/game_over.js"></script>
    <script src="assets/js/main.js"></script>
</head>
<body>

</body>
</html>
```

Для створення гри нам знадобиться такі зображення:

- спрайт одного елементу тіла змійки (рис. 5.2а);
- спрайт предмету (рис. 5.2б);
- зображення для початкового екрану (рис. 5.2в);
- зображення для завершального екрану (рис. 5.2г).

а)

б)

в) г)

Рисунок 5.2 – Необхідні графічні елементи (спрайти) гри

Розіб’ємо процес роботи над грою на декілька етапів та розглянемо кожен із них більш докладно. Треб зазначити, що після виконання кожного етапу можна запустити гру та оцінити її поточну готовність.

5.4.2 **Організація ігрового процесу**

У Phaser основним елементом організації ігрового процесу є стани (states). Кожен зі станів представляє окрему частину гри. Тобто в нашій «змійці» можна виділити такі основні стани:

- *стан «Меню» (“Menu”)* – обробляється у menu.js та реалізує лише відображення стартового зображення. Коли гравець клікає на ньому мишею, виконується перехід до стану «Гра»;
- *стан «Гра» (“Game”)* – обробляється у game.js та представляє ігровий процес. Гравець керує змійкою та збирає предмети. Коли він помиляється, змійка гине та виконується перехід то стану «Кінець гри»;
- *стан «Кінець гри» (“Game Over”)* – обробляється game_over.js, відображає завершальне зображення та кількість зароблених балів (назбираних предметів). За кліком по картинці гра перезапускається – виконується перехід до стану «гра».

Файл *main.js* є основним файлом проекту, де створюється екземпляр об’єкту гри та додається стан «Меню».

5.4.3 **Завантаження ресурсів**

У файлі *index.html* до проекту була додана бібліотека Phaser, що дозволяє отримати доступ до глобального об’єкту Phaser. Через нього

отримується доступ до усіх методів фреймворку та створюються ігрові об'єкти.

Тепер через Phaser можна створити новий екземпляр ігрового об'єкту, що буде містити саму гру та дозволить додати до неї необхідні стани. Отже, файл main.js набуває такого вигляду:

```
var game;

// Створюємо екземпляр нової гри із розмірами 600 на 450
game = new Phaser.Game(600, 450, Phaser.AUTO, '');

// Першим параметром є назва стану
// Другим параметром є об'єкт, що містить необхідні функціональні методи
game.state.add('Menu', Menu);

// Перехід до стану
game.state.start('Menu');
```

Далі необхідно виконати ініціалізацію об'єкту *Menu*, що керує станом «menu». В файлі *menu.js* об'явимо новий об'єкт та додамо необхідні функції. При запуску стану першою викликається функція *preload*, яка завантажує усі необхідні ресурси. Після цього викликається функція *create* – вона ініціалізує ігрове поле та розміщує на ньому те, що необхідно. Отже, файл *menu.js* набуває такого вигляду:

```
var Menu = {

  preload : function() {
    // Завантажуємо зображення, на основі яких потім будуть створені спрайти
    // Перший параметр - назва зображення для посилання на нього
    // Перший - шлях до зображення на диску
    game.load.image('menu', './assets/images/menu.png');
  },

  create: function () {
    // Додаємо спрайт, який буде логотипом гри
    // Параметри - X, Y та назва (яка була призначена раніше)
    this.add.sprite(0, 0, 'menu');
  }

};
```

5.4.4 Створення персонажу

Як вже відмічалося вище, стан «*Game*» втілює основний ігровий процес. Тобто у нашому випадку це саме те місце, де потрібно створювати персонаж та предмети, а потім програмувати їх поведінку.

Але перед тим треба переробити декілька попередніх кроків. Спочатку треба зареєструвати стан «Game» в файлі *main.js*:

```
...
// Додаємо стан "Гра"
game.state.add('Game', Game);
...
```

Потім треба реалізувати перехід до стану «Гра» із стану «Меню» за кліком по зображенню. Для цього замінемо спрайт на кнопку – це буде майже те саме, але з тією відмінністю, що необхідно буде задати функцію-обробник для кнопки. Отже із цими змінами файл *menu.js* набуває вигляду:

```
var Menu = {

  ...

  create: function () {
    // Кнопка, за кліком по якій розпочинається гра.
    this.add.button(0, 0, 'menu', this.startGame, this);
  },

  startGame: function () {
    // Обробник кнопки, що переключається до стану "Гра"
    this.state.start('Game');
  }

};
```

Тепер можна перейти безпосередньо до малювання персонажу. Файл game.js має аналогічну структуру, але є більш наповненим. Усі основні моменти пояснюються в коментарях до коду:

```
var snake, apple, squareSize, score, speed,
    updateDelay, direction, new_direction,
    addNew, cursors, scoreTextValue, speedTextValue,
    textStyle_Key, textStyle_Value;

var Game = {

  preload : function() {
    // Завантаження ресурсів.
    // Два квадрати - один елемент "тіла" змійки та предмет
    game.load.image('snake', './assets/images/snake.png');
    game.load.image('apple', './assets/images/apple.png');
  },

  create : function() {
    // Ініціалізація глобальних змінних
    snake = [];          // Стек, що містить елементи тіла змійки
    apple = {};          // Об'єкт, що представляє предмет
    squareSize = 15;     // Сторона квадрата (зображення - 15 на 15 пікселів).
    score = 0;           // Ігровий рахунок
    speed = 0;           // Швидкість гри
    updateDelay = 0;        // Змінна для управління швидкістю оновлення
    direction = 'right';    // Напрямок змійки
    new_direction = null;   // Буфер для зберігання нового напряму
    addNew = false;         // Прапор підібраного предмету

    // Phaser-контролер клавіатури
    cursors = game.input.keyboard.createCursorKeys();

    game.stage.backgroundColor = '#061f27'; // Колір фону

    // Генерація початкового стеку тіла змійки, длина складає 10 елементів
    // Початок із позиції X=150 Y=150 та збільшення X у кожній ітерації
```

```
    for(var i = 0; i < 10; i++){
      // Параметри - X, Y, картинка
      snake[i] = game.add.sprite(150+i*squareSize, 150, 'snake');
    }

    // Генерація першого предмету
    this.generateApple();

    // Додавання тексту до верхньої частини ігрового поля
    textStyle_Key = {font:"bold 14px sans-serif",fill:"#46c0f9",
                     align:"center" };
    textStyle_Value = {font:"bold 18px sans-serif",fill:"#fff",
                     align:"center" };

    // Рахунок предметів
    game.add.text(30, 20, "SCORE", textStyle_Key);
    scoreTextValue = game.add.text(90, 18, score.toString(),
                                   textStyle_Value);

    // Швидкість
    game.add.text(500, 20, "SPEED", textStyle_Key);
    speedTextValue = game.add.text(558, 18, speed.toString(),
                                   textStyle_Value);
  },

  update: function() {
    // Викликається приблизно 60 разів на секунду для оновлення ігрового поля
    // Поки залишимо її порожньою
  },

  generateApple: function(){
    // Обираємо випадкове місце на ігровій сітці
    // X між 0 та 585 (39*15)
    // Y між 0 та 435 (29*15)
    var randomX = Math.floor(Math.random() * 40 ) * squareSize,
        randomY = Math.floor(Math.random() * 30 ) * squareSize;

    // Додаємо новий предмет
    apple = game.add.sprite(randomX, randomY, 'apple');
  }

};
```

5.4.5 **Переміщення та керування персонажем**

Для реалізації переміщення змійки необхідно модифікувати функцію *update()* в *game.js*. Перш за все необхідно створити обробники подій, що будуть контролювати напрям переміщення змійки за допомогою клавіш-стрілок на клавіатурі.

Реалізація ж переміщення набуває додаткової складності за тієї причини, що функція update має дуже високу частоту викликів. І якщо ми будемо переміщати змійку на кожному виклику функції *update()*, то отримаємо дуже швидкий, але повністю неконтрольований ігровий процес. Для запобігання цього використаємо конструкцію, що виділяє кожний 10-й виклик функції за допомогою лічильника *updateDelay*.

Тобто якщо черговий виклик *update()* є кратним 10, то ми видаляємо останній елемент змійки (він є першим у стеку), присвоюємо йому нові координати відповідно до напрямку руху та розміщуємо його перед головою змійки (на вершині стеку). Отже, відповідний код буде мати такий вигляд:

```
update: function() {

  // Обробка натискання кнопок стрілок
  // із урахування заборонених комбінацій, що вб'ють персонажа
  if (cursors.right.isDown && direction!='left') {
      new_direction = 'right';
  }
  else if (cursors.left.isDown && direction!='right') {
      new_direction = 'left';
  }
  else if (cursors.up.isDown && direction!='down') {
      new_direction = 'up';
  }
  else if (cursors.down.isDown && direction!='up') {
      new_direction = 'down';
  }

  // Формула для розрахування швидкості на основі здобутків
  // Чи більше здобутки, ти швидше гра (збільшуються кожні 5 предметів)
  speed = Math.min(10, Math.floor(score/5));
  // Оновлення швидкості на екрані
  speedTextValue.text = '' + speed;

  // Так як функція update в Phaser викликається 60 разі на секунду,
  // то необхідно уповільнити виклик щоб процес був іграбельним

  // Збільшуємо лічильник кожного виклику update
  updateDelay++;

  // Обробка виконується лише якщо лічильник ділиться на (10 - speed) без
залишку
  // збільшення значення speed робить виклики більш частими і пришвидшує гру
  if (updateDelay % (10 - speed) == 0) {

    // Переміщення змійки
    var firstCell = snake[snake.length - 1],
      lastCell = snake.shift(),
      oldLastCellx = lastCell.x,
      oldLastCelly = lastCell.y;

    // Якщо гравцем вибраний новий напрям, то робимо його поточним напрямом
змійки
    if(new_direction){
      direction = new_direction;
      new_direction = null;
    }

      // Зміна координат останньої клітинки відповідно до голови змійки
      // з урахуванням напрямку руху
      if(direction == 'right'){
          lastCell.x = firstCell.x + 15;
          lastCell.y = firstCell.y;
      }
      else if(direction == 'left'){
```

```
            lastCell.x = firstCell.x - 15;
            lastCell.y = firstCell.y;
        }
        else if(direction == 'up'){
            lastCell.x = firstCell.x;
            lastCell.y = firstCell.y - 15;
        }
        else if(direction == 'down'){
            lastCell.x = firstCell.x;
            lastCell.y = firstCell.y + 15;
        }
        // Переміщаємо останню клітинку до голови стеку
        // та відмічаємо її як першу
        snake.push(lastCell);
        firstCell = lastCell;
    }
}
```

5.4.6 **Фіксація зіткнень**

Для організації повноцінного ігрового процесу необхідно додати фіксацію зіткнень змійки із краями ігрового поля та нею самою. Крім того, необхідно реалізувати «поїдання» предметів та відповідне збільшення довжини змійки.

Зазвичай фіксація зіткнень у Phaser реалізується за допомогою одного із фізичних движків, що присутні у фреймворку. Але в такий простій грі як змійка можна обмежитися порівнянням координат та не задіювати жодного движка зіткнень.

В функції *update()* після реалізації переміщення змійки додамо виклик декількох методів:

- *appleCollision* фіксує та обробляє подію зіткнення (підбирання) предмета;
- *selfCollision* обробляє зіткнення змійки із нею самою;
- *wallCollision* обробляє зіткнення із краями ігрового поля стінами.

Отже, код набуває такого вигляду:

```
update: function() {

    // Переміщення змійки
    // ...

    // Збільшуємо довжину змійки якщо вона з'їла предмет
    // Створюємо блок в кінці змійки на позиції попереднього останнього блоку
    if(addNew){
        snake.unshift(game.add.sprite(oldLastCellx, oldLastCelly, 'snake'));
        addNew = false;
    }

    // Перевірка зіткнення із предметом
    this.appleCollision();
```

```
  // Перевірка зіткнення із собою, параметр - голова змійки
  this.selfCollision(firstCell);

  // Перевірка зіткнення із стіною, параметр - голова змійки
  this.wallCollision(firstCell);
}

appleCollision: function() {
  // Перевіряємо, чи перекриває будь-який блок змійки предмет
  // Це необхідно для випадку коли предмет з'являється в середині змійки
  for(var i = 0; i < snake.length; i++){
    if(snake[i].x == apple.x && snake[i].y == apple.y){

      // При наступному переміщенні змійки до неї буде додано новий блок
      addNew = true;

      // Видаляємо старий предмет...
      apple.destroy();

      // ... та генеруємо новий
      this.generateApple();

      // Збільшуємо рахунок
      score++;

      // Оновлюємо надпис із рахунком
      scoreTextValue.text = score.toString();
    }
  }
},

selfCollision: function(head) {
  // Перевіряємо, чи перетинає голова змійки будь-який інший елемент
  for(var i = 0; i < snake.length - 1; i++){
    if(head.x == snake[i].x && head.y == snake[i].y){
      // Якщо так, то переходимо до завершення гри
      game.state.start('Game_Over');
    }
  }
},

wallCollision: function(head) {
  // Перевіряємо, чи знаходиться голова змійки у межах ігрового поля?
  if(head.x >= 600 || head.x < 0 || head.y >= 450 || head.y < 0){
      // Якщо ні, то змійка зіткнулася із краєм та треба завершувати гру
      game.state.start('Game_Over');
  }
}
```

Відповідно до наведеного коду, коли змійка зіткається із предметом, ми збільшуємо рахунок та довжину змійки. Але коли відбувається зіткнення із стінами або собою, то необхідно завершити гру. Для цього потрібно ввести новий стан «Кінець гри» (Game_Over). Спочатку його потрібно зареєструвати в файлі *main.js*.

```
game.state.add('Game_Over', Game_Over);
```

а потім створити відповідний файл і код:

```
var Game_Over = {
```

```
  preload : function() {
    // Завантажуємо необхідне зображення
    game.load.image('gameover', './assets/images/gameover.png');
  },

  create : function() {
    // Створюємо кнопку на основі зображення для перезапуску гри
    this.add.button(0, 0, 'gameover', this.startGame, this);

    // Додаємо текст про результат останньої гри
    game.add.text(235, 350, "LAST SCORE", {
                font: "bold 16px sans-serif", fill: "#46c0f9", align:
"center"});
    game.add.text(350, 348, score.toString(), {
                 font: "bold 20px sans-serif", fill: "#fff", align: "center"
});
  },

  startGame: function () {
    // Переключаємося до стану "Гра"
    this.state.start('Game');
  }
};
```

На рис. 5.3 наведені стани «Гра» та «Кінець гри». Отже, таким чином, гра є завершеною та її можна протестувати.

В майбутньому гра може бути вдосконалена та покращена різними шляхами. Наприклад, можна створити різні види предметів: перші мають звичайну поведінку, другі – збільшують рахунок, але не довжину змійки, треті – навіть скорочують її тощо. Або додати перепони-стіни посеред карти, або щось таке інше. Метою розробки самі цієї простої гри було ознайомлення із основними принципами побудування ігрових додатків на Phaser і цю мету було досягнуто.

Рисунок 5.3 – Стани «Гра» та «Кінець гри»

Очікувані практичні результати роботи:

- повністю закінчений ігровий HTML5-додаток із класичним ігровим процесом, готовий до вдосконалення та розширення у рамках самостійної роботи.

6 Лабораторна робота № 2. Створення шаблону додатку на Phaser із реалізацією типових ігрових станів

Лабораторна робота орієнтована на оволодіння студентами навичок із організації в ігровому додатку системи станів, зв’язаних між собою. В роботі докладно розглянутий процес створення базового шаблону ігрового додатку, який в подальшому часі може бути використаний для побудови HTML5-додатку любого жанру.

Мета лабораторної роботи:

- докладно ознайомитися із системою типових ігрових станів, які є основою будь-якого додатку на Phaser;
- ознайомитися із реалізацією попереднього завантаження ресурсів із підтримкою прогрес-бару;
- розглянути механізм та особливості використання в ігровому додатку користувацьких шрифтів;
- ознайомитися із завантаженням та керування звуковими файлами і ігровому додатку;
- розглянути особливості використання часових затримок та їх налаштувань відповідно до особливостей гри;
- розглянути способи організацій головного ігрового меню та створення універсальних засобів для розміщення меню в будь-якому стані додатку;
- ознайомитися із можливими варіантами повторного використання загальної таблиці стилів для однотипних ігрових елементів;
- розглянути особливості використання механізму домішок (mixins) для запобігання багаторазового дублювання коду;
- ознайомитися із створенням глобальних ігрових параметрів для використання в будь-якому стані додатку.

Відповідно до завдання студент створює шаблон ігрового додатку на Phaser, який містить:

- підтримку усіх типових ігрових режимів (Заставка, Головне меню, Гра, Кінець гри, Налаштування, Автори) та переходи між ними;
- попереднє завантаження усіх ігрових ресурсів в режимі заставки із прогрес-баром;
- універсальну домішку, за допомогою якої в будь якому режимі додатку може бути створене меню (підтримується позиціонування та різні стилі пунктів);
- можливість налаштування ігрових параметрів гри через відповідне меню (для прикладу реалізований контроль звукових параметрів).

Отже, у разі успішного виконання лабораторної роботи студент отримає як навички у розбудові ігрових проектів та реалізації типових ігрових станів, так й готовий універсальний шаблон ігрового додатку на Phaser, який може використати як основу для роботи над грою будь-якого жанру. Саме цей шаблон буде використаний у ЛР №3.

Успішне засвоєння матеріалів та виконання лабораторної роботи сприяє формування у студента наступних соціальних навичок та вмінь: оцінювати строки проведення та виконання роботи, обґрунтувати той чи інший спосіб діяльності при виконанні практичних завдань, бажання постійно вчитися, вміння дотримуватися задекларованих регламентних рамок, вільність оволодіння сучасними технологіями та їх використання в усіх сферах життя. Форми прояву: прагнення вирішувати поточні проблеми самостійно, бажання пробувати щось нове, йти на розумний ризик при прийнятті рішень, елементи самоконтролю і самооцінки при виконанні роботи, вміння планувати свій час.

6.1 **Завдання до лабораторної роботи № 2**

1. Створити структуру папок та HTML-файл
2. В основному файлі *main.js*:
   - створити основний ігровий об’єкт Phaser.Game;
   - завантажити ресурси для створення стану Splash (екрану-заставки);
   - організувати перехід до стану Splash;
3. Створити файл splash.js для стану *Splash* (екрана-заставки), в якому:
   - створити спрайт із прогрес-баром та інформаційні надписи для супроводу завантаження;

- реалізувати попереднє завантаження усіх ігрових ресурсів в окремих функціях;
- завантажити користувацький шрифт;
- створити усі інші ігрові стани;
- запустити звукове супроводження.

4. Створити файл *gamemenu.js* для стану GameMenu (головне меню), в якому:
   - створити функцію додавання пунктів меню для переходу до інших режимів з можливістю встановлення необхідних обробників;
   - реалізувати підтримку таблиці стилів для керування виглядом однотипних елементів;
   - оптимізувати код та реалізувати функцію додавання пункту меню за допомогою домішок (mixins);
   - реалізувати позиціонування меню з допомогою зовнішнього об’єкту.
5. Створити файл *options.js* для стану Options (Налаштування), в якому:
   - створити пункт меню для повернення назад до головного меню;
   - реалізувати зміну глобальних параметрів гри, що відповідають за звукове оформлення.
6. Створити файл *gameover.js* для стану GameOver (Гру завершено) аналогічно до стану налаштувань;
7. Створити файл *credits.js* для стану Credits (Автори) аналогічно до стану налаштувань.

6.2 Підготовка до лабораторної роботи № 2

При підготовці до виконання лабораторної роботи необхідно:

1. Ознайомитися із основами систем и ігрових станів в Phaser.
2. Ознайомитися із методичними вказівками щодо виконання лабораторної роботи (див. 6.4).

6.3 Контрольні питання і попередні матеріали для допуску до лабораторної роботи № 2

6.3.1 Контрольні питання

1. Яким чином в Phaser створюється гра?

2. Як в Phaser контролюються розміри ігрового світу?
3. Який об’єкт керує камерою та які основні налаштування він має?
4. Які основні стани є у ігрового об’єкту, яке їх призначення?
5. Як реалізований кеш, як поводитися із об’єктами в кеші?
6. Як реалізований процес завантаження зовнішніх ігрових ресурсів?
7. Які типи даних підтримує завантажувач в Phaser?
8. Як реалізоване масштабування ігрових елементів та ігрової карти?
9. З чого складається подієва модель, як її використати?
10. Як відбувається робота із доповненнями в Phaser?
11. В чому призначення та особливості групування об’єктів в Phaser?
12. Як реалізована система користувацького введення та керування грою?

6.3.2 **Попередні матеріали**

Для допуску к виконанню лабораторної роботи необхідно представити:

1. Фонове зображення екрану-заставки та логотип.
2. Спрайт прогрес-бару.
3. Фон головного меню.
4. Фон екрану налаштувань.
5. Фон екрану завершення гри.
6. Аудіо-файл для фонової музики.
7. Аудіо-файл для прикладу звукового ефекту.

6.4 **Методичні вказівки до виконання лабораторної роботи № 2**

6.4.1 **Структура папок та HTML-файл**

В лабораторній роботі пропонується використовувати таку структуру папок:

`game/`	Коренева папка гри (index.html та main.js)
`  assets/`	Ігрові ресурси
`    bgm/`	Звукові файли
`    fonts/`	Шрифти
`    images/`	Зображення
`    style`	CSS-файли
`  states/`	Скрипти станів

`vendor/` Фреймворк та допоміжні скрипти

При цьому HTML-файл гри буде мати такий вигляд:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
</head>
<body>
    <main id="game"></main>
    <script src="vendor/phaser.js"></script>
    <script src="main.js"></script>
</body>
</html>
```

Зверніть увагу, що в HTML-файлі підключаються лише два скрипти – сам движок Phaser (*phaser.js*) та основний скрипт (*main.js*), який вже в свою чергу підключає усі інші файли. Такий підхід дозволяє трохи пришвидшити завантаження. Ще більш важливим є те, що користувач не буде довго чекати завантаження екрана-заставки, сидячи перед чорним екраном. Тобто перший користувацький досвід після запуску гри буде позитивним.

6.4.2 Основний скрипт та ресурси для екрану-заставки

Отже, для відображення екрана-заставки знадобляться такі ресурси:

- фонова картинка розміром 800 на 600 пікселів;
- логотип гри, проекту, компанії, розробника, тощо;
- прогрес-бар, у ролі якого можна використати звичайний прямокутник із заливкою бажаним кольором.

Усі ці ресурси необхідно помістити до папки *game/assets/images*.

Далі перейдемо до файлу main.js, в якому буде створюватися екземпляр гри та завантажуватися усе необхідне для створення екрану заставки. Завдяки системі стані Phaser ми зможемо переключитися до екрану-заставки одразу після того, як усі ресурсу буде завантажено:

```
var game = new Phaser.Game(800, 600, Phaser.AUTO, 'game'),
    Main = function () {};

Main.prototype = {
  preload: function () {
    game.load.image('stars',    'assets/images/stars.jpg');
    game.load.image('loading',  'assets/images/loading.png');
    game.load.image('brand',    'assets/images/logo.png');
    game.load.script('splash',  'states/Splash.js');
  },

  create: function () {
    game.state.add('Splash', Splash);
    game.state.start('Splash');
```

```
  }
};

game.state.add('Main', Main);
game.state.start('Main');
```

6.4.3 Створення скрипту для екрану-заставки

Створимо файл *assets/states/splash.js*, який буде завантажувати усі інші ресурси ігрового додатку. Отже, для більш логічно організації коду, створимо окремі функцій для кожного типу ресурсів:

```
splash.prototype = {

  loadScripts: function () {
  },

  loadBgm: function () {
  },

  loadImages: function () {
  },

  loadFonts: function () {
  },

  // Функція передзавантаження просто викликає функції, описані вище
  preload: function () {
    this.loadScripts();
    this.loadImages();
    this.loadFonts();
    this.loadBgm();
  },
```

Добре побудоване завантаження завжди повинне мати прогрес-бар – щоб користувач мав інформаціє про проходження процесу. Додання прогрес-бару є досить простим:

```
// Створюємо спрайт із прогрес-баром
var loadingBar = game.add.sprite(game.world.centerX, 400, "loading");
// Сповіщаємо Phaser, що саме цей спрайт необхідно використати як прогрес-бар
this.load.setPreloadSprite(loadingBar);
```

Крім завантаження прогрес-бару в функції *preload()* необхідно створити об’єкти на основі усіх ресурсів, що були завантажені раніше:

```
preload: function () {
  var myLogo, loadingBar, status;

  // Створення фонового зображення
  game.add.sprite(0, 0, 'stars');

  // Створення логотипу
  myLogo = game.add.sprite(game.world.centerX, 100, 'brand');
  myLogo.anchor.setTo(0.5);
  myLogo.scale.setTo(0.5);

  // Надпис про завантаження
  status = game.add.text(game.world.centerX, 380, 'Йде завантаження...',
                         {fill: 'white'});
```

```
  status.anchor.setTo(0.5);

  // Створення прогрес-бару
  loadingBar = game.add.sprite(game.world.centerX, 400, "loading");
  loadingBar.anchor.setTo(0.5);
  this.load.setPreloadSprite(loadingBar);

  ...
}
```

В наведеному коді дуже часто зустрічається виклик *object.anchor.setTo(0.5)*, що забезпечує вирівнювання елементу по центру відповідно до батьківського контейнеру. Метод викликається багато разів для всіх візуальних елементів, тому доцільно створити допоміжний скрипт-файл *lib/utils.js* та помістити до нього такий код:

```
var utils = {
  centerGameObjects: function (objects) {
    objects.forEach(function (object) {
      object.anchor.setTo(0.5);
    })
  }
};
```

та додати до функції *preload()* файлу *main.js* підключення *utils.js*:

```
preload: function () {
  ...
  game.load.script('utils',   'lib/utils.js');
},
```

Наступним покращенням в організації коду буде винесення створення спрайтів до окремої функції *init()*:

```
init: function() {
  this.loadingBar = game.make.sprite(game.world.centerX-(387/2), 400, "loading");
  this.logo = game.make.sprite(game.world.centerX, 200, 'brand');
  this.status = game.make.text(game.world.centerX, 380, 'Loading...',{fill: 'white'});
  utils.centerGameObjects([this.logo, this.status]);
},

preload: function () {
  game.add.sprite(0, 0, 'stars');
  game.add.existing(this.logo).scale.setTo(0.5);
  game.add.existing(this.loadingBar);
  game.add.existing(this.status);
  this.load.setPreloadSprite(this.loadingBar);

  this.loadScripts();
  this.loadImages();
  this.loadFonts();
  this.loadBgm();
},
```

Зверніть увагу, що в ній використаний *game.**make**.sprite* замість *game.**add**.sprite* – це дозволяє спочатку створити необхідні спрайти, а потім завантажити їх в функції *preload()* за допомогою *game.add.existing*. Крім того,

в коді продемостровано використання «ланцюжків викликів» на прикладі масштабування спрайту із логотипом.

6.4.4 Завантаження ресурсів та користувацького шрифту

Наповнимо функції, що були створені раніше для завантаження кожного типу ресурсів:

```
loadScripts: function () {
  // Подключаємо WebFont (https://github.com/typekit/webfontloader)
  game.load.script('WebFont', 'vendor/webfontloader.js');
  // Скрипти основних станів (режимів) гри
  game.load.script('gamemenu','states/gamemenu.js');
  game.load.script('thegame', 'states/thegame.js');
  game.load.script('gameover','states/gameover.js');
  game.load.script('credits', 'states/credits.js');
  game.load.script('options', 'states/options.js');
},

loadBgm: function () {
  // Необхідні звукові файли
  game.load.audio('dangerous', 'assets/bgm/Dangerous.mp3');
  game.load.audio('exit', 'assets/bgm/Exit the Premises.mp3');
},

loadImages: function () {
  // Фонові зображення для станів гри
  game.load.image('menu-bg', 'assets/images/menu-bg.jpg');
  game.load.image('options-bg', 'assets/images/options-bg.jpg');
  game.load.image('gameover-bg', 'assets/images/gameover-bg.jpg');
},
```

Далі розглянемо завантаження користувацьких шрифтів. Phaser добре працює із шрифтовим завантажувачем від Google. Але для того, що не залежати від зовнішніх серверів, доцільно буде завантажити файл із шрифтом та зберегти його як частину ігрових ресурсів. Отже, додавання шрифту включає такі етапи:

1. Завантаження файлу шрифту з відкритих джерел (наприклад, http://www.dafont.com або https://www.fonts-online.ru).
2. Розміщення файлу «*.ttf» до папки *game/assets/fonts* (для прикладу оберемо *BertramModern.ttf*).
3. Створення css-файлу для визначення шрифту:

```
@font-face {
  font-family: 'BertramModern';
  src: url('../fonts/BertramModern.otf');
}
```

Далі скористаємось *WebFontLoader* для завантаження шрифта. Це дозволить не додавати посилання на шрифт до HTML-файлу, а значить користувач буде бачити екран-заставку підчас завантаження. Отже, функція *loadFonts()* набуває вигляду:

```
loadFonts: function () {
  WebFontConfig = {
    custom: {
      families: ['BertramModern'],
      urls: ['assets/style/BertramModern.css']
    }
  }
}
```

6.4.5 Створення решти ігрових станів

Перед створення решти ігрових станів, додамо до стану *Splash* функцію *create()*, яка змінює надпис з «Завантажуємо…» на «Готово!» та додає 3-ти секундну затримку перед переходом до наступного стану:

```
create: function() {
  this.status.setText('Ready!');

  setTimeout(function () {
    // We will load the main menu here
  }, 3000);
}
```

За замовчуванням Phaser призупиняє гру у випадку втрати фокусу. Тобто якщо користувач бачить екран-заставку і під час цього переключається на іншу вкладку браузера, то стан гри не зміниться. Але в той же час ігрові ресурси будуть завантажені, що й буде підтверджено надписом «Готово!». І коли користувач переключиться назад до гри, стан зміниться на наступний (головне меню).

Якщо ж користувач не буде нікуди переключатися, то затримка в 3 секунди буде просто зайвим очікуванням. Отже під час тестування гри спробувати підібрати розмір затримки або навіть повністю її видалити.

Далі треба додати усі інші стани, що будуть присутні в ігровому додатку:

```
game.state.add("GameMenu",GameMenu);
game.state.add("Game",Game);
game.state.add("GameOver",GameOver);
game.state.add("Credits",Credits);
game.state.add("Options",Options);
```

Для кожного стану створюється відповідний файл. Наприклад для стану «Credits» створюємо файл *game/states/Credits.js* із таким змістом:

```
var Credits = function () {};
```

Аналогічно для всіх інших станів створюються файли-заглушки, які будуть заповнені кодом пізніше.

6.4.6 Звук та підсумковий код екрана-заставки

Останнім кроком в оформленні екрана-заставки є додання музики, що засобами Phaser робиться дуже просто:

```
music = game.add.audio('dangerous');
music.loop = true;
music.play();
```

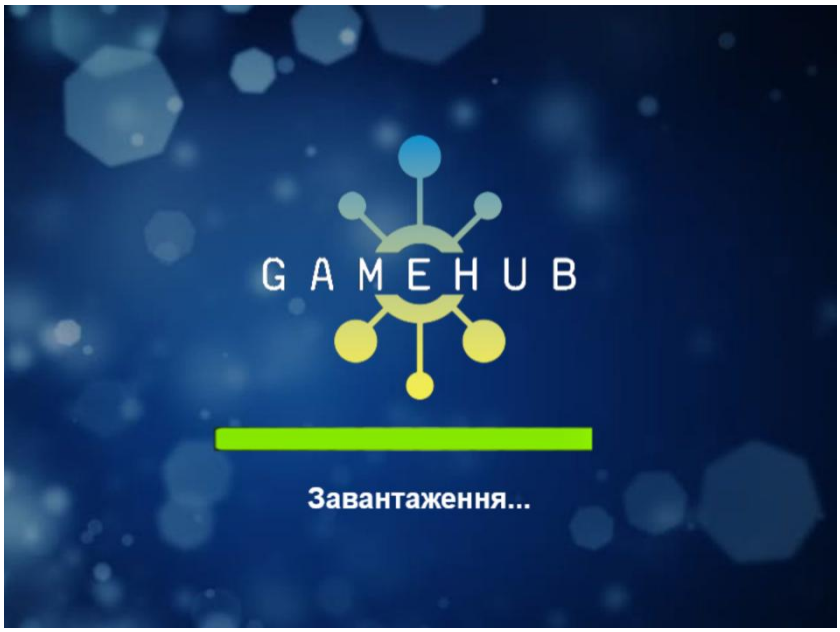

Рисунок 6.1 – Вигляд екрану-заставки

Але треба зазначити, що більш правильним рішення є визначення змінної *music* як глобальної. Програвання музики зберігається при переході між станами, і тому об’єкт, що забезпечує це, не може належати жодному стану. Крім того, доцільним буде є визначення ще двох параметрів – *playSound* та *playMusic*. Вони дозволять керувати програванням звуків та музики та прив’язуватися до загальноігрових налаштувань.

Отже, робота над екраном-заставкою (рис. 6.1) є завершеною. Із урахуванням усіх покращень та доповнень файл *splash.js* набуває такого вигляду:

```
var Splash = function () {},
    playSound = true,
    playMusic = true,
    music;

Splash.prototype = {

  loadScripts: function () {
    game.load.script('WebFont', 'vendor/webfontloader.js');
    game.load.script('gamemenu','states/gamemenu.js');
    game.load.script('thegame', 'states/thegame.js');
    game.load.script('gameover','states/gameover.js');
    game.load.script('credits', 'states/credits.js');
    game.load.script('options', 'states/options.js');
  },
```

```
loadBgm: function () {
  game.load.audio('dangerous', 'assets/bgm/Dangerous.mp3');
  game.load.audio('exit', 'assets/bgm/Exit the Premises.mp3');
},

loadImages: function () {
  game.load.image('menu-bg', 'assets/images/menu-bg.jpg');
  game.load.image('options-bg', 'assets/images/options-bg.jpg');
  game.load.image('gameover-bg', 'assets/images/gameover-bg.jpg');
},

loadFonts: function () {
  WebFontConfig = {
    custom: {
      families: ['BertramModern'],
      urls: ['assets/style/BertramModern.css']
    }
  }
},

init: function () {
  this.loadingBar = game.make.sprite(game.world.centerX-(387/2), 400,
                    "loading");
  this.logo = game.make.sprite(game.world.centerX, 200, 'brand');
  this.status = game.make.text(game.world.centerX, 380, 'Завантаження...',
                {fill: 'white'});
  utils.centerGameObjects([this.logo, this.status]);
},

preload: function () {
  game.add.sprite(0, 0, 'stars');
  game.add.existing(this.logo).scale.setTo(0.5);
  game.add.existing(this.loadingBar);
  game.add.existing(this.status);
  this.load.setPreloadSprite(this.loadingBar);

  this.loadScripts();
  this.loadImages();
  this.loadFonts();
  this.loadBgm();
},

addGameStates: function () {
  game.state.add("GameMenu",GameMenu);
  game.state.add("Game",Game);
  game.state.add("GameOver",GameOver);
  game.state.add("Credits",Credits);
  game.state.add("Options",Options);
},

addGameMusic: function () {
  music = game.add.audio('dangerous');
  music.loop = true;
  music.play();
},

create: function() {
  this.status.setText('Готово!');
  this.addGameStates();
  this.addGameMusic();
```

```
    setTimeout(function () {
      //game.state.start("GameMenu");
    }, 3000);
  }
};
```

6.4.7 Прототипування екрану меню

Екран-заставка, який було розглянуто вище, посилається на стан *GameMenu*, отже створимо відповідний каркас:

```
var GameMenu = function() {};

GameMenu.prototype = {
  preload: function () {
  },

  create: function () {
  }
};
```

Далі необхідно додати фонове зображення та надпис-назву ігрового додатку, використавши користувацький шрифт, який було завантажено раніше:

```
var bg = game.add.sprite(0, 0, 'menu-bg'),
    // titleStyle використовує налаштування шрифта для встановлення
    // користувацького шрифта, який було завантажено раніше
    titleStyle = { font: 'bold 60pt BertramModern', fill: '#FDFFB5',
                   align: 'center'},
    text = game.add.text(game.world.centerX, 100, "Game Title", titleStyle);

text.setShadow(3, 3, 'rgba(0,0,0,0.5)', 5);
text.anchor.set(0.5);
```

Наведений код працює, але з точки зору якості має недолік, – в ньому декларуються змінні, які використовуються лише один раз. Але така поведінка набуває смислу, якщо використати модель з функцією *init()*, яка викликається ще до *preload()*:

```
init: function () {
  this.titleText = game.make.text(game.world.centerX, 100, "Game Title", {
    font: 'bold 60pt BertramModern',
    fill: '#FDFFB5',
    align: 'center'
  });
  this.titleText.setShadow(3, 3, 'rgba(0,0,0,0.5)', 5);
  this.titleText.anchor.set(0.5);
},
```

З урахуванням цього, *create()* набуває вигляду:

```
create: function () {
  game.add.sprite(0, 0, 'menu-bg');
  game.add.existing(this.titleText);
}
```

Але на цей час в поведінці додатку є ше один значний недолік – при втраті фокусу музика в грі переривається. При чому річ йде про втрату фокусу при переключенні на інший додаток, а не до іншої вкладки браузера. І якщо така поведінка в екрані-заставці була допустимою, то в режимі меню така пауза виглядає зайвою.

Вирішення є дуже простим завдяки глибині та універсальності фреймворку Phaser. Достатньо до функції *create()* додати

```
game.stage.disableVisibilityChange = true;
```

Але треба пам’ятати, що це налаштування є перехідним – тобто при переході до іншого стану, значення *true* збережеться. І один раз змінивши його, ми беремо на себе відповідальність про керування ним.

6.4.8 **Додавання пунктів меню**

Початковий код додавання пунктів є досить простим і майже не потребує коментарів:

```
// Створюємо стиль для пункту меню
var optionStyle = {font: '30pt BertramModern', fill: 'white',
                   align: 'left' };
// Додаємо текст
var txt = game.add.text(30, 280, 'Розпочати', optionStyle);
// Активуємо систему введення
txt.inputEnabled = true;
// Додаємо обробник натискання пункту меню
txt.events.onInputUp.add(function () { console.log('Натиснуто пункт меню!')
});
```

Зверніть увагу, що елемент не буде реагувати на жодні дії користувача, доки на цьому елементі ви не активуєте прослуховування подій введення за допомогою атрибута *inputEnabled*.

Для більшої переконливості пункту меню потрібно додати більше інтерактивності. Частіше за все це робиться шляхом реалізації подій «*Over*» та «*Out*» – тобто фіксації моментів, коли вказівник миші переміщається на об’єкт та покидає його. У якості наповнення події частіше за все використовується якась візуальна зміна об’єкту – кольору, наповнення, розміру тощо.

У нашому випадку обмежимось зміною кольору надпису, тому додамо ще два простих обробники:

```
txt.events.onInputOver.add(function (target) {
    target.fill = "#FEFFD5";
});

txt.events.onInputOut.add(function (target) {
    target.fill = "white";
});
```

Отже, все працює, але якщо спробувати додати ще один пункт меню, то доведеться повторити весь цей код ще раз із мінімальними змінами – бо оформлення кожного пункту меню має бути подібним до інших. А якщо потім ми побажаємо щось змінити (наприклад, колір), то міняти доведеться у декількох місцях, і це дуже незручно. Тобто, виникає питання про розбудову деякої «фабрики пунктів меню», яка б створювала однотипні пункти, які відрізнялися би лише надписами та обробниками дій, при цьому використовуючи єдину глобальну таблицю стилів для оформлення.

Якщо спробувати проаналізувати ці проблеми більш предметно, то ми отримаємо дві задачі для вирішення:

1. Реалізувати повторне використання стилю оформлення (шрифт, колір тощо).
2. Реалізувати універсальний засіб додавання елементів без постійного дублювання коду.

Розпочнемо з того, що створимо функцію *addMenuOption()* із таким кодом:

```
addMenuOption: function(text, callback) {
  var optionStyle = { font: '30pt BertramModern', fill: 'white',
              align: 'left', stroke: 'rgba(0,0,0,0)', srokeThickness: 4};
  var txt = game.add.text(30, (this.optionCount * 80) + 200, text,
optionStyle);
  var onOver = function (target) {
    target.fill = "#FEFFD5";
    target.stroke = "rgba(200,200,200,0.5)";
  };
  var onOut = function (target) {
    target.fill = "white";
    target.stroke = "rgba(0,0,0,0)";
  };
  txt.stroke = "rgba(0,0,0,0";
  txt.strokeThickness = 4;
  txt.inputEnabled = true;
  txt.events.onInputUp.add(callback);
  txt.events.onInputOver.add(onOver);
  txt.events.onInputOut.add(onOut);
  this.optionCount++;
}
```

Зверніть увагу, що нам треба знати – як позиціонувати черговий пункт меню по вертикалі. Для цього використаємо підрахунок пунктів меню у змінній *optionCount*, яку треба скинути в функції *init()*:

```
    this.optionCount = 1;
```

Отже тепер в нас є фабрика із двома параметрами – назвою пункту меню та функцією, яку є обробником події натискання. Тобто, для кожного пункту меню можна задати певну послідовність дій:

```
this.addMenuOption('Почати', function (target) {
  console.log('Натиснуто "Почати"!');
});

this.addMenuOption('Налаштування', function (target) {
  console.log('Натиснуто "налаштування"!');
});

this.addMenuOption('Автори', function (target) {
  console.log('Натиснуто "Автори"!');
});
```

В цьому коді *target* – це тільки що створений об’єкт, який передається у якості параметра.

6.4.9 **Таблиця стилів і підсумковий код головного меню**

Наступним покращенням буде формування глобальної таблиці стилів, в якому задаються налаштування для основних елементів гри. Це дозволить більш швидко та універсально керувати зовнішнім виглядом гри.

Отже, для підтримки таблиці стилів необхідно виконати такі дії:

1. Створити файл підтримки стилів *game/lib/style.js*.
2. Завантажити його в у відповідній функції екрану-заставки.
3. Створити об’єкт JavaScript, що буде містити пари «ключ-значення» для елементів таблиці стилів.
4. Використати стилі із таблиці за допомогою «.*setStyle*» замість декларування їх в коді меню.

Початковий вигляд файлу *style.js* буде таким:

```
var defaultColor = "white",
  highlightColor = "#FEFFD5";

navitem: {
  base: {
  },
  default: {
    fill: defaultColor,
    stroke: 'rgba(0,0,0,0)'
    font: '30pt BertramModern',
    align: 'left',
    strokeThickness: 4
  },
  hover: {
    fill: highlightColor,
    stroke: 'rgba(200,200,200,0.5)'
  }
}
```

Головним недоліком коду є те, що змінні *defaultColor* та *highlightColor* є глобальними, але використовуються лише в ініціалізації *navitem*. Для таких випадків в JavaScript існують анонимні функції-обгортки, які не мають власного імені та виконуються одразу ж після декларації. При цьому вони

мають доступ до глобальних елементів, тобто можуть їх ініціалізувати. Отже, реалізація таблиці стилів на основі функції-обгортки буде виглядати так:

```
// глобальна змінна, що буде доступна звідусіль
var style;

// анонимна функція-обертка
(function () {

  // ці змінні не будуть об'явлені ніде, окрім цієї функції
  var defaultColor = "white",
    highlightColor = "#FEFFD5";

  style = {
    header: {
      font: 'bold 60pt BertramModern',
      fill: defaultColor,
      align: 'center'
    },
    navitem: {
      base: {
        font: '30pt BertramModern',
        align: 'left',
        strokeThickness: 4
      },
      default: {
        fill: defaultColor,
        stroke: 'rgba(0,0,0,0)'
      },
      hover: {
        fill: highlightColor,
        stroke: 'rgba(200,200,200,0.5)'
      }
    }
  };
})();
```

Відповідно змінимо і функцію *addMenuOption()*:

```
addMenuOption: function(text, callback) {

  var txt = game.add.text(30, (this.optionCount * 80) + 200, text,
style.navitem.default);
  txt.inputEnabled = true;
  txt.events.onInputUp.add(callback);

  txt.events.onInputOver.add(function (target) {
    target.setStyle(style.navitem.hover);
  });

  txt.events.onInputOut.add(function (target) {
    target.setStyle(style.navitem.default);
  });

  this.optionCount++;
},
```

Але при цьому виникає проблема – виклик призначення стилів (*setStyle*) повністю затирає всі інші атрибути. Тому необхідно додати в кінці функції-обгортки код об’єднання, а не перезапису властивостей:

```
Object.assign(style.navitem.hover, style.navitem.base);
Object.assign(style.navitem.default, style.navitem.base);
```

Object.assign виконує об'єднання елементів, переписуючі лише ті елементи, які є в вихідному об'єктів.

Єдиною перепоною використання *Object.assign* є те, що ця функція з'явилася лише в ECMAScript 6, тобто старі браузери можуть це не підтримувати. Але, по-перше, треба завжди орієнтуватися на новітні розробки, бо сам таки вони будуть поширюватися швидше. А по-друге, завжди можна скористатися наповнювачами (*polyfill*) – бібліотек, які додають в старі браузери підтримку можливостей, які в сучасних браузерах є вбудованими.

Для нашого випадку створимо файл *game/lib/polyfill.js* для підримки *Object.assign* та *.forEach*, та додамо туди код, який можна отримати за посиланнями:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign

та

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach

(дивіться розділи «PolyFill»).

Результат роботи над головним меню наведений на рис. 6.2.

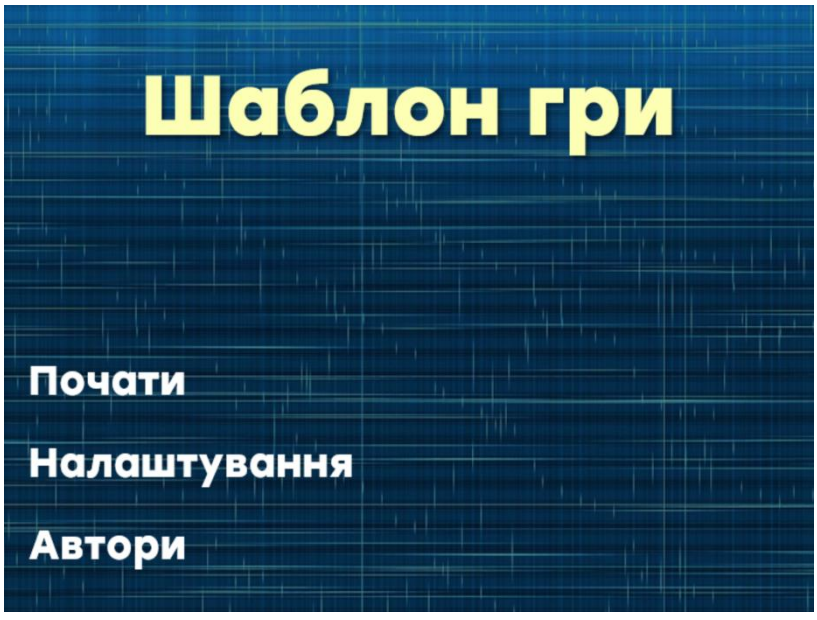

Рисунок 6.2 – Головне меню ігрового додатку

Із урахуванням усіх покращень, код *gamemenu.js* набуває вигляду:

```
var GameMenu = function() {};
```

```
GameMenu.prototype = {

  addMenuOption: function(text, callback) {
    var txt = game.add.text(30, (this.optionCount * 80) + 200, text,
                            style.navitem.default);
    txt.inputEnabled = true;
    txt.events.onInputUp.add(callback);

    txt.events.onInputOver.add(function (target) {
      target.setStyle(style.navitem.hover);
    });

    txt.events.onInputOut.add(function (target) {
      target.setStyle(style.navitem.default);
    });
    this.optionCount ++;
  },

  init: function () {
    this.titleText = game.make.text(game.world.centerX, 100, "Game Title", {
      font: 'bold 60pt TheMinion',
      fill: '#FDFFB5',
      align: 'center'
    });
    this.titleText.setShadow(3, 3, 'rgba(0,0,0,0.5)', 5);
    this.titleText.anchor.set(0.5);
    this.optionCount = 1;
  },

  create: function () {
    game.stage.disableVisibilityChange = true;

    game.add.sprite(0, 0, 'menu-bg');
    game.add.existing(this.titleText);

    this.addMenuOption('Почати', function() {
      console.log('Натиснуто "Почати"!');
    });

    this.addMenuOption('Налаштування', function() {
      console.log('Натиснуто "налаштування"!');
    });

    this.addMenuOption('Автори', function() {
      console.log('Натиснуто "Автори"!');
    });
  }
};
```

6.4.10 **Створення екрану налаштувань**

Перш за все треба створити файл *game/states/Options.js*, який буде відповідати стану *Options*:

```
var Options = function() {};

Options.prototype = {
  preload: function () {
  },
```

```
  create: function () {
  }
};
```

Далі необхідно додати стан налаштувань до файлу *game/states/Splash.js*:

```
game.state.add("Options",Options);
```

та додати відповідний пункт меню до головного меню (*game/states/GameMenu.js*):

```
this.addMenuOption('Options', function () {
  game.state.start("Options");
});
```

Фонову картинку для стану налаштувань є доцільно завантажити разом із іншими ресурсами у функції *loadImages()* стану *Splash*:

```
loadImages: function () {
  ...
  game.load.image('options-bg', 'assets/images/options-bg.jpg');
},
```

Для додання навігаційного елементу повернення із налаштувань до головного меню використаємо той же код, що й в самому головному меню. Треба тільки трохи модифікувати обробник функції зворотнього виклику:

```
this.addMenuOption('<- Назад', function (e) {
  // Переміщуємося назад до головного меню
  game.state.start("GameMenu");
});
```

Але в такого способу використання *addMenuOption()* є значний недолік – для використання цієї функції в стані *Options* доведеться скопіювати достатньо багато коду, який фактично становиться дублюючим. І при можливих змінах у коді в майбутньому доведеться міняти його в двох (а може й більше!) місцях. Якщо пригадати, що створюємо шаблон для побудування додатків, то така ситуація виглядає вельми некоректною.

Для вирішення цієї проблеми можна йти декількома шляхами. Першим і очевидним є використання наслідування – у вигляді наслідування прототипів в рамках JavaScript ES5. Отже ми можемо задекларувати деякий «базовий» стан та помістити до нього функцію *addMenuOption()*:

```
var BaseState = function () {};

BaseState.prototype.addMenuOption = function (text, callback) {
  var txt = game.add.text(30, (this.optionCount * 80) + 200, text,
                          style.navitem.default);
  txt.inputEnabled = true;
  txt.events.onInputUp.add(callback);

  txt.events.onInputOver.add(function (target) {
    target.setStyle(style.navitem.hover);
```

```
  });

  txt.events.onInputOut.add(function (target) {
    target.setStyle(style.navitem.default);
  });
  this.optionCount++;
};
```

І після цього просто наслідувати кожен стан від базового, додаючи свій код для кожного стану. Але із урахуванням процедури наповнення об’єктів, шлях наслідування приведе нас до використання *Object.assign* та *Object.create* на кшталт такої конструкції:

```
var GameMenu = Object.assign(Object.create(BaseState.prototype), { ... });
```

Вона працює, але є досить громіздкою, особливо із урахуванням того, що ми прагнули лише до повторного використання єдиної функції *addMenuOption()* – і більше нічого.

Таким чином ми переходимо на другий можливий шлях – до використання домішок (mixin). Домішка в JavaScript – клас або об'єкт, який реалізує якусь чітко виділену поведінку, і використовується для уточнення поведінки інших класів, не призначений для самостійного використання.

В Phaser домішки реалізуються зо допомогою функції *mixinPrototype*. Отже створимо файл *game/lib/mixins.js* та додамо до нього домішку із функцією *addMenuOption()*:

```
var mixins = {
  addMenuOption: function(text, callback) {
    var txt = game.add.text(30, (this.optionCount * 80) + 200, text,
                            style.navitem.default);
    txt.inputEnabled = true;
    txt.events.onInputUp.add(callback);

    txt.events.onInputOver.add(function (target) {
      target.setStyle(style.navitem.hover);
    });
    txt.events.onInputOut.add(function (target) {

      target.setStyle(style.navitem.default);
    });
    this.optionCount ++;
  }
};
```

Після підключення файлу *game/lib/mixins.js* до проекту ми можемо додавати домішку там, де забажаємо:

```
GameMenu.prototype = {

  init: function () {
    ...
  },

  create: function () {
```

```
    ...
  }
};

Phaser.Utils.mixinPrototype(GameMenu.prototype, mixins);
```

Останній рядок коду фактично означає, що ми домішуємо до *GameMenu* вміст змінної *mixins*, тобто функцію *addMenuOption()*.

6.4.11 Додаткові параметри меню

На даний момент меню, що було створене для використання лише в головному меню додатку, може бути використане в будь-якому стані. Але при такій моделі використання з'являються дві проблеми, що необхідно розв'язати:

1. Позиціонування меню на екрані, бо для різних ситуацій необхідні різні позиції.
2. Налаштування стилю меню, бо фонові картинки можуть бути різними.

Почнемо з другої проблеми і створимо для *navitem* додатковий стиль *inverse*, який «інверсує» заповнення і обведення для шрифту меню:

```
inverse: {
  fill: 'black',
  stroke: 'black'
},
```

Після цього потрібно не забути виконати *Object.assign* аналогічно до попередніх стилів:

```
Object.assign(style.navitem.inverse, style.navitem.base);
Object.assign(style.navitem.hover, style.navitem.base);
Object.assign(style.navitem.default, style.navitem.base);
```

Все працює, але виглядає вельми некрасиво з-за кількості однотипних викликів. Якщо проаналізувати усі ці *Object.assign*, то вони беруть *style.navitem.base* та зв'язують його з *style.navitem.everythingelse*. А раз так, то це можна узагальнити:

```
for (var key in style.navitem) {
  if (key !== "base") {
    Object.assign(style.navitem[key], style.navitem.base)
  }
}
```

Тепер додамо до *addMenuOption()* необов'язковий аргумент, що відповідає за стиль оформлення пункту меню. Виклик функції буде таким:

```
// addMenuOption(label, callback, style);
this.addMenuOption('<- Назад', function (e) {
  this.game.state.start("GameMenu");
}, 'inverse');
```

Відповідно зміниться й домішка:

```
addMenuOption: function(text, callback, className) {
  className || (className = "default");
  var txt = game.add.text(30, (this.optionCount * 80) + 200, text,
style.navitem[className]);
  txt.inputEnabled = true;
  txt.events.onInputUp.add(callback);
  txt.events.onInputOver.add(function (target) {
    target.setStyle(style.navitem.hover);
  });
  txt.events.onInputOut.add(function (target) {
    target.setStyle(style.navitem[className]);
  });
  this.optionCount ++;
}
```

Тепер перейдемо до проблеми позиціонування. Логічним рішенням було б використати ще більше параметрів до функції, та передавати крім стилю ще й позицію. Однак це є не зовсім правильним, бо код набуває все більшої складності та стає менш зрозумілим. Отже оберемо найбільш очевидний та простий спосіб – ініціалізацію конфігурації меню на початку кожного режиму:

Це дозволяє відмовитися від перевантаження конструктору, замість чого ми використовуємо конфігураційний об’єкт, який фабрика пунктів меню може задіяти через посилання:

```
Options.prototype = {

  menuConfig: {
    className: '' // ім'я класу стилю для пункту меню
    startY: {},
    startX: {},  // Координата x або "center" для центрування
  },
```

Відповідно зміниться і функція *addMenuOption()*:

```
addMenuOption: function(text, callback, className) {

  // Використовуємо вказаний або значення за замовчуванням
  className || (className = this.menuConfig.className || 'default');

  // Встановлюємо X-координату в game.world.center якщо в параметрі
  // задано "center", інакше використовує momenuConfig.startX
  var x = this.menuConfig.startX === "center" ?
    game.world.centerX :
    this.menuConfig.startX;

  // Встановлюємо значення із об'єкту ініціалізації
  var y = this.menuConfig.startY;

  // створюємо надпис
  var txt = game.add.text(
    x,
    (this.optionCount * 80) + y,
    text,
```

```
    style.navitem[className]
  );

  // Використовуємо прив'язку об'єкта до центру, якщо задано "center"
  txt.anchor.setTo(this.menuConfig.startX === "center" ? 0.5 : 0.0);

  txt.inputEnabled = true;

  txt.events.onInputUp.add(callback);

  txt.events.onInputOver.add(function (target) {
    target.setStyle(style.navitem.hover);
  });

  txt.events.onInputOut.add(function (target) {
    target.setStyle(style.navitem[className]);
  });

  this.optionCount++;
}
```

Із урахуванням змін в *addMenuOption()* треба додати об’єкт ініціалізации і до стану *GameMenu*:

```
GameMenu.prototype = {
  // className не задаємо, тобто використовується параметр за замовчуванням
  menuConfig: {
    startY: 260,
    startX: 30
  },
  ...
```

А для стану *Options* об’єкт буде виглядати так:

```
menuConfig: {
  className: "inverse",
  startY: 260,
  startX: "center"
},
```

6.4.12 **Функціональність режиму налаштувань**

Прийшов час наповнити екран опцій налаштуваннями. Ще раз нагадаємо собі, що ми створюємо шаблон ігрового додатку, і спробуємо представити які параметри є в будь-якій грі, незважаючи на її жанр. Перш за все на думку спадають налаштування звуку – при чому в кожній грі є як фонова музика, так і ігрові звуки. І користувач повинен мати можливість відключати ці параметри окремо один від одного. Отже, додамо звукові налаштування до нашого шаблону.

Перш за все треба обрати, а де саме розмістити змінні, що будуть керувати цими налаштуваннями. Проаналізувавши загальну структуру додатку, можна зробити висновок, що найвдалішим місцем для розміщення буде *game/main.js*. А якщо подумати трохи наперед, то краще об’єднати параметри до якоїсь структури – потім можна буде дуже просто додавати

інші налаштування, які обов'язково з'являться у грі. Отже, отримаємо такий код:

```
var
  game = new Phaser.Game(800, 600, Phaser.AUTO, 'game'),
  Main = function () {},
  gameOptions = {
    playSound: true,
    playMusic: true
  },
  musicPlayer;
```

Зверніть увагу, що обидва параметри мають за замовченням значення true, тобто і фонова музика, і звуки повинні програватися після первісного запуску гри.

Тепер треба вказати, що саме пункти меню в стані *Options* будуть керувати цими налаштуваннями:

```
create: function () {
  var playSound = gameOptions.playSound,
      playMusic = gameOptions.playMusic;

  game.add.sprite(0, 0, 'options-bg');
  game.add.existing(this.titleText);

  this.addMenuOption(playMusic ? 'Вимкнути музику' : 'Увімкнути музику',
    function (target) {
      playMusic = !playMusic;
      target.text = playMusic ? 'Вимкнути музику' : 'Увімкнути музику';
      musicPlayer.volume = playMusic ? 1 : 0;
    });

  this.addMenuOption(playSound ? 'Вимкнути звуки' : 'Увімкнути звуки',
    function (target) {
      playSound = !playSound;
      target.text = playSound ? 'Вимкнути звуки' : 'Увімкнути звуки';
    });

  this.addMenuOption('<- Назад', function () {
    game.state.start("GameMenu");
  });
}
```

Тепер залишилося додати екрану-заставки ініціалізацію об'єкту *musicPlayer*:

```
addGameMusic: function () {
  musicPlayer = game.add.audio('dangerous');
  musicPlayer.loop = true;
  musicPlayer.play();
},
```

та викликати *addGameMusic()* в функції *create()*:

```
create: function() {
  this.status.setText('Готово!');
  this.addGameStates();
  this.addGameMusic();
```

```
  setTimeout(function () {
    game.state.start("GameMenu");
  }, 1000);
}
```

Отже на цьому можна завершити роботу над станом *Options*. Результат наведений на рис. 6.3.

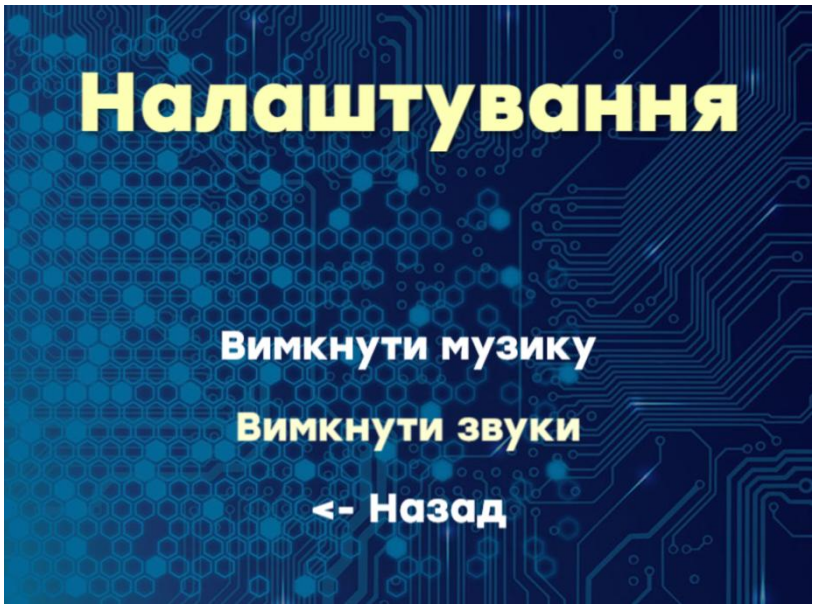

Рисунок 6.3 – Екран налаштувань

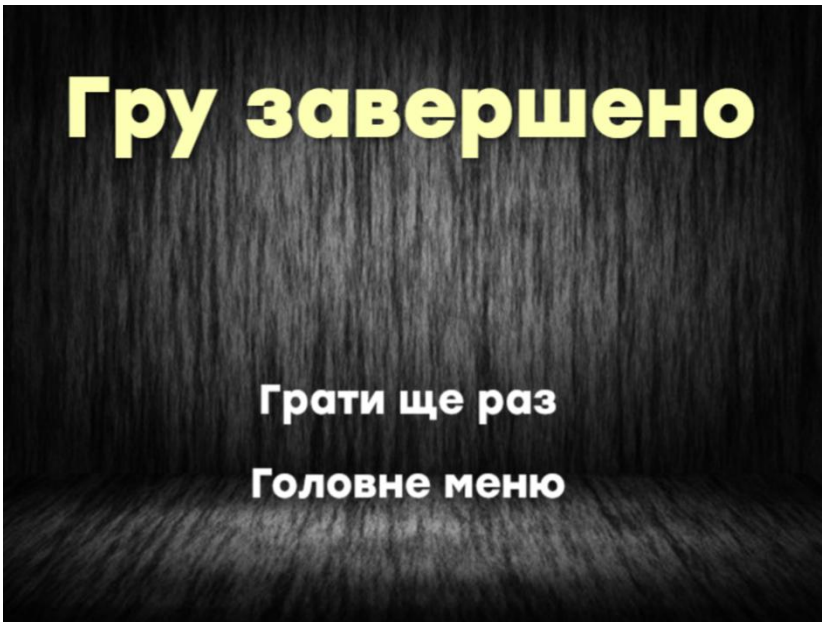

Рисунок 6.4 – Екран завершення гри

Та й взагалі шаблон додатку є майже завершеним.

Стан *GameOver* (Кінець гри) повинен містити результати гри (рахунок очок, монет, час проходження або щось таке інше), кнопку повторної гри та повернення до головного меню (рис. 6.4). Розглядати його реалізацію окремо та докладно немає необхідності.

Стан *Credits* (Автори), який був задекларований ще в самому початку, теж фактично є повним аналогом стану *Options*. Або навіть простішим, бо замість пунктів меню для налаштувань буде мати звичайний статичний об’єкт із описом авторів та правової інформації.

Очікувані практичні результати роботи:

– шаблон ігрового додатку із базовими станами, в якому для отримання повноцінного ігрового HTML5-додатку треба лише наповнити стан *Game* ігровим процесом.

7 Лабораторна робота №3. Особливості організації ігрового процесу та основи роботи із фізичними движками в Phaser.

Лабораторна робота орієнтована на оволодіння студентами навичок із організації ігрового процесу, створення ігрових додатків із підтримкою фізичних движків фреймворку Phaser та випадковою генерацією ігрових елементів.

Мета лабораторної роботи:

– навчитися готувати ресурси, необхідні для реалізації ігрових додатків із підтримкою фізичних движків (моделі об’єктів, фізичні карти тощо);
– ознайомитися із особливостями створення і налаштування об’єктів для використання в рамках фізичних процесів ігрового движка P2;
– навчитися налаштовувати різноманітні матеріали, що є основною взаємодії в фізичному движку P2, а також створювати контактні матеріали на основі базових із налаштуванням параметрів взаємодії;
– ознайомитися із відмінностями візуальних і фізичних об’єктів движку P2 та засобами перетворення координат між різними представленнями;
– навчитися прив’язувати об’єкти один до одного із налаштуванням різноманітних типів обмежень (constraints) та вказувати точки прив’язки;

- навчитися організувати користувацьке керування ігровим об’єктом із урахуванням особливостей фізичного движка та поточного положення об’єкту;
- ознайомитися із способами динамічної генерації контенту (поверхонь або ландшафтів) на основі масивів вершин, що були згенеровані шляхом наповнення випадковими значеннями.

Відповідно до завдання лабораторної роботи студент реалізує ігровий процес переміщення автомобіля по автоматично згенерованій поверхні за допомогою засобів фізичного движка P2, що є частиною фреймворку Phaser. Після створення ігрового процесу студент інтегрує його до шаблону додатку, який було створено в ЛР №2 і таким чином отримує повністю закінчений ігровий HTML5-додаток.

У разі успішного виконання лабораторної роботи студент оволодіє знаннями щодо особливостей застосування фізичного движка P2, використання фізичних карт об’єктів, базових і контактних матеріалів, прив’язування об’єктів та створить алгоритм автоматичної генерації ігрової поверхні на основі випадкових значень.

Успішне засвоєння матеріалів та виконання лабораторної роботи сприяє формування у студента наступних соціальних навичок та вмінь: бажання постійно вчитися, вміння дотримуватися регламентних рамок, оцінювати строки проведення та виконання роботи, обґрунтувати той чи інший спосіб діяльності при виконанні практичних завдань. Форми прояву: прагнення вирішувати поточні проблеми самостійно, бажання пробувати щось нове, йти на розумний ризик при прийнятті рішень, елементи самоконтролю і самооцінки при виконанні роботи, вміння планувати свій час.

7.1 **Завдання до лабораторної роботи**

1. Створити та завантажити необхідні ігрові ресурси.
2. Проініціалізувати ігровий світ та запустити движок P2.
3. Створити ігровий об’єкт автомобілю на включити для нього використання фізики P2.
4. Створити групу коліс та додати до ней об’єкти-колеса.
5. Створити матеріали світу, коліс та контактний матеріал між ними, налаштувати тертя та пружність.
6. Прив’язати колеса до автомобілю за допомогою механізму обмежень (constraints) движка P2.

7. Організувати підтримку керування автомобілем за допомогою клавиш-стрілок, забезпечити можливість налаштування швидкості обертання коліс.
8. Додати підтримку стрибків, урахувати можливість стрибків тільки при знаходження автомобіля на поверхні.
9. Оптимізувати код:
   - розділити створення автомобіля та поверхні по окремих функціях;
   - винести створення та обробку фізичних P2-тіл автомобіля та коліс в окремий клас *Vehicle*;
   - створити окремий клас *TerrainController* для автоматичної генерації ігрової поверхні в заданому вікні (x1, y1, x2, y2).
10. Проконтролювати коректність вхідних параметрів конструктора TerrainController, на основі яких формуються горизонтальні та вертикальні обмеження поверхні.
11. Організувати масив для збереження вершин поверхні, створити окремі функції-прототипи для контейнеру та криволінійної поверхні.
12. Створити функцію генерації контейнеру вершин із виходом за межі ігрового світу, передбачити проміжок для генерації криволінійної поверхні.
13. Створити алгоритм генерації ігрової поверхні із послідовним створення підйомів та схилів із випадковою амплітудою, передбачити створення випадкових пагорбів та впадин.
14. Створити функцію малювання поверхні у вигляді сплайнової кривої на основі масиву вершин із заданими візуальними налаштуваннями.
15. Побудувати на основі масиву вершин фізичне P2-тіло поверхні та додати його до ігрового світу (урахувати необхідність перетворення фізичних P2-координат у Phaser-координати).
16. Протестувати ігровий процес та внести корективи до алгоритму генерації поверхні та іншого коду.
17. Додати ігровий процес до шаблону гри із ЛР №2, протестувати отриманий ігровий додаток.

7.2 Підготовка до лабораторної роботи № 3

При підготовці до виконання лабораторної роботи необхідно:

1. Ознайомитися із особливостями, перевагами, недоліками та відмінностями різних фізичних движків, що входять до складу Phaser.
2. Ознайомитися із методичними вказівками щодо виконання лабораторної роботи (див. 7.4).

7.3 Контрольні питання і попередні матеріали для допуску до лабораторної роботи № 3

7.3.1 Контрольні питання

1. Вкажіть особливості створення спрайтів в Phaser.
2. В чому відмінність зображень (*Image*) та спрайтів (*Sprite*)?
3. Чим тайловий спрайт (*TileSprite*) відрізняється від звичайного спрайта?
4. Які основні обробники подій має кнопка (*Button*)?
5. В яких випадках використовується пакет спрайтів (*SpriteBatch*), в чому особливості використання?
6. В чому призначення та особливості використання мотузки (*Rope*)?
7. Який об’єкт використовується для рисування примітивної графіки, в чому переваги і недоліки його використання?
8. Наведіть особливості та приклад використання бітової карти (*BitmapData*).
9. Як і де використовується об’єкт текстура для рендеру (*RenderTexture*)?
10. Які види текстових об’єктів підтримуються в Phaser, в чому відмінності їх налаштування та використання?
11. Які основні класи використовуються для анімації в Phaser, в чому їх відмінності?
12. Які типи часу підтримує Phaser та чим вони відрізняються?
13. Для чого використовуються тайлові карти (*TileMap*), які основні класи задіяні і для чого?
14. Яке призначення та особливості використання системи частинок (*Particles*)?

7.3.2 Попередні матеріали

Для допуску к виконанню лабораторної роботи необхідно представити:

1. Спрайт та фізична карта автомобіля (PNG + JSON).
2. Спрайт колеса (PNG).

3. Спрайт пагорба (PNG).

7.4 Методичні вказівки до виконання лабораторної роботи № 3

7.4.1 Створення та завантаження ресурсів

В лабораторній роботі створимо досить простий додаток, який демонструє використання фізичного движка P2, що є найбільш досконалим варіантом для Phaser. Запропонований додаток реалізує ігровий об’єкт-автомобіль, який може рухатися вперед і назад, долати різноманітний рельєф та підстрибувати.

В цій лабораторній роботі не будемо розглядати створення структури папок та HTML-файле точки входу, бо це вже розглядалося в попередніх роботах, одразу перейдемо до розробки. Перш за все, треба створити необхідні спрайти на помістити їх до папки assets. Для першого варіанту додатку необхідні три об’єкти:

- власне автомобіль (*truck.png*);
- колеса автомобіля (*wheel.png*);
- пагорб, який буде долати автомобіль (*hill.png*).

Автомобіль і пагорб можна створити за допомогою будь-якого графічного редактора, наприклад, онлайн-інструментом Loon Physics (https://loonride.com/tools/physics) або будь-яким інструментом створення ігрової графіки, бо для автомобіля знадобиться не лише зображення, а й фізична карта у вигляді JSON-файлу (рис. 7.1).

Автомобіль може мати будь який вигляд, оптимальним є 15-20 вершин. Отже, всі необхідні для ігрового додатку графічні ресурси наведені на рис. 7.2.

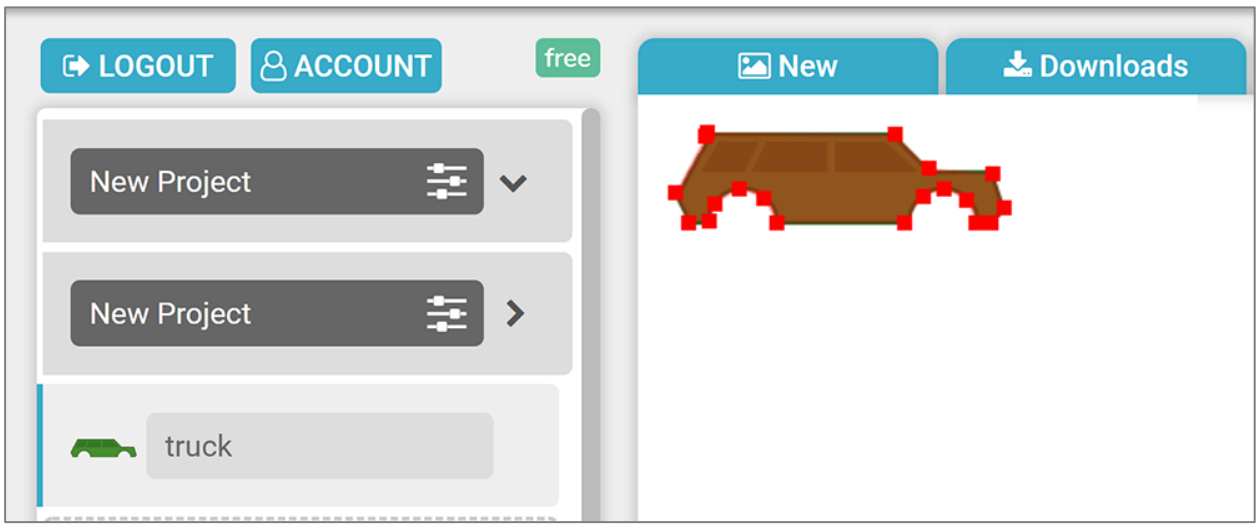

Рисунок 7.1 – Модель автомобіля в Loon Physics

Рисунок 7.2 – Графічні ресурси ігрового додатку

Далі треба завантажити ресурси в функції preload():

```
game.load.image('truck', 'asset/truck.png');
game.load.image('wheel', 'asset/wheel.png');
game.load.image('hill', 'asset/hill.png');
game.load.physics("physics", "asset/physics.json");
```

7.4.2 Налаштування автомобіля та фізики

Перейдемо до функції *create()*, де ми перш за все повинні створити ігровий світ. Вкажемо достатньо великий розмір, щоб автомобіль мав можливість вільно рухатися. Крім того, завантажимо фізичний движок P2 та налаштуємо рівень гравітації:

```
game.world.setBounds(0,0,width*2,height);
game.physics.startSystem(Phaser.Physics.P2JS);
game.physics.p2.gravity.y = 300;
```

Перейдемо до створення автомобілю. Додамо відповідний спрайт, налаштуємо для нього підтримку P2-фізики та завантажимо фізичну карту, що буле створена раніше

```
truck = game.add.sprite(width*0.25, height*0.8, "truck");
game.physics.p2.enable(truck, showBodies);
truck.body.clearShapes();
truck.body.loadPolygon("physics", "truck");
```

Із урахуванням того, що ігровий світ більше розміру вікна перегляду додатку, то необхідно прив'язати камеру до автомобіля:

```
game.camera.follow(truck);
```

Тепер перейдемо до коліс. Маємо два колеса, тому логічно створити для них групу:

```
wheels = game.add.group();
```

Далі перед безпосереднім додаванням коліс створимо два матеріали – для коліс та для ігрового світу. Пізніше ми налаштуємо, що саме буде відбуватися при взаємодії цих двох матеріалів, а зараз просто створимо їх:

```
wheelMaterial = game.physics.p2.createMaterial("wheelMaterial");
var worldMaterial = game.physics.p2.createMaterial("worldMaterial");
```

Для створення об'єктів-коліс задекларуємо функцію *initWheel([x,y])*, яке буде додавати колеса за вказаними координатами відносно автомобіля:

```
var distBelowTruck = 24;
initWheel([55,distBelowTruck]);
initWheel([-52, distBelowTruck]);
```

Отже далі розглянемо вміст функції *initWheel()*, і почнемо із декларації:

```
function initWheel(offsetFromTruck) {
    ...
}
```

В функції перш за все необхідно виконати позиціонування колеса відносно автомобіля

```
var truckX = truck.position.x;
var truckY = truck.position.y;
var wheel = game.add.sprite(truckX + offsetFromTruck[0],
      truckY + offsetFromTruck[1], "wheel");
```

Таке позиціонування просто тримає колеса біля автомобіля, не більше того. Але це дуже просто виправити, додавши до колеса круговий полігон:

```
game.physics.p2.enable(wheel, showBodies);
wheel.body.clearShapes();
wheel.body.addCircle(15.5);
```

Зауважимо, що колеса повинні кріпитися до автомобіля в точках опори, на яких вони можуть вільно обертатися. Для таких випадків в P2-движку є обертальне обмеження *createRevoluteConstraint()*, що «з'єднує два тіла в зазначеній точці зміщення та дозволяє їм вільно обертатися один відносно іншого навколо саме цієї точки» (це видержка із документації на P2):

```
createRevoluteConstraint(bodyA, pivotA, bodyB, pivotB, maxForce)
```

Отже, використаємо такий код на основі *createRevoluteConstraint()*:

```
var maxForce = 100;
var rev = game.physics.p2.createRevoluteConstraint(truck.body,
offsetFromTruck,
   wheel.body, [0,0], maxForce);
```

Отже, точкою опори для автомобіля є місце, де розміщено колесо. Опорна точка колеса – це просто його центр, тому й маємо [0,0]. *maxForce* є необов'язковим параметром, який задає максимальна сила, яку буде застосовано для узгодження поворотів двох тіл. За замовченням це буде еквівалентом нескінченної сили. Але методом підбору було обране значення 100 для створення ефекту прискорення обертання коліс.

Далі додамо колесо до групи

```
wheels.add(wheel);
```

та створимо обробник події контакту колеса з якимось іншим об’єктом:

```
wheel.body.onBeginContact.add(onWheelContact, game);
```

До цієї функції ми повернемось пізніше. А наприкінці функції *initWheel()* додамо до колеса матеріал, створений раніше:

```
wheel.body.setMaterial(wheelMaterial);
```

7.4.3 Контактні матеріали

Повернемося до функції *create()*, де ми викликали функцію створення коліс. Після цього необхідно налаштувати граничний матеріал для всіх сторін ігрового світу:

```
game.physics.p2.setWorldMaterial(worldMaterial, true, true, true, true);
```

В Phaser існує поняття «контактного матеріалу», який дозволяє задати ступінь тертя та реституції (пружності) між двома матеріалами при їх контакті. Отже, створимо такий контактний матеріал між матеріалами ігрового світу та колесами:

```
var contactMaterial =
      game.physics.p2.createContactMaterial(wheelMaterial,worldMaterial);
```

Це робиться тому, що за замовчуванням тертя між колесами та поверхнею є замалим для створення хоча б якогось тертя. Змінити ситуацію можна шляхом встановлення великого значення для тертя та малого – для пружності:

```
contactMaterial.friction = 1e3;
contactMaterial.restitution = .3;
```

Обов’язково проекспериментуйте із цими значеннями для розуміння особливостей взаємодії двох матеріалів.

7.4.4 Організація керування автомобілем

Перейдемо до керування автомобілем і перш за все проініціалізуємо клавіші-стрілки та пробіл:

```
cursors = game.input.keyboard.createCursorKeys();
var spaceKey = game.input.keyboard.addKey(Phaser.Keyboard.SPACEBAR);
```

Тепер при натисканні клавіші пробілу, треба підкинути автомобіль до гори, що й буде реалізацією механізму стрибання:

```
spaceKey.onDown.add(function() {
   if (allowTruckBounce) {
      truck.body.moveUp(500);
      allowTruckBounce = false;
   }
}, game);
```

Висота стрибка може налаштовуватися завдяки параметру функції *moveUp()*.

Зверніть увагу, що підстрибувати автомобіль може тільки у разі знаходження на «землі» – за це відповідає змінна *allowTruckBounce*. Тобто, після стрибку *allowTruckBounce* скидається в false та автомобіль не може стрибнути ще раз. Але де тоді буде правильним встановлювати *allowTruckBounce* в true?

Повернемося до функції *onWheelContact()*, яка була раніше прив'язана до контакту колеса із будь-яким іншим об'єктом. Отже, при виклику *onWheelContact()* до неї передається багато параметрів, що докладно описують контакт. Але ми використаємо лише два з них:

```
function onWheelContact(phaserBody, p2Body) {
    ...
}
```

В середині функції ми за допомогою цих аргументів з'ясуємо чи був контакт колеса із поверхнею або пагорбом. Якщо був – встановлюємо *allowTruckBounce* та дозволяємо стрибати знову:

```
if ((phaserBody === null && p2Body.id == 4)
|| (phaserBody && phaserBody.sprite.key == "hill")) {
   allowTruckBounce = true;
}
```

Зверніть увагу, що «p2Body.id == 4» – це визначення нижньої границі ігрового світу в движку P2.

І нарешті останнє, що ми повинні реалізувати, це переміщення автомобіля за допомогою клавіш стрілок. Для цього в функції *update()* ми обробляємо натискання клавіш «вправо» та «вліво» і встановлюємо відповідні значення швидкості обертання коліс:

```
var rotationSpeed = 300;
if (cursors.left.isDown) {
   wheels.children.forEach(function(wheel,index) {
      wheel.body.rotateLeft(rotationSpeed);
   });
}
else if (cursors.right.isDown) {
   wheels.children.forEach(function(wheel,index) {
      wheel.body.rotateRight(rotationSpeed);
```

```
    });
}
else {
    wheels.children.forEach(function(wheel,index) {
        wheel.body.setZeroRotation();
    });
}
```

Результат виконання роботи наведений на рис. 7.3.

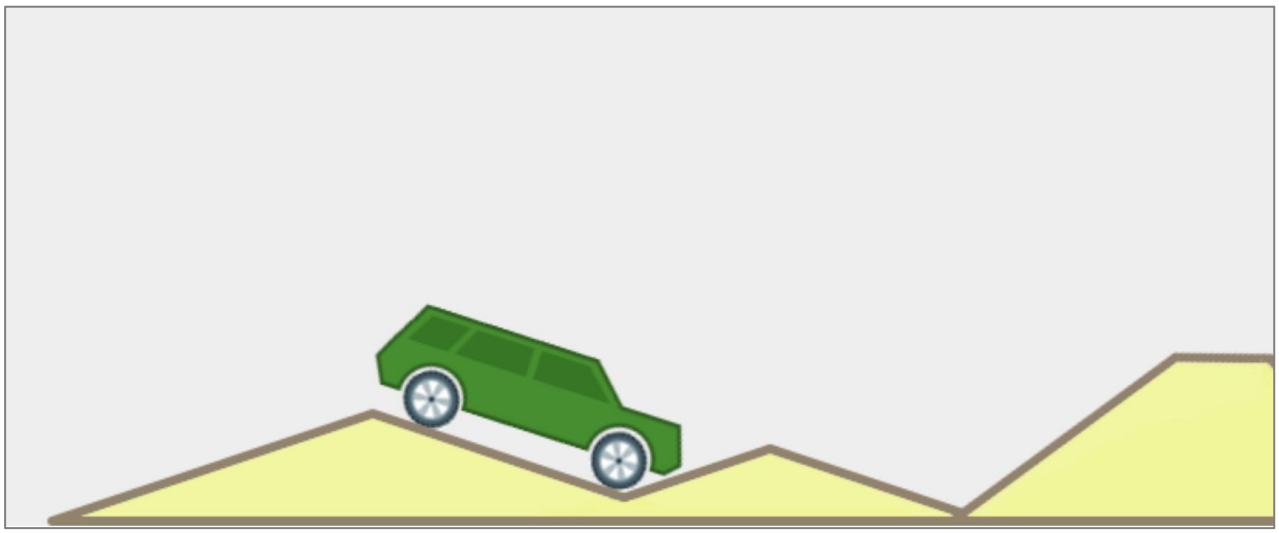

Рисунок 7.3 – Модель використання движку P2

7.4.5 Організація та вдосконалення коду

Наступним кроком в розвитку додатку буде автоматична генерація поверхні (рельєфу), по якому буде пересуватися автомобіль. Але перед тим виконаємо деяку оптимізацію коду та поділимо його на три скрипти:

- *game.js* – базовий код, що обслуговує загальний механізм гри та створює усі об’єкти;
- *vehicle.js* – містить клас, що описує автомобіль;
- *terrain.js* – клас, що представляє об’єкт-поверхню.

Коротко розглянемо перші два файли, а потім перейдемо до докладного опису автоматичної генерації поверхні. Отже, створення головного ігрового об’єкту та завантаження ресурсів залишаться майже без змін:

```
var width = 800;
var height = 500;
var game = new Phaser.Game(width, height, Phaser.AUTO, null,
    {preload: preload, create: create, update: update});

var truck;
var wheelMaterial;
var worldMaterial;
var allowTruckBounce = true;
```

```
function preload() {
  game.stage.backgroundColor = '#eee';
  game.load.image('truck', 'asset/truck.png');
  game.load.image('wheel', 'asset/wheel.png');
  game.load.physics("physics", "asset/physics.json");
}
```

В функції *create()* ми виконуємо ініціалізацію движка P2 та створення світу із великим розмірами для переміщення автомобіля. Також додаються матеріали для коліс та світу, після чого для цієї пари створюється контактний матеріал. Це дозволяє налаштувати ступінь тертя та пружності між колесами та поверхнею. Наприкінці коду викликаються функції *initTruck()* та *initTerrain()* для ініціалізації автомобіля та поверхні.

```
function create() {
  // встановлення великих меж світу відповідно до розмірі екрану
  game.world.setBounds(0,-height,width*10,height*2);
  game.physics.startSystem(Phaser.Physics.P2JS);
  game.physics.p2.gravity.y = 600;

  wheelMaterial = game.physics.p2.createMaterial("wheelMaterial");
  worldMaterial = game.physics.p2.createMaterial("worldMaterial");
  game.physics.p2.setWorldMaterial(worldMaterial, true, true, true, true);

  // створення контактного матеріалу
  var contactMaterial = game.physics.p2.createContactMaterial(
                        wheelMaterial, worldMaterial);
  contactMaterial.friction = 1e3;
  contactMaterial.restitution = .3;

  // обробник натискання на пробіл
  var spaceKey = game.input.keyboard.addKey(Phaser.Keyboard.SPACEBAR);
  spaceKey.onDown.add(onSpaceKeyDown, game);

  initTruck();
  initTerrain();
}
```

Функція *initTruck()* має такий вигляд:

```
// ініціалізація автомобіля і відповідного фізичного тіла
function initTruck() {
  var truckFrame = game.add.sprite(width*0.25, height*0.4, "truck");
  truck = new Vehicle(truckFrame);
  truck.frame.body.clearShapes();
  truck.frame.body.loadPolygon("physics", "truck");
  game.camera.follow(truck.frame);

  var distBelowTruck = 24;
  initWheel([55, distBelowTruck]);
  initWheel([-52, distBelowTruck]);
}
```

В функції створюється спрайт автомобіля та передається у якості параметру для створення *Vehicle*-об’єкту, після чого завантажує контури

фізичного тіла із JSON-файлу. Далі відбувається налаштування камери та додавання двох коліс через функцію *initWheel()*:

```
// ініціалізація колеса
function initWheel(offsetFromTruck) {
  var wheel = truck.addWheel("wheel", offsetFromTruck);
  wheel.body.setMaterial(wheelMaterial);
  wheel.body.onBeginContact.add(onWheelContact, game);
  return wheel;
}
```

Зверніть увагу, що ця функція використовує метод *addWheel()* класу *Vehicle* для створення і розміщення колеса відносно автомобіля.

Що до інших функцій game.js – *onSpaceKeyDown*, *onWheelContact* – то вони не залишилися незмінними, а функцію *initTerrain()* ми розглянемо докладно трохи пізніше.

А поки подивимось на реалізацію класу *Vehicle*:

```
Vehicle = function(frameSprite) {
  this.frame = frameSprite;
  this.game = this.frame.game;
  this.wheels = this.game.add.group();
  this.debug = false;

  this.game.physics.p2.enable(this.frame, this.debug);

  this.cursors = this.game.input.keyboard.createCursorKeys();

  this.maxWheelForce = 1000;
  this.rotationSpeed = 400;
}

Vehicle.prototype = {
  /*
   параметри:
     {String} wheelSpriteKey - передзавантажений спрайт для колеса
     {Array} offsetFromVehicle - координати колеса відносно авто
                                 у вигляді зміщення x,y
   значення, що повертається:
     {Phaser.Sprite} - колесо, що було створене
  */
  addWheel: function(wheelSpriteKey, offsetFromVehicle) {
    var vehicleX = this.frame.position.x;
    var vehicleY = this.frame.position.y;
    var wheel = this.game.add.sprite(vehicleX + offsetFromVehicle[0],
            vehicleY + offsetFromVehicle[1], wheelSpriteKey);

    this.game.physics.p2.enable(wheel, this.debug);

    wheel.body.clearShapes();
    wheel.body.addCircle(wheel.width * 0.5);

    // прив'язка колеса  до автом у вказаній точці
    // із можливістю обертання коло неї
    var rev = this.game.physics.p2.createRevoluteConstraint(
        this.frame.body, offsetFromVehicle,
        wheel.body, [0,0], this.maxWheelForce
```

```
      );

    this.wheels.add(wheel);
    return wheel;
  },

  // керування обертанням коліс
  update: function() {
    var rotationSpeed = this.rotationSpeed;
    if (this.cursors.left.isDown) {
      this.wheels.children.forEach(function(wheel,index) {
        wheel.body.rotateLeft(rotationSpeed);
      });
    }
    else if (this.cursors.right.isDown) {
      this.wheels.children.forEach(function(wheel,index) {
        wheel.body.rotateRight(rotationSpeed);
      });
    }
    else {
      this.wheels.children.forEach(function(wheel,index) {
        wheel.body.setZeroRotation();
      });
    }
  }
};
```

В конструкторі цього класу відбувається ініціалізація P2-кадру та створення колісної групи. Метод *addWheels()* створює для колеса круговий фізичний полігон та прив'язує його до авто в заданій точці, надаючи колесу можливість обертання навколо неї.

В метод update винесене керування автомобілем за допомогою клавіш-стрілок, отже тепер його треба викликати в функції *update()* в *game.js*:

```
function update() {
  truck.update();
}
```

Отже, на даний момент ми створили автомобіль та розмістило його в лівому куті ігрового світу. Залишилось згенерувати поверхню, по якій буде переміщуватися автомобіль.

7.4.6 Генерація поверхні для переміщення автомобіля

Спочатку розглянемо функцію *initTerrain()* з файлу *game.js*:

```
function initTerrain() {
 // ініціалізація поверхні із вказанням меж
 var terrain = new TerrainController(game, 50, game.world.width - 50,
    100, height - 50);

 // малюємо поверхню
 terrain.drawOutline();

 // додаємо фізичне тіло
 var groundBody = terrain.addToWorld();
 groundBody.setMaterial(worldMaterial);
```

```
 groundBody.name = "terrain";
}
```

Тут створюється новий контролер *TerrainController* та до нього передаються межі, в яких необхідно створити поверхню. Далі викликається метод контролера для відмальовування поверхні у вигляді вигнутої чорної лінії. Після цього необхідно додати тіло поверхні до ігрового світу. Зверніть увагу, що жодних спрайтів не створюється, бо нас цікавить лише фізичне тіло.

Далі, перш ніж перейти до розгляду самого контролеру, спробуємо сформулювати – а що саме ми повинні зробити. Для побудування поверхні нам знадобиться відсортований масив вигляду [[x1, y1], [x2, y2], ...], який містить вершини нашої майбутньої поверхні.

Доцільно організувати поверхню у вигляді контейнеру та спочатку оточити нашу «довільну» частину із обох боків. Це дозволить безпечно розміщати вершини в межах контейнеру (рис. 7.4).

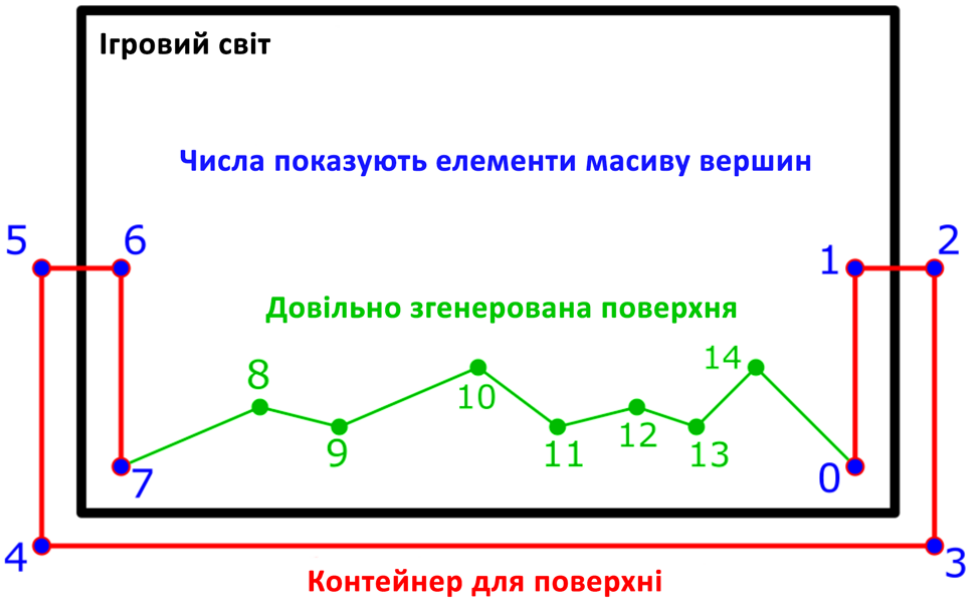

Рисунок 7.4 – Моделювання контейнеру для поверхні

Тепер ми маємо чітке уявлення про параметри поверхні та можемо перейти до створення *TerrainController* у файлі *terrain.js*. Почнемо із конструктора:

```
TerrainController = function(game, x1, x2, y1, y2) {
  this.game = game;
  if (x1 < x2) {
      this.leftBound = x1;
```

```
        this.rightBound = x2;
    }
    else {
        this.leftBound = x2;
        this.rightBound = x1;
    }

    if (y1 < y2) {
        this.topBound = y1;
        this.bottomBound = y2;
    }
    else {
        this.topBound = y2;
        this.bottomBound = y1;
    }
    // параметри для відлагодження
    this.debug = false;
    this.cliffs = true;
    this.fissures = true;

    // створюємо контейнер та додаємо його до масиву вершин
    this.vertices = [];
    var container = this.createContainer();
    this.vertices = this.vertices.concat(container);

    // створюємо поверхню та теж додаємо її до масиву вершин
    var terrain = this.generateTerrain(this.leftBound, this.rightBound,
        this.topBound, this.bottomBound);
    // перевертання координат для відповідності системі координат Phaser
    terrain = this.flipTerrain(terrain, this.topBound, this.bottomBound);
    this.vertices = this.vertices.concat(terrain);
}
```

На початку коду розміщуємо контроль за всіма можливими варіантами вхідних параметрів. Потім викликаємо *createContainer()* для генерації вершин контейнеру та *generateTerrain()* для генерації поверхні із вказанням обмежень. Вершини обидвох структур додаються до масиву вершин один за одним.

По суті, метод *createContainer()* виконує деяку прості арифметичні операції для підрахунку координат своїх вершин:

```
createContainer: function() {
    var w = this.game.world.width;
    var startHeight = this.bottomBound;
    var maxHeight = this.topBound;
    var sideWidth = 50;
    var lowerJut = 10;
    var upperJut = 20;

    // створюємо обмежувальний контейнер,
    //в який ми вставимо нашу довільно згенеровану поверхню
    var coreContainer = [
      [lowerJut,startHeight],
      [upperJut,maxHeight],
      [-sideWidth,maxHeight],
      [-sideWidth,startHeight+sideWidth],
      [w+sideWidth,startHeight+sideWidth],
```

```
    [w+sideWidth,maxHeight],
    [w-upperJut,maxHeight],
    [w-lowerJut,startHeight]
  ];

  // перевертаємо порядок вершин у контейнері так,
  // що останні вершини були з лівої частини ігрового світу
  coreContainer.reverse();
  return coreContainer;
},
```

Метод *generateTerrain* проходить по осі X із певним кроком, що задається параметром. На кожному кроці виконується генерація наступної точки із прийнятним збільшенням координати Y (генерація підйому) відносно попередньої точки. При досягненні максимального значення алгоритм переключається на зменшення координати Y (генерація уклону). Крім того, в деяких місцях, коли значення Y є відносно високим, алгоритм додає впадини або круті пагорби. Отже алгоритм генерація поверхні є дуже творчим, тому ви можете поекспериментувати для створення власних ефектів. Але базовий варіант виглядає так:

```
// параметри задають межі, в яких буде згенерований ланшафт
generateTerrain: function(xMin, xMax, yMin, yMax) {
  var points = [];
  // шаг переміщення зліва направо для генерації Y-значень
  stepSize = 32;

  var dif = Math.abs(yMax-yMin);
  // Направлення зміни Y-координати
  // 1 - збільшуємо висоту поверхні
  // 0 - зменшуємо висоту поверхні
  var trend = 1;
  var height = yMin;
  var fissureCount = 0;
  for (var i = xMin ; i <= xMax - stepSize ; i += stepSize) {
    var lowChange = 0;
    var highChange = 0.08;

    // випадкове значення для зміни висоти
    var randChange = this.randomInt(Math.round(dif*lowChange),
        Math.round(dif*highChange));
    var inc = trend*randChange;

    // можлива випадкова зміна напрямку
    var switchDirec = this.randomInt(0,3);

    fissureCount++;
    // зміна напрямку при досягненні yMax
    // та моделювання впадин та пагорбів
    if (height + inc > yMax ||
    (switchDirec == 0 && height+inc > yMin+dif*0.5)) {
        var rand = this.randomInt(0,8);
        if (rand == 0 && fissureCount > 5 && this.cliffs) {
          // генерація пагорба
          trend = 1;
          points.push([i-10,height-dif*0.4]);
```

```
          points.push([i,yMin+dif*0.02]);
          i += stepSize;
          points.push([i,yMin+dif*0.01]);
          i += stepSize;
          points.push([i,yMin]);
          height = yMin;
        }
        else if (rand == 1 && fissureCount > 5 && this.fissures) {
            // генерація впадини
            fissureCount = 0;
            trend = -1;
            points.push([i-10,height-dif*0.4]);
            for (var j = 0 ; j < 5 ; j++) {
              var randVariance = this.randomInt(
                  Math.round(dif*lowChange),Math.round(dif*highChange)
              );
              points.push([i,yMin+randVariance]);
              i+= stepSize;
            }
            points.push([i,yMin]);
            points.push([i+10,height-dif*0.4]);
            points.push([i,height-dif*highChange]);
            height = height-dif*highChange;
        }
        else {
            // нормальний рух
            trend = -1;
            inc = trend*randChange;
            height += inc;
            points.push([i,height]);
        }
    }
    // переключаємо напрямок при досягнення мінімальної висоти
    else if (height + inc < yMin ||
    (switchDirec == 0 && height+inc < yMax-dif*0.5)) {
        trend = 1;
        inc = trend*randChange;
        height += inc;
        points.push([i,height]);
    }
    // продовження в заданому нвпрямку
    else {
        height += inc;
        points.push([i,height]);
    }
  }

  return points;
},
```

Крім того, для функції генерації знадобляться ще дві додаткові функції: *flipTerrain()* перевертає Y-координату вершини відносно вказаних меж, а randomInt генерує число із заданого діапазону:

```
flipTerrain: function(points, yMin, yMax) {
  points.forEach(function(point,index) {
      points[index][1] = yMax - (point[1] - yMin);
  });
  return points;
},
```

```
randomInt: function(min, max) {
  min = Math.ceil(min);
  max = Math.floor(max);
  return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

Тепер перейдемо до методів, які викликаються із *game.js*. Метод *drawOutline()* має такий код:

```
drawOutline: function() {
  var points = this.vertices.flatten();
  var bmd = this.game.add.bitmapData(this.game.world.width,
                                     this.game.world.height);
  var ctx = bmd.context;
  ctx.fillStyle = 'black';
  ctx.lineWidth = 5;
  ctx.beginPath();
  ctx.moveTo(points[0],points[1]);

  // створюємо сплайнову криву, що проходить через сершини
  ctx.curve(points, null, null, true);
  ctx.stroke();

  bmd.addToWorld();
}
```

В коді будується сплайнова крива через вершини поверхні. Для цього нам знадобіться функція *curve()*, яка є відомою і загальнодоступною. Її можна додати до проекту напряму за посиланням https://github.com/pkorac/cardinal-spline-js/blob/master/src/curve.js або завантажити вручну. Ще одним допоміжним методом, що використовується в *drawOutline()* є *Array.prototype.flatten()*, що перетворює заданий масив на однорідний:

```
Array.prototype.flatten = function() {
  var clone = this.slice(0);
  return clone.reduce(function(a, b) {
    return a.concat(b);
  }, []);
}
```

Далі до ігрового світу треба додати фізичне P2-тіло, що й виконує функція *addToWorld()*:

```
addToWorld: function() {
  var groundBody = new Phaser.Physics.P2.Body(this.game,null,0,0);
  var points = this.vertices.slice(0);

  // Конвертуємо вершини із пікселей в метри для движка P2
  // у відповідній орієнтації
  for (var i = 0 ; i < points.length ; i++) {
    points[i][0] = this.game.physics.p2.pxmi(points[i][0]);
    points[i][1] = this.game.physics.p2.pxmi(points[i][1]);
  };

  // додаємо групу опуклих багатокутників до тіла поверхні
  // із масиву - це формує загальний увігнутий багатокутник
  groundBody.data.fromPolygon(points);
  groundBody.debug = this.debug;
```

```
  groundBody.kinematic = true;
  this.game.physics.p2.addBody(groundBody);

  return groundBody;
}
```

Треба зазначити, що коли ми викликаємо нативні P2-методи (не адаптовані під Phaser), то ми повинні перетворити вершини до того формату, який вимагає P2. Фактично це є конвертація пікселів у метри, що робиться за допомогою методу pxmi.

Після цього ми можемо викликати P2-метод *fromPolygon()*. Але зверніть увагу, що ми викликаємо його для *groundBody.data*, а не просто *groundBody*. Різниця в тому, що *groundBody()* – це Phaser-P2-тіло, така собі обгортка для ініціалізації, тоді як оригінальне P2-тіло зберігається в атрибуті *data*.

Метод *fromPolygon()* компонує нашу увігнуту поверхню за допомогою опуклих багатокутників. У підсумку ми додаємо створене тіло до ігрового світу в лівому верхньому куті в точці (0,0). Решта тіла автоматично розміщуються відносно цієї позиції.

Отже на цьому робота закінчена, результат наведений на рис. 7.4.

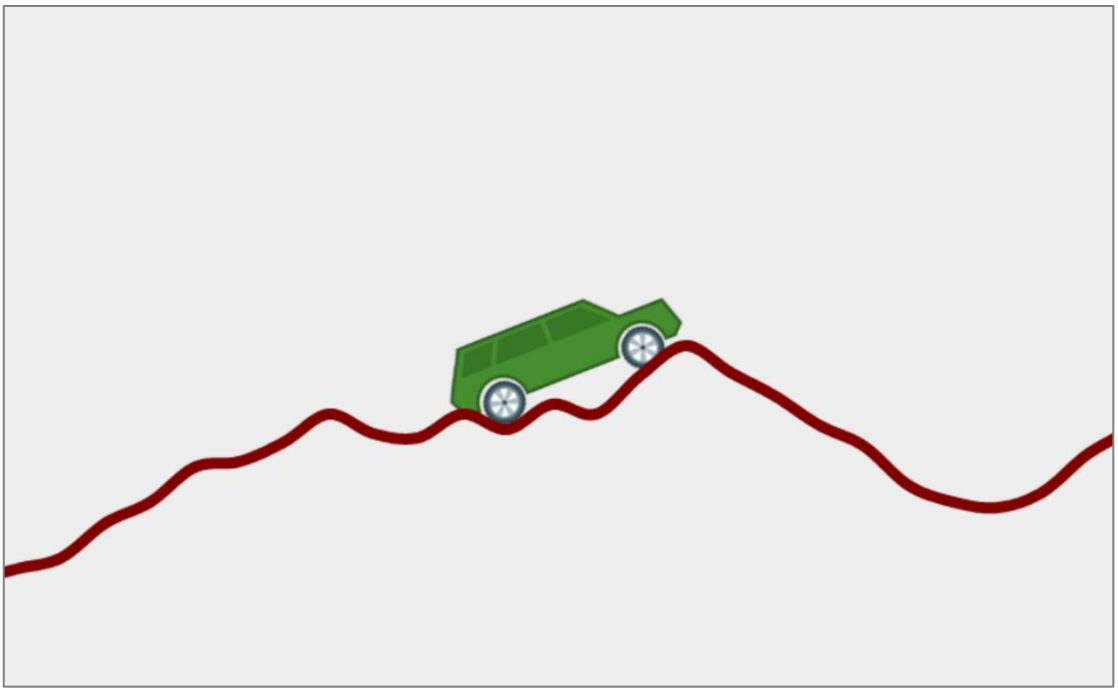

Рисунок 7.5 – Підсумковий вигляд ігрового процесу

Наприкінці зазначимо, що генерація поверхні (або ландшафту взагалі) є дуже цікавою темою в ігровому світі. І результати виконаної лабораторної

роботи можуть бути значно вдосконалені та використані в різних ігрових додатках.

Тепер можна додати розроблений ігровий процес до шаблону додатку, який було створено в лабораторній роботі №2.

Очікувані практичні результати роботи:

- готовий HTML5-додаток із підтримкою усіх типових ігрових станів та ігровим процесом, в якому поверхня генерується довільним чином для кожного запуску гри.

8 Заключний практичний семінар

Заключний практичний семінар орієнтовано на обговорення та тестування створеного студентом при виконанні лабораторних робіт 2-3 кросплатформеного ігрового HTML-додатку за допомогою фреймворку Phaser.

Мета практичного семінару:

- оцінити правильність визначення концепції ігрового додатку та коректності відображення основних ігрових об’єктів;
- оцінити правильність визначення поведінки та взаємодії ігрових об’єктів відповідно до основних принципів використаної ігрової механіки і фізичного движка;
- оцінити як саме були враховані обмеження та особливості платформи HTML5 та порівняти поведінку додатку при запуску в режимах Canvas та WebGL;
- оцінити можливі вдосконалення та зміни, які треба внести до ігрового додатку задля його коректного перетворення в мобільний ігровий додаток за допомогою HTML-to-app;
- оцінити привабливість проекту та надати рекомендації щодо способів подальшого розвитку розробленого мобільного ігрового додатку.

На заключному семінарі кожен студент виступає з презентацією свого ігрового додатку в якій повинен:

1. Надати відомості щодо головної мети та завдань розробки кросплатформеного ігрового додатку.
2. Обґрунтувати основі рішення, прийняті при виконанні лабораторних робіт.

3. Продемонструвати розроблений ігровий додаток як у браузері, так і на мобільному пристрої (завдяки технології HTML-to-app).
4. Визначити основні вади і недоліки розробленого додатку.
5. Запропонувати щонайменше п’ять основних покращень, що можуть бути внесені до проекту.

Крім того доповідач повинен надати можливість тестування розробленого додатку. Викладач довільно призначає студента, який тестує розроблений ігровий додаток і виступає як опонент доповідача, висвітлюючи вади та недоліки розробки.

В ході обговорення студенти групи оцінюють правильність визначення основних графічних елементів ігрового додатку, правильність визначення основних подій та дій об’єктів (ігрової механіки) та визначають рекомендації щодо подальшого розвитку розробленого мобільного ігрового додатку. Результати виступу доповідача та «опонента» оцінюються за критеріями, що наведені в додатку В.

За результатами обговорення на практичному семінарі студенти навчаються оцінювати ігрові мобільні додатки, виявляти недоліки та помилки, що були допущені на етапах розробки графічного інтерфейсу, ігрової механіки та тестування ігрового додатку, визначати подальші шаги розвитку додатку з урахуванням мобільної платформи реалізації та загального результатів обговорення.

Успішне засвоєння матеріалів практичного семінару сприяє формуванню у студента наступних соціальних навичок та вмінь: критично ставитися до результатів особистої роботи, вміти проводити самооцінку виконаних робіт, обґрунтувати той чи інший спосіб діяльності при виконанні практичних завдань, презентувати та аргументувати свої рішення, обговорювати в групі результати роботи. Форми прояву: приймати участь у груповому обговоренні, оцінюванні результатів виконаної роботи, вміння самостійно формулювати та доносити до співрозмовника свої думки, вміння ясно і конкретно висловлювати думки, слухати та розуміти співрозмовника, використовувати прийоми ділового спілкування (публічного мовлення, презентаційних виступів, доповідей) та раніше отримані знання як основу для засвоєння нових знань, дотримуватись етичних норм поведінки, ділового та партнерського спілкування.

9 Заключний звіт. Вимоги

9.1 Структура звіту

У заключний звіт з виконання лабораторних робіт повинні бути включені наступні пункти, що описують ігровий додаток, створений в ЛР № 2-3:

1. Титульна сторінка.
2. Мета роботи.
3. Завдання до лабораторних робіт.
4. Короткі теоретичні відомості.
5. Попередні ескізи загального вигляду ігрового додатку, ескізи спрайтів для усіх ігрових об’єктів.
6. Опис використаного технічного і програмного забезпечення.
7. Опис проекту для користувача із поясненнями основних моментів ігрової механіки.
8. Опис закінченого проекту з зазначенням всіх об’єктів, подій і дій.
9. Пропозиції щодо подальшого розвитку ігрового додатку.
10. Висновки.

9.2 Вимоги до змісту окремих частин заключного звіту

Перш за все, треба за заначити, що звіт повинен представляти собою цільний самостійний документ, який не потребує жодних уточнень та не визиває у читача багато уточнюючих питань.

Мета роботи повинна відображати тему комплексу лабораторних робіт а також конкретні завдання, поставлені студенту на період виконання роботи. За обсягом мета роботи становить від кількох рядків до 0,5 сторінки, але всі заявлені задачі повинні бути повністю виконані.

Наскрізне завдання формується у відповідності до поставлених задач, а також із урахуванням особливостей жанру ігрового додатку, який було реалізовано під час виконання лабораторних робіт.

Короткі теоретичні відомості представляють собою короткий опис архітектури та особливостей фреймворку Phaser. Матеріал розділу не повинен копіювати зміст методичного посібника або підручника з даної теми, а обмежується викладом основних понять, що необхідні для розуміння подальшого змісту звіту. Обсяг цього підрозділу не повинен перевищувати 1/3 частини всього звіту.

Попередні ескізи ігрових елементів наводяться для того, що можна було оцінити відповідність результату та початкового бачення гри її розробником. Треба звернути увагу, що мають бути наведені саме початкові ескізи, без урахування досвіду та змін, що було зроблено в початкових планах вже в період розробки ігрового додатку.

Опис використаного технічного і програмного забезпечення містить дуже стислий огляд усіх інструментальних засобів, що були задіяні і процесі створення ігрового додатку.

Опис проекту для користувача із поясненнями як саме необхідно грати, в чому є основні принципи ігрової механіки, яку мету переслідує та які нагороди отримує користувач. Сам тут буде правильним розмістити скриншоти ключових моментів гри із необхідними поясненнями.

Опис закінченого проекту для розробника повинен містити усі ігрові об’єкти, що присутні у грі. Для кожного об’єкту наводяться його назва, призначення та спосіб використання (або поведінка) у грі. Також необхідно навести усі скрипти із необхідними коментарями.

Пропозиції щодо подальшого розвитку створеного ігрового додатку повинні містити щонайменше п’ять конкретних пропозицій щодо вдосконалення та розширення ігрового додатку.

Висновки. У висновках коротко викладаються результати роботи: отримані результати та навички, здобуті підчас виконання комплексу лабораторних робіт.

Заключний звіт оформлюється відповідно до ДСТУ [8]. Зразок оформлення титульного аркуша звіту наведено в додатку Д.

10 Література

1. Довідник модуля «Розробка кросплатформених ігрових додатків на HTML5» дисципліни «Інструментальна підтримка розробки комп'ютерних ігрових додатків»
2. K. Dyrr, P.F. Navarro, R. Oliveira, B. Sparks. Game Development for Human Beings. Build Cross-Platform Games with Phaser – Zenva Pty Ltd, 2016. – 472 p.
3. Phaser – A fast, fun and free open source HTML5 game framework. – https://phaser.io/.
4. Learn How To Make HTML5 Games.– https://www.discoverphaser.com
5. Phaser Editor – A friendly IDE for HTML5 games creation. – http://phasereditor.boniatillo.com/.
6. How to Learn the Phaser HTML5 Game Engine. – https://gamedevelopment.tutsplus.com/articles/how-to-learn-the-phaser-html5-game-engine--gamedev-13643.
7. Discussion about the Phaser game framework. – http://www.html5gamedevs.com/forum/14-phaser/
8. ДСТУ 3008-95 «Документи. Звіти у сфері науки і техніки. Структура і правила оформлення».

Додаток А. Критерії оцінювання лабораторної роботи

Хід виконання	Самостійність виконання	• Робота виконується повністю самостійно (без допомоги викладача) з елементами інновації, креативності • Робота виконується повністю самостійно • Робота викується з незначною допомогою викладача • Робота не може бути виконана без допомоги викладача	3 2 1 0
	Виконання лабораторної роботи в передбачені терміни	• Робота виконана з випередженням терміну • Робота виконана у відведений термін • Робота виконана із незначним (менше ніж тиждень) порушенням терміну • Робота виконана із значним (більш ніж тиждень) порушенням терміну	3 2 1 0
Результат виконання	Відповідність результату теоретичним посилання	• Знає та використовує методи, способи, технології інтелектуального аналізу та обробки даних, здійснює опрацювання, інтерпретацію та узагальнення отриманих результатів • Знає але не в повній мірі використовує методи, способи, технології інтелектуального аналізу та обробки даних, здійснює опрацювання, інтерпретацію та узагальнення отриманих результатів • Має часткові знання щодо методів, способів, технологій інтелектуального аналізу та обробки даних. Не в повній мірі здійснює опрацювання, інтерпретацію та узагальнення отриманих результатів • Не здійснює опрацювання, інтерпретацію та узагальнення отриманих результатів	3 2 1 0
	Збіг отриманих результатів з тестовими розрахункам	• Отримані результати повністю співпадають з тестовими розрахунками, виконано аналіз трудомісткості отримання результату • Отримані результати повністю співпадають з тестовими розрахунками. • Отримані результати частково співпадають з тестовими розрахунками. • Отримані результати не співпадають з тестовими розрахунками.	3 2 1 0

Підсумкова оцінка виконання лабораторної роботи

12 – 10	Перевищує вимоги	4
9 – 8	Відповідає вимогам	3
7 – 5	Переважно відповідає вимогам	2
4 – 3	Майже відповідає вимогам	1
2 – 0	Не відповідає вимогам	0

Додаток Б. Критерії оцінювання презентації результатів на заключному практичному семінарі

Студент пояснює процес та висновки проекту та демонструє отримані знання.	Визначення теми	• Чітко визначає тему чи основні питання, їх значення. • Чітко визначає тему. • Визначає тему. • Не визначає тему.	3 2 1 0
	Доказовість	• Підтверджує достовірність основних результатів шляхом аналізу відповідних та точних доказів • Підтверджує достовірність основних результатів шляхом використання необхідних доказів • Підтверджує достовірність результатів шляхом використання мінімально необхідних доказів • Не підтверджує результати доказами	3 2 1 0
	Джерела	• Використовує технології та інструментарії пошукових систем. Надає докази обширного та достовірного дослідження з використанням численних та різноманітних джерел. • Не використовує технології та інструментарії пошукових систем. Надає докази достовірності дослідження за різноманітними джерелами. • Надає докази достовірності дослідження за рядом джерел. • Надає незначну кількість доказів дослідження (або не надає взагалі) за сторонніми джерелами.	3 2 1 0
	Вирішення проблеми	• Генерує нові ідеї вирішення складної проблеми та виходить з поясненнями за межі навчання. • Надає різноманітні свідчення вирішення складної проблеми та виходить з поясненнями за межі навчання. • Надає деякі свідчення вирішення проблеми. • Надає мало свідчень вирішення проблеми та виходу за межі навчання.	3 2 1 0

	Ідейна база	• Поєднує і оцінює існуючі ідеї для формування нових поглядів. • Поєднує існуючі ідеї для формування нових поглядів. • Поєднує існуючі ідеї. • Незначною мірою поєднує існуючі ідеї.	3 2 1 0
Організація матеріалу презентації Студент демонструє логічність та організованість	Представлення теми	• Представляє тему чітко та творчо • Представляє тему чітко • Представляє тему • Не представляє тему	3 2 1 0
	Утримання фокусу на темі	• Підтримує чіткий фокус на темі. • Підтримує фокус на темі. • Деякою мірою підтримує фокус на темі. • Не підтримує фокус на темі.	3 2 1 0
	Використання переходів	• Ефективно використовує поступові переходи для поєднання ключових моментів. • Використовує поступові переходи для поєднання ключових моментів. • Використовує деякі переходи для поєднання ключових моментів. • Використовує деякі переходи, що рідко дозволяють поєднати ключові моменти	3 2 1 0
	Висновки	• Презентація закінчується логічним, ефективним і відповідним висновком. • Презентація закінчується відповідним очевидним висновком • Презентація закінчується очевидним висновком. • Презентація закінчується без висновку	3 2 1 0
Питання й відповіді	Представлення теми	• Має систематичні знання в досліджуваній тематиці, демонструє вміння осмислювати тему і робити обґрунтовані висновки. Точно та належним чином на всі запитання аудиторії • Показує широкі знання теми, відповідаючи впевнено, точно та належним чином на всі запитання та зауваження аудиторії. • Демонструє знання теми, відповідаючи точно та відповідним чином на питання та відгуки.	3 2 1

		• Демонструє неповне знання теми, відповідаючи неточно і невідповідно на питання та відгуки.	0
Використання мови та стиль вираження себе в усній формі Студент ефективно передає свої ідеї	Зоровий контакт	• Ефективно використовує зоровий контакт. • Підтримує зоровий контакт. • Є деякий зоровий контакт, але він не підтримується. • Неефективний зоровий контакт.	3 2 1 0
	Мова	• Говорить чітко, по суті і впевнено, використовуючи правильні обсяг і темп • Говорить чітко, використовуючи правильні обсяг і темп • Мова частково чітка, частково нечітка • Мова нечітка, не підходящий темп	3 2 1 0
	Спілкування	• Демонструє знання граматичних, стилістичних особливостей лексики, термінології, лексичних структур. Використовує типові для тематичної комунікації лексико-синтаксичні моделі. • Демонструє знання граматичних, стилістичних особливостей лексики, термінології, лексичних структур. Не використовує типові для тематичної комунікації лексико-синтаксичні моделі. • Частково відсутні знання граматичних, стилістичних особливостей лексики, термінології. • Відсутні знання граматичних, стилістичних особливостей лексики, термінології.	3 2 1 0
	Залучення аудиторії	• Повністю залучає аудиторію. • Старається залучити аудиторію. • Випадкове залучення аудиторії. • Аудиторія не залучається.	3 2 1 0
	Одяг	• Одягнений належним чином • Одягнений неналежним чином	1 0

Підсумкова оцінка презентації та виступу на заключному практичному семінарі

Сумарний бал		Зарахований бал
44 – 35	Перевищує вимоги	4
34 – 27	Відповідає вимогам	3
26 – 20	Частково відповідає вимогам	2
19 – 10	Майже відповідає вимогам	1
9 – 0	Не відповідає вимогам	0

Додаток В. Критерії оцінювання заключного звіту

Завдання роботи	Формулювання завдання роботи	• Завдання на роботу сформульовано докладно, з можливістю перевірки зрозуміння виконавцем • Завдання на роботу сформульовано, можливість перевірки розуміння утруднена • Завдання на роботу сформульовано частково, перевірка розуміння не можлива • Завдання на роботу не сформульовані, перевірка розуміння не можлива	3 2 1 0
	Очікувані результати	• Очікувані результати сформульовані повністю, дозволяють легке порівняння з отриманими результатами, наведені повні тестові розрахунки • Очікувані результати сформульовані повністю, наведені частково тестові розрахунки, що не дозволяє легкого порівняння з отриманими результатами • Очікувані результати сформульовані частково, наведені частково тестові розрахунки, що не дозволяє порівняння з отриманими результатами • Очікувані результати не сформульовані, тестові розрахунки не наведені	3 2 1 0
Опис процедури (процесу) виконання роботи	Опис устаткування	• Наведено докладний перелік та опис використаного устаткування (комп'ютери, засоби виводу графічної інформації, комунікаційне обладнання, тощо) • Наведено перелік та частковий опис використаного устаткування (комп'ютери, засоби виводу графічної інформації, комунікаційне обладнання, тощо) • Наведено частковий перелік використаного устаткування (комп'ютери, засоби виводу графічної інформації, комунікаційне обладнання, тощо) • Перелік та опис використаного устаткування (комп'ютери, засоби виводу графічної інформації, комунікаційне обладнання, тощо) відсутній	3 2 1 0
	Опис використаного програмного продукту	• Наведено докладний перелік та опис використаного стандартного програмного продукту (операційна система, середа розробки, прикладні пакети, тощо) • Наведено перелік та частковий опис використаного стандартного програмного продукту (операційна система, середа розробки, прикладні пакети, тощо) • Наведено частковий перелік використаного стандартного програмного продукту (операційна система, середа розробки, прикладні пакети, тощо)	3 2 1 0

		• Перелік використаного стандартного програмного продукту (операційна система, середа розробки, прикладні пакети, тощо) не наведено	
	Процедура виконання	• Усі кроки докладно перераховані та легко можуть бути повторені • Усі кроки докладно перераховані, але не можуть бути легко повторені • Не всі кроки перераховані і не можуть бути легко повторені • Процедури виконання не приведена	3 2 1 0
	Наведено опис розроблених за завданням програм (алгоритмів)	• Наведено блок-схеми та докладний текстовий опис розроблених програм (алгоритмів) • Наведено блок-схеми та частковий текстовий опис розроблених програм (алгоритмів) • Наведено блок-схеми розроблених програм (алгоритмів). Текстовий опис розроблених алгоритмів не наведено. • Опис розроблених за завданням прогам та алгоритмів не наведено	3 2 1 0
	Наведено опис отриманих за завданням результаті	• Наведені отримані результати та їх порівняння очікуваними результатами та з тестовими розрахунками • Наведені отримані результати. Їх порівняння з очікуваними результатами та з тестовими розрахунками не наведено. • Відсутній опис отриманих результатів	3 2 0
Заключна частина	Висновки	• Обґрунтовано підтверджуються теоретичні положення, підтверджена правильність роботи програми, алгоритму • Теоретичні положення підтверджуються частково, правильність роботи алгоритму (програми) підтверджена частково • Теоретичні положення , правильність роботи алгоритму (програми) не підтверджені • Відсутні висновки за роботою	3 2 1 0
	Подальший розвиток за напрямом роботи	• Надано розширений опис шляхів удосконалення запропонованих при виконанні роботи рішень (процедур, алгоритмів, програм), представлені напрямки подальшого розвитку підходів щодо рішень поставленої проблеми. • Надано опис шляхів удосконалення запропонованих при виконанні роботи рішень (процедур, алгоритмів, програм), представлені напрямки подальшого розвитку підходів щодо рішень	3 2

		поставленої проблеми. • Надано опис шляхів удосконалення запропонованих при виконанні роботи рішень (процедур, алгоритмів, програм), напрямки подальшого розвитку підходів щодо рішень поставленої проблеми не представлені. • Шляхи удосконалення запропонованих при виконанні роботи рішень не запропоновані, напрямки подальшого розвитку підходів щодо рішень поставленої проблеми не представлені.	1 0
Звіт як технічний текст	Містить: назву, прізвище студента, прізвище викладача, дату	• Немає відсутніх компонентів • Відсутня 1 компонента • Відсутні 2 -4 компоненти • Відсутні більш ніж 4 компоненти	3 2 1 0
	Акуратно оформлена	• Звіт оформлений акуратно • Звіт оформлено • Звіт оформлено не акуратно	2 1 0
	Помилки	• Помилки відсутні • Невелика кількість граматичних та синтаксичних помилок (на сторінці не більш ніж 1 помилка) • Велика кількість граматичних та синтаксичних помилок (кількість помилок більше ніж кількість сторінок звіту)	2 1 0
	Технічні аспекти	• Відсутні помилки в пунктуації, використанні заголовних букв і орфографії. • Майже немає помилок в пунктуації, використанні заголовних букв і орфографії. • Багато помилок в пунктуації, використанні заголовних букв і орфографії. • Численні помилки в пунктуації, використанні заголовних букв і орфографії, що не дають можливості зрозуміти думку автора	3 2 1 0
	Використання слів	• Речення побудовані без помилок, всі слова використовуються вірно. • Майже немає помилок в структурі речень і використанні слів.	3 2 1

		• Багато помилок в побудові структури речень структури і використанні слів. • Численні і відволікаючі помилки в структурі речень і використанні слова	0

Підсумкова оцінка заключного звіту

Сумарний бал		Зарахований бал
40 – 36	Перевищує вимоги	4
35 – 30	Відповідає вимогам	3
29 – 25	Відповідає вимогам із зауваженнями	2
24 – 15	Частково відповідає вимогам	1
14 – 0	Не відповідає вимогам	0

Додаток Д. Приклад оформлення титульної сторінки заключного звіту.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Кафедра прикладної математики та інформатики

ЗАКЛЮЧНИЙ ЗВІТ
з виконання комплексу лабораторних робіт
з дисципліни:
«Інструментальна підтримка розробки комп'ютерних ігрових додатків.
Розробка кросплатформених ігрових додатків на HTML5»

Виконав студент гр. ІПЗ 14
_____ Іванов І.І.
«____»_____2017 р.

Прийняв_____
Викладач кафедри ПМІ
_____ Петров В.О.
«____»_____2017 р.

Покровськ 2018

Навчальне видання

Навчальний модуль «Розробка ігрових додатків для ОС Android» з дисципліни «Інструментальна підтримка розробки комп'ютерних ігрових додатків» підготовки студентів освітнього рівня «магістр» спеціальності 121 «Інженерія програмного забезпечення»

Укладачі: Цололо Сергій Олексійович
Дікова Юлія Леонідівна