**Міністерство освіти і науки України**

**Національний технічний університет України «КПІ» імені Ігоря Сікорського**

**Кафедра обчислювальної техніки ФІОТ**

**ЗВІТ**

**з лабораторної роботи № 9**

**з навчальної дисципліни «Computer Vision»**

**Тема:**

**СИНТЕЗ РЕАЛІСТИЧНИХ ОБ’ЄКТІВ ДОПОВНЕНОЇ РЕАЛЬНОСТІ**

**Виконав:**

Студент 3 курсу кафедри ІПІ ФІОТ,

Навчальної групи ІП-14

Бабіч Д. В.

**Перевірив:**

Професор кафедри ОТ ФІОТ

Писарчук О. О.

**Київ 2024**

1. **Мета:**

Дослідити методологію і технології створення доповненої реальності.

1. **Завдання:**

**Завдання ІІ**

Таблиця 1.1 – Варіант завдання

| **Варіант**  (місяць народження) | **Технічні умови** |
| --- | --- |
| 9 | Для формування модельного Dataset з метою навчання нейромережі для розпізнавання заданих об’єктів за технологіями Computer Vision створити динамічну модель руху легкових автомобілів. |

1. **Результати виконання лабораторної роботи.**
2. **Синтезована математична модель перетворень об’єктів у 3Д просторі.**

Для виконання обертання побудованого 3Д об’єкту необхідно було використати матриці трансформацій. Матриці трансформацій використовуються для виконання різних видів перетворень у просторі, таких як обертання, масштабування та перенесення. Вони є основою для багатьох операцій в графіці та комп’ютерному моделюванні. Також ці матриці можуть бути комбіновані для виконання складених перетворень. Наприклад, можна спочатку обернути об’єкт, потім масштабувати його, а потім перенести його до нового місця. Комбінація цих перетворень може бути представлена однією матрицею, яка є результатом множення окремих матриць перетворень, проте у цьому випадку особливу увагу потрібно зважати на порядок, у якому виконуються перетворення, має значення, оскільки множення матриць не є комутативним.

Матриця обертання, довкола осі X, де θ – заданий кут обертання.

|  | 1  0  0 | 0  cos(θ)  sin(θ) | 0  -sin(θ)  cos(θ) |  |
| --- | --- | --- | --- | --- |

Матриця обертання, довкола осі Y, де θ – заданий кут обертання.

|  | cos(θ)  0  -sin(θ) | 0  1  0 | sin(θ)  0  cos(θ) |  |
| --- | --- | --- | --- | --- |

Матриця обертання, довкола осі Z, де θ – заданий кут обертання.

|  | cos(θ)  sin(θ)  0 | -sin(θ)  cos(θ)  0 | 0  0  1 |  |
| --- | --- | --- | --- | --- |

У якості абстракції представлення обертання об’єктів у зручний для людини вигляд були використані кути Ейлера. Кути Ейлера – це три кути, які визначають орієнтацію об'єкта в тривимірному просторі. Вони включають кутові величини, які відображають обертання навколо трьох взаємно перпендикулярних осей.

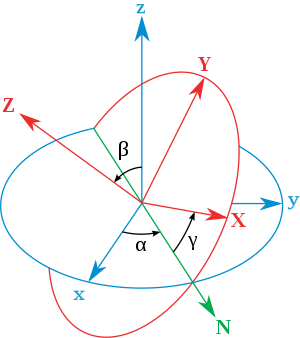
****

Рисунок 1.1 – Представлення кутів Ейлера

Для роботи з 3Д моделями був створений відповідний завантажувач моделей, який підтримує .obj формат. Формат OBJ (Wavefront OBJ) є текстовим форматом файлу, що використовується для зберігання геометричної інформації про 3D-моделі. Він може містити дані про вершини (координати точок у тривимірному просторі), текстурні координати (які використовуються для нанесення текстур на поверхні об'єкта), нормалі (орієнтація поверхонь у просторі), а також інформацію про полігони (трикутники або чотирикутники, що складають модель). Він зазвичай складається з набору ключових слів та числових значень, що представляють різні аспекти моделі. Наприклад, він може включати рядки, що описують вершини, текстурні координати, нормалі та полігони, розділені відповідними ключовими словами, такими як "v" для вершин, "vt" для текстурних координат, "vn" для нормалей та "f" для полігонів та інформації про них.

1. **Результати архітектурного проектування та їх опис.**

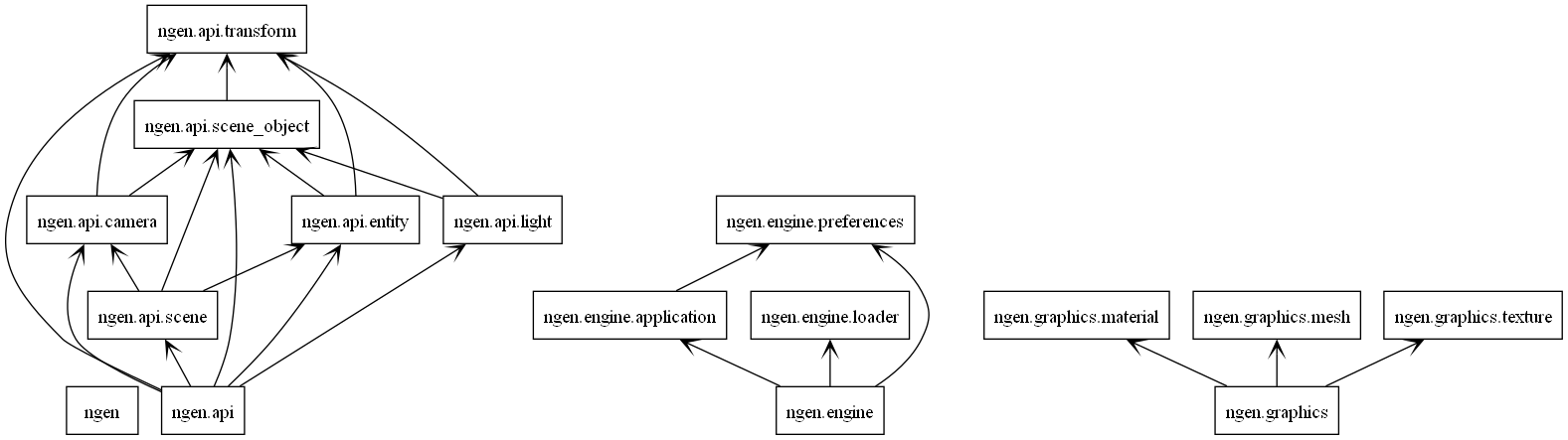
****

Рисунок 1.2 – Архітектура рушія

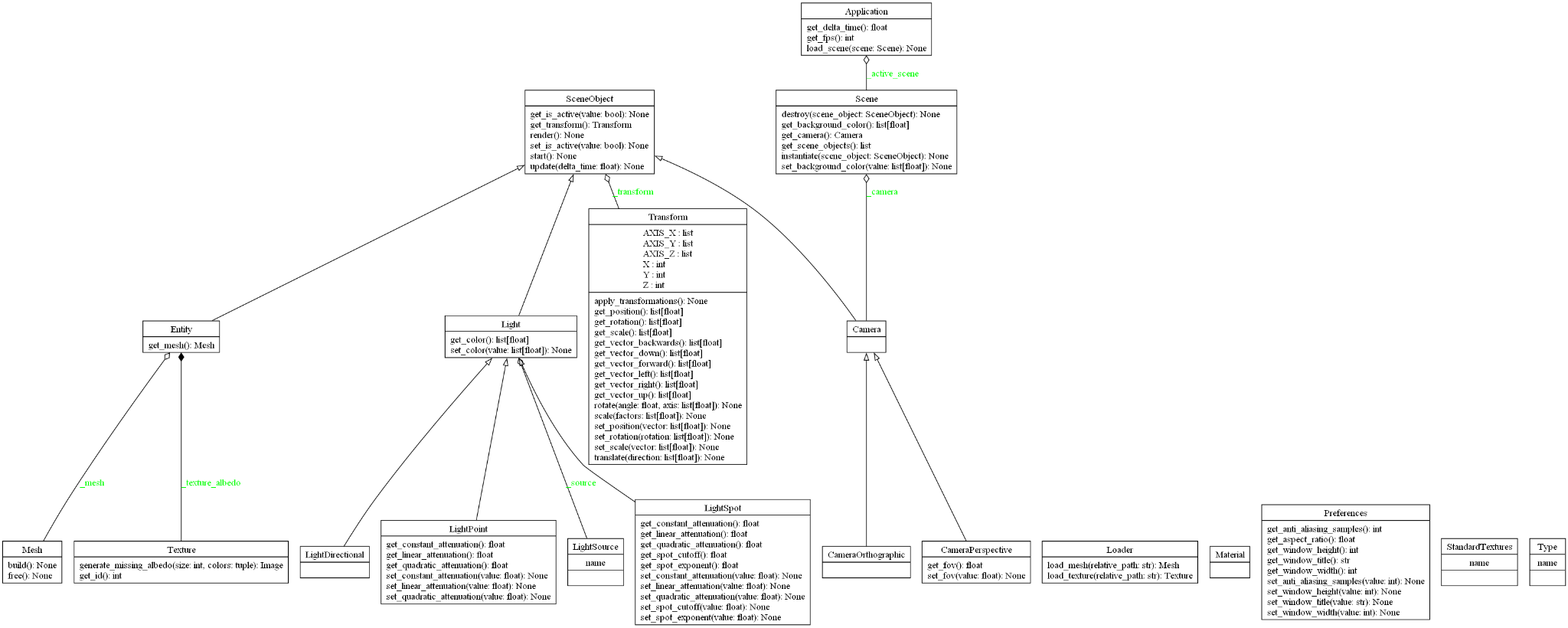


Рисунок 1.3 – UML-діаграма класів

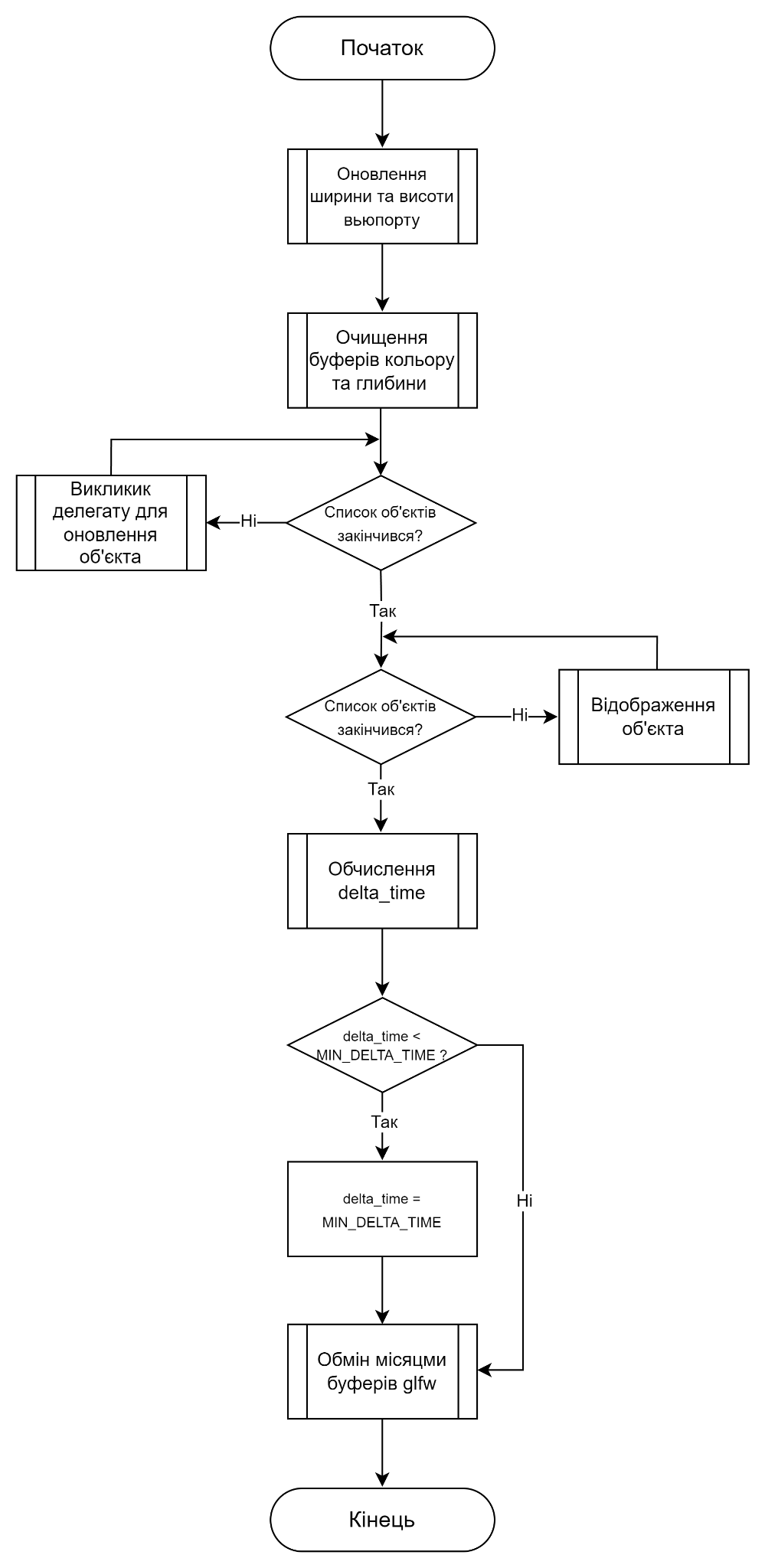


Рисунок 1.4 – Діаграма головного рантайм-циклу застосунку

У наведеній діаграмі класів позначені основні приватні та публічні члени, за допомогою яких і функціонує створене програмне забезпечення. Особливу увагу варто звернути на клас SceneObject, який інкапсулює представлення кожного об’єкту у контексті віртуальної сцени у OpenGL. Цей клас має створені об’єкт класу Transform, який інкапсулює логіку представлення об’єкта у контексті простору 3Д сцени та має збережені значення позиції, розмірів та обертання об’єкту. Оновлена інформація про положення об’єкта переводиться у зручний формат за допомогою кутів Ейлера.

Клас SceneObject – є базовим класом для всіх створюваних об’єктів на сцені, тому він зберігає створений об’єкт класу Transform та через конструктор надає своїм нащадкам можливість передати делегати на update (методі, який викликатиметься кожен кадр), render (метод, який відповідає за відображення об’єкта).

Клас Entity – є класом, який відповідає за представлення будь-якого 3Д об’єкта, який можна завантажити за допомогою Loader, так і альбедо-текстури до нього.

Такі класи як різні типи джерел світла, камер і мають на меті представлення цих типів об’єктів з особливостями їх реалізації стосовно підходу до відображення у вьюпорті, по-кадрового оновлення, тощо.

Клас Application надає доступ до самого вікна вьюпорту, у якому і відбувається відображення всіх 3Д об’єктів, джерел світла.

Клас Preferences надає доступ рівня класу до налаштувань застосунку, де можна встановити бажай рівень згладжування, розмір вікна та інші параметри.

Клас Scene фактично є реалізацією такої собі колекції всіх об’єктів на сцені, через який можна отримати легкий доступ до такого функціоналу, як додавання об’єктів, їх видалення та інші дії.

Клас Mesh інкапсулює представлення за допомогою списку вершин та поверхонь об’єкта і за кожен умовний draw call виконує відображення 3Д об’єкта за допомогою побудови завчасно скомпільованого мешу, який зберігається у VRAM відеочіпу та створюється у конструкторі екземпляру.

Клас Texture створений для представлення текстур об’єкту, екземпляри створюються подібно до мешів через клас Loader.

Клас Loader має 2 статичні методи, які відповідають за парсинг .obj-файлу для створення нових мешів і завантаження текстур зі звичайних файлів із зображеннями.

Окремо варто зазначити логіку делегату update\_delegate з SceneObject, який є делегатом, який приймає методи зазначеної сигнатури, які викликаються кожен оновлений кадр, тим самим для кожного об’єкта у сцені можуть виконуватися дії зазначені у цьому методі.

1. **Опис структури проекту програми.**

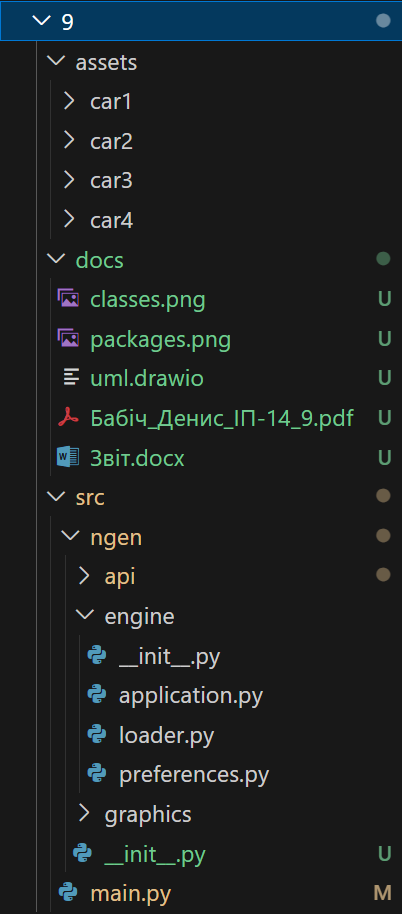


Рисунок 1.5 – Структура проєкту

У директорії src зберігається main.py, який і є модулем з вихідним кодом, яка зберігає функцію main, директорія ngen – зберігає вихідний код рушія, директорія docs – зберігає файли звіту у форматі pdf, docx та drawio – файл з діаграмами та .png файли з діаграмами, assets – зберігає 3Д моделі машин та їх альбедо текстури.

1. **Результати роботи програми відповідно до завдання.**

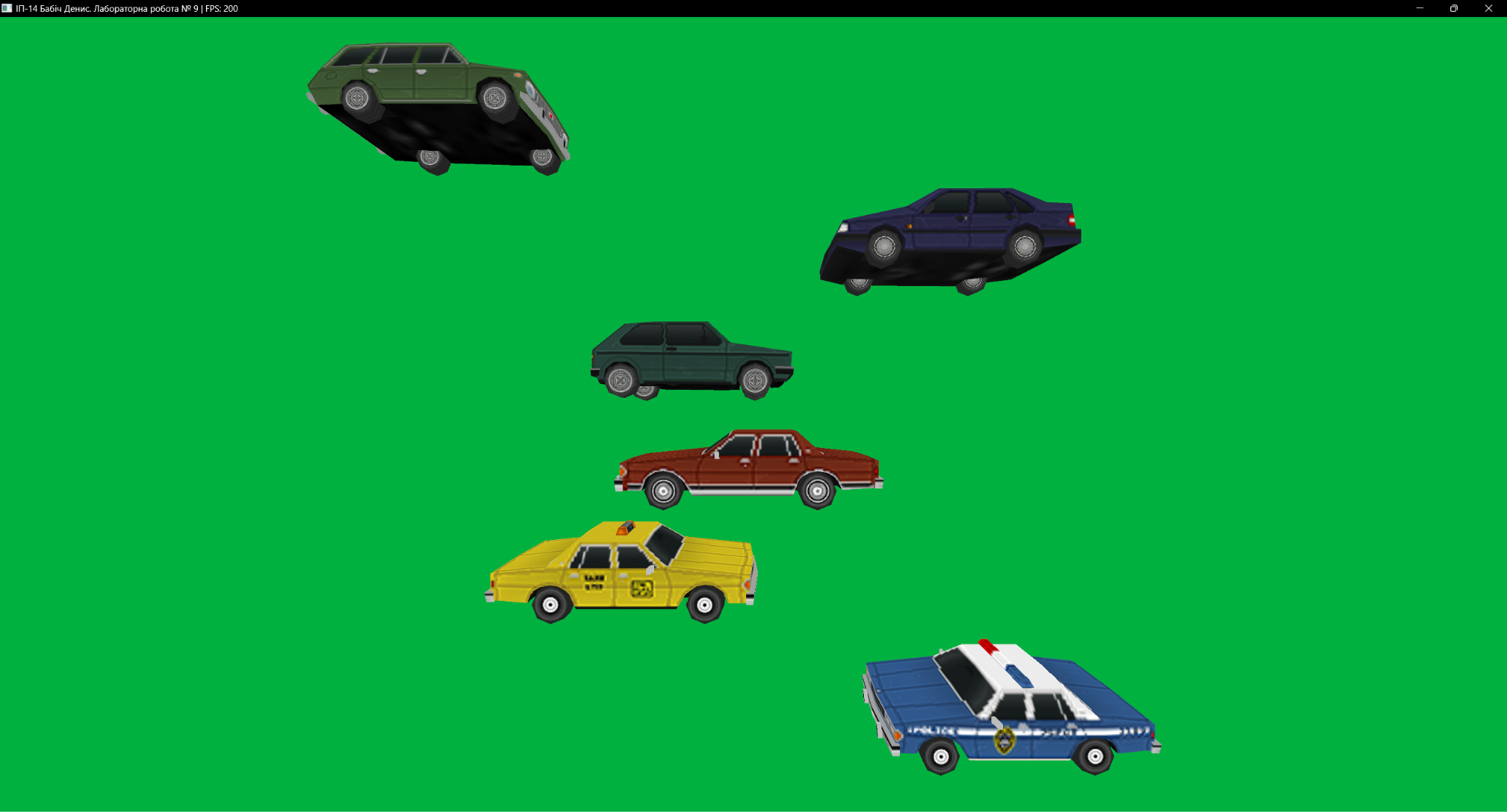


Рисунок 1.6 – Робота створеного застосунку

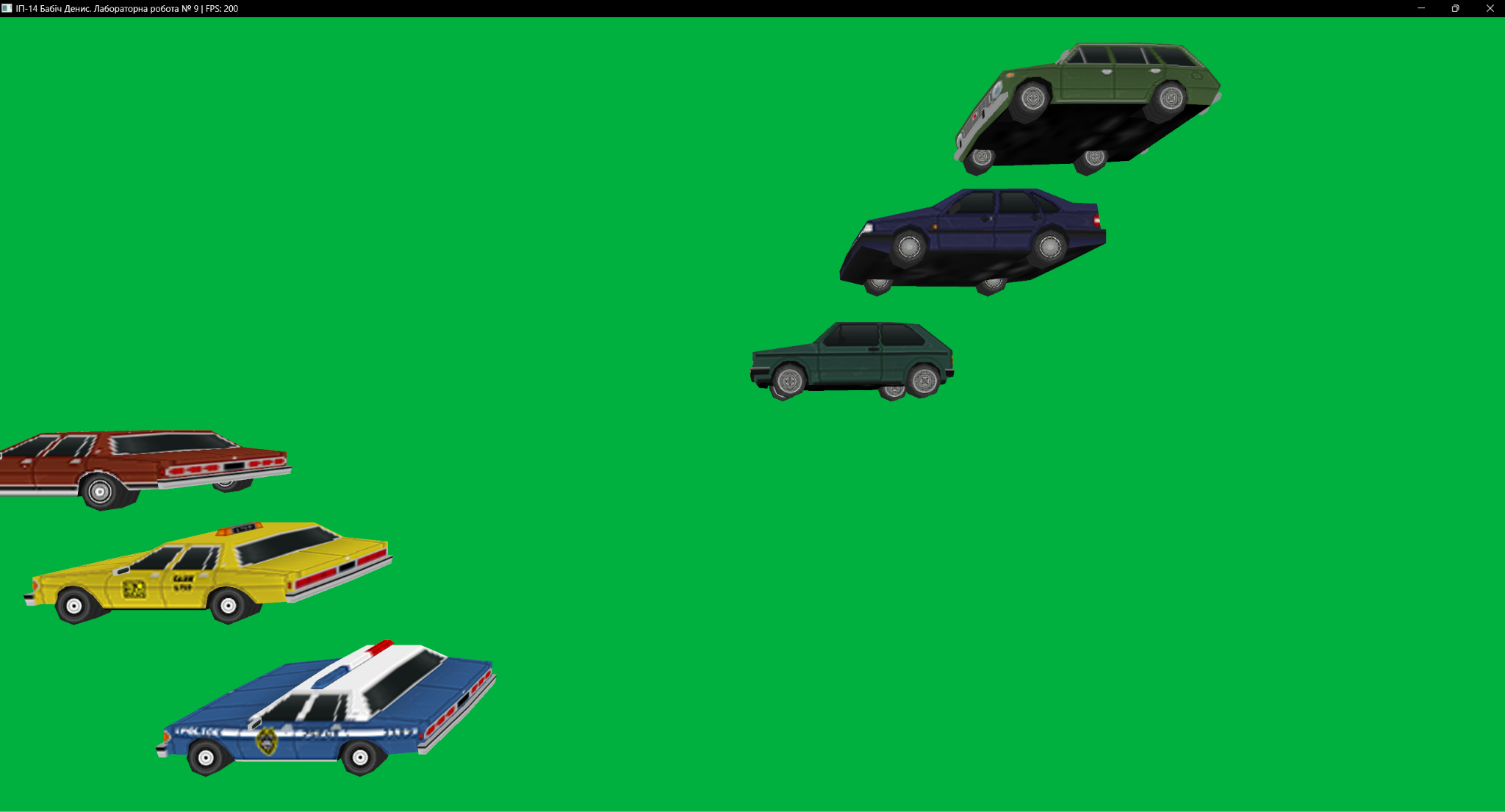


Рисунок 1.7 – Демонстрація руху машин

1. **Програмний код, що забезпечує отримання результату.**

from OpenGL.GL import \*

from OpenGL.GLU import \*

from typing import Callable

from .transform import Transform

from .scene\_object import SceneObject

from ..engine.preferences import Preferences

class Camera(SceneObject):

def \_\_init\_\_(self, transform: Transform, clipping\_plane\_near: float, clipping\_plane\_far: float, render\_delegate: Callable[[], None], start\_delegate: Callable[['Camera'], None] = None, \*update\_delegates: Callable[['Camera', float], None]) -> None:

super().\_\_init\_\_(transform, self.\_render, start\_delegate, \*update\_delegates)

self.\_clipping\_plane\_far = clipping\_plane\_far

self.\_clipping\_plane\_near = clipping\_plane\_near

self.\_camera\_render\_delegate = render\_delegate

def \_render(self) -> None:

glMatrixMode(GL\_PROJECTION)

glLoadIdentity()

self.\_camera\_render\_delegate()

gluLookAt(\*self.\_transform.get\_position(), \*[(position + forward) for position, forward in zip(self.\_transform.\_position, self.\_transform.get\_vector\_forward())], \*self.\_transform.get\_vector\_up())

class CameraPerspective(Camera):

def \_\_init\_\_(self, transform: Transform, fov: float, clipping\_plane\_near: float, clipping\_plane\_far: float, start\_delegate: Callable[['CameraPerspective'], None] = None, \*update\_delegates: Callable[['CameraPerspective', float], None]) -> None:

super().\_\_init\_\_(transform, clipping\_plane\_near, clipping\_plane\_far, self.\_render\_perspective\_camera, start\_delegate, \*update\_delegates)

self.\_fov = fov

def get\_fov(self) -> float:

return self.\_fov

def set\_fov(self, value: float) -> None:

self.\_fov = value

def \_render\_perspective\_camera(self) -> None:

gluPerspective(self.\_fov, Preferences.get\_aspect\_ratio(), self.\_clipping\_plane\_near, self.\_clipping\_plane\_far)

class CameraOrthographic(Camera):

def \_\_init\_\_(self, transform: Transform, clipping\_plane\_near: float, clipping\_plane\_far: float, start\_delegate: Callable[['CameraOrthographic'], None] = None, \*update\_delegates: Callable[['CameraOrthographic', float], None]) -> None:

super().\_\_init\_\_(transform, clipping\_plane\_near, clipping\_plane\_far, self.\_render\_orthographic\_camera, start\_delegate, \*update\_delegates)

def \_render\_orthographic\_camera(self) -> None:

VIEWPORT\_CENTER\_WIDTH = Preferences.get\_window\_width() / 2.0

VIEWPORT\_CENTER\_HEIGHT = Preferences.get\_window\_height() / 2.0

glOrtho(-VIEWPORT\_CENTER\_WIDTH, VIEWPORT\_CENTER\_WIDTH, -VIEWPORT\_CENTER\_HEIGHT, VIEWPORT\_CENTER\_HEIGHT, self.\_clipping\_plane\_near, self.\_clipping\_plane\_far)

from OpenGL.GL import \*

from typing import Callable

from ..graphics.mesh import Mesh

from .transform import Transform

from .scene\_object import SceneObject

from ..graphics.texture import Texture, StandardTextures

class Entity(SceneObject):

def \_\_init\_\_(self, transform: Transform, mesh: Mesh, texture\_albedo: Texture = None, start\_delegate: Callable[['Entity'], None] = None, \*update\_delegates: Callable[['Entity', float], None]) -> None:

super().\_\_init\_\_(transform, self.\_render, start\_delegate, \*update\_delegates)

self.\_mesh = mesh

self.\_texture\_albedo = texture\_albedo if texture\_albedo is not None else Texture(StandardTextures.MISSING\_ALBEDO.value)

def get\_mesh(self) -> Mesh:

return self.\_mesh

def \_render(self) -> None:

glMatrixMode(GL\_MODELVIEW)

glLoadIdentity()

glPolygonMode(GL\_FRONT, GL\_FILL)

glBindTexture(GL\_TEXTURE\_2D, self.\_texture\_albedo.get\_id())

self.\_transform.apply\_transformations()

self.\_mesh.build()

glBindTexture(GL\_TEXTURE\_2D, 0)

from enum import Enum

from OpenGL.GL import \*

from typing import Callable

from .transform import Transform

from .scene\_object import SceneObject

class LightSource(Enum):

LIGHT0 = GL\_LIGHT0

LIGHT1 = GL\_LIGHT1

LIGHT2 = GL\_LIGHT2

LIGHT3 = GL\_LIGHT3

LIGHT4 = GL\_LIGHT4

LIGHT5 = GL\_LIGHT5

LIGHT6 = GL\_LIGHT6

LIGHT7 = GL\_LIGHT7

class Light(SceneObject):

class Type(Enum):

SPOT = 2.0

POINT = 1.0

DIRECTIONAL = 0.0

def \_\_init\_\_(self, transform: Transform, type: Type, source: LightSource, color: list[float], render\_delegate: Callable[[], None] = None, start\_delegate: Callable[['Light'], None] = None, \*update\_delegates: Callable[['Light', float], None]) -> None:

super().\_\_init\_\_(transform, self.\_render, start\_delegate, \*update\_delegates)

self.\_type = type

self.\_color = color

self.\_source = source

self.\_render\_delegate\_light = render\_delegate

def get\_color(self) -> list[float]:

return self.\_color

def set\_color(self, value: list[float]) -> None:

self.\_color = value

def \_render(self) -> None:

glMatrixMode(GL\_MODELVIEW)

glLoadIdentity()

glLightfv(self.\_source.value, GL\_POSITION, [\*self.\_transform.\_position, self.\_type.value])

glLightfv(self.\_source.value, GL\_AMBIENT, self.\_color)

glLightfv(self.\_source.value, GL\_DIFFUSE, self.\_color)

glLightfv(self.\_source.value, GL\_SPECULAR, self.\_color)

if self.\_render\_delegate\_light is not None:

self.\_render\_delegate\_light()

glEnable(self.\_source.value)

class LightDirectional(Light):

def \_\_init\_\_(self, transform: Transform, source: LightSource, color: list[float], start\_delegate: Callable[['LightDirectional'], None] = None, \*update\_delegates: Callable[['LightDirectional', float], None]) -> None:

super().\_\_init\_\_(transform, Light.Type.DIRECTIONAL, source, color, None, start\_delegate, \*update\_delegates)

class LightPoint(Light):

def \_\_init\_\_(self, transform: Transform, source: LightSource, color: list[float], linear\_attenuation: float, constant\_attenuation: float, quadratic\_attenuation: float, start\_delegate: Callable[['LightPoint'], None] = None, \*update\_delegates: Callable[['LightPoint', float], None]) -> None:

super().\_\_init\_\_(transform, Light.Type.POINT, source, color, self.\_render\_point\_light, start\_delegate, \*update\_delegates)

self.\_linear\_attenuation = linear\_attenuation

self.\_constant\_attenuation = constant\_attenuation

self.\_quadratic\_attenuation = quadratic\_attenuation

def get\_constant\_attenuation(self) -> float:

return self.\_constant\_attenuation

def set\_constant\_attenuation(self, value: float) -> None:

self.\_constant\_attenuation = value

def get\_linear\_attenuation(self) -> float:

return self.\_linear\_attenuation

def set\_linear\_attenuation(self, value: float) -> None:

self.\_linear\_attenuation = value

def get\_quadratic\_attenuation(self) -> float:

return self.\_quadratic\_attenuation

def set\_quadratic\_attenuation(self, value: float) -> None:

self.\_quadratic\_attenuation = value

def \_render\_point\_light(self) -> None:

glLightf(self.\_source.value, GL\_LINEAR\_ATTENUATION, self.\_linear\_attenuation)

glLightf(self.\_source.value, GL\_CONSTANT\_ATTENUATION, self.\_constant\_attenuation)

glLightf(self.\_source.value, GL\_QUADRATIC\_ATTENUATION, self.\_quadratic\_attenuation)

class LightSpot(Light):

def \_\_init\_\_(self, transform: Transform, source: LightSource, color: list[float], spot\_cutoff: float, spot\_exponent: float, linear\_attenuation: float, constant\_attenuation: float, quadratic\_attenuation: float, start\_delegate: Callable[['LightSpot'], None] = None, \*update\_delegates: Callable[['LightSpot', float], None]) -> None:

super().\_\_init\_\_(transform, Light.Type.SPOT, source, color, self.\_render\_spot\_light, start\_delegate, \*update\_delegates)

self.\_spot\_cutoff = spot\_cutoff

self.\_spot\_exponent = spot\_exponent

self.\_linear\_attenuation = linear\_attenuation

self.\_constant\_attenuation = constant\_attenuation

self.\_quadratic\_attenuation = quadratic\_attenuation

def get\_constant\_attenuation(self) -> float:

return self.\_constant\_attenuation

def set\_constant\_attenuation(self, value: float) -> None:

self.\_constant\_attenuation = value

def get\_linear\_attenuation(self) -> float:

return self.\_linear\_attenuation

def set\_linear\_attenuation(self, value: float) -> None:

self.\_linear\_attenuation = value

def get\_quadratic\_attenuation(self) -> float:

return self.\_quadratic\_attenuation

def set\_quadratic\_attenuation(self, value: float) -> None:

self.\_quadratic\_attenuation = value

def get\_spot\_cutoff(self) -> float:

return self.\_spot\_cutoff

def set\_spot\_cutoff(self, value: float) -> None:

self.\_spot\_cutoff = value

def get\_spot\_exponent(self) -> float:

return self.\_spot\_exponent

def set\_spot\_exponent(self, value: float) -> None:

self.\_spot\_exponent = value

def \_render\_spot\_light(self) -> None:

glLightf(self.\_source.value, GL\_LINEAR\_ATTENUATION, self.\_linear\_attenuation)

glLightf(self.\_source.value, GL\_CONSTANT\_ATTENUATION, self.\_constant\_attenuation)

glLightf(self.\_source.value, GL\_QUADRATIC\_ATTENUATION, self.\_quadratic\_attenuation)

glLightf(self.\_source.value, GL\_SPOT\_CUTOFF, self.\_spot\_cutoff)

glLightf(self.\_source.value, GL\_SPOT\_EXPONENT, self.\_spot\_exponent)

glLightfv(self.\_source.value, GL\_SPOT\_DIRECTION, self.\_transform.get\_vector\_forward())

from .camera import Camera

from .entity import Entity

from .scene\_object import SceneObject

class Scene:

def \_\_init\_\_(self, background\_color: list[float], camera: Camera, \*scene\_objects: SceneObject) -> None:

self.\_camera = camera

self.\_background\_color = background\_color

self.\_scene\_objects = list(scene\_objects)

def get\_camera(self) -> Camera:

return self.\_camera

def get\_scene\_objects(self) -> list:

return self.\_scene\_objects

def get\_background\_color(self) -> list[float]:

return self.\_background\_color

def set\_background\_color(self, value: list[float]) -> None:

self.\_background\_color = value

def destroy(self, scene\_object: SceneObject) -> None:

self.\_scene\_objects.remove(scene\_object)

if (isinstance(scene\_object, Entity)):

scene\_object.get\_mesh().free()

def instantiate(self, scene\_object: SceneObject) -> None:

self.\_scene\_objects.append(scene\_object)

from abc import ABC

from OpenGL.GL import \*

from typing import Callable

from .transform import Transform

class SceneObject(ABC):

def \_\_init\_\_(self, transform: Transform, render\_delegate: Callable[[], None] = None, start\_delegate: Callable[['SceneObject'], None] = None, \*update\_delegates: Callable[['SceneObject', float], None]) -> None:

self.\_is\_active = True

self.\_transform = transform

self.\_start\_delegate = start\_delegate

self.\_render\_delegate = render\_delegate

self.\_update\_delegates = update\_delegates

def get\_transform(self) -> Transform:

return self.\_transform

def get\_is\_active(self, value: bool) -> None:

self.\_is\_active = value

def set\_is\_active(self, value: bool) -> None:

self.\_is\_active = value

def render(self) -> None:

if self.\_is\_active:

self.\_render\_delegate()

glFlush()

def start(self) -> None:

if self.\_is\_active and self.\_start\_delegate is not None:

self.\_start\_delegate(self)

def update(self, delta\_time: float) -> None:

if self.\_is\_active:

for update\_delegate in self.\_update\_delegates:

update\_delegate(self, delta\_time)

import math

from OpenGL.GL import \*

class Transform:

X = 0

Y = 1

Z = 2

AXIS\_X = [1.0, 0.0, 0.0]

AXIS\_Y = [0.0, 1.0, 0.0]

AXIS\_Z = [0.0, 0.0, 1.0]

def \_\_init\_\_(self, position: list[float], rotation: list[float], scale: list[float]) -> None:

self.\_scale = scale

self.\_position = position

self.\_rotation = rotation

self.\_up = [0, 1, 0]

self.\_right = [1, 0, 0]

self.\_forward = [0, 0, -1]

self.\_update\_vectors()

def get\_scale(self) -> list[float]:

return self.\_scale

def set\_scale(self, vector: list[float]) -> None:

self.\_scale = vector

def get\_position(self) -> list[float]:

return self.\_position

def set\_position(self, vector: list[float]) -> None:

self.\_position = vector

def get\_rotation(self) -> list[float]:

return self.\_rotation

def set\_rotation(self, rotation: list[float]) -> None:

self.\_rotation = rotation

self.\_update\_vectors()

def get\_vector\_forward(self) -> list[float]:

return self.\_forward

def get\_vector\_backwards(self) -> list[float]:

return [-x for x in self.\_forward]

def get\_vector\_up(self) -> list[float]:

return self.\_up

def get\_vector\_down(self) -> list[float]:

return [-x for x in self.\_up]

def get\_vector\_right(self) -> list[float]:

return self.\_right

def get\_vector\_left(self) -> list[float]:

return [-x for x in self.\_right]

def apply\_transformations(self) -> None:

glTranslatef(\*self.\_position)

glRotatef(self.\_rotation[Transform.X], \*self.AXIS\_X)

glRotatef(self.\_rotation[Transform.Y], \*self.AXIS\_Y)

glRotatef(self.\_rotation[Transform.Z], \*self.AXIS\_Z)

glScalef(\*self.\_scale)

def scale(self, factors: list[float]) -> None:

self.\_scale = [s \* f for s, f in zip(self.\_scale, factors)]

def translate(self, direction: list[float]) -> None:

self.\_position = [sum(x) for x in zip(self.\_position, direction)]

def rotate(self, angle: float, axis: list[float]) -> None:

self.\_rotation = [sum(x) for x in zip(self.\_rotation, [angle \* a for a in axis])]

self.\_update\_vectors()

def \_update\_vectors(self) -> None:

cosY = math.cos(math.radians(self.\_rotation[Transform.Y]))

sinY = math.sin(math.radians(self.\_rotation[Transform.Y]))

cosP = math.cos(math.radians(self.\_rotation[Transform.X]))

sinP = math.sin(math.radians(self.\_rotation[Transform.X]))

cosR = math.cos(math.radians(self.\_rotation[Transform.Z]))

sinR = math.sin(math.radians(self.\_rotation[Transform.Z]))

self.\_right[Transform.X] = cosR \* cosY - sinR \* sinP \* sinY

self.\_right[Transform.Y] = cosP \* sinR

self.\_right[Transform.Z] = -cosR \* sinY - sinR \* sinP \* cosY

self.\_up[Transform.X] = sinR \* cosY + cosR \* sinP \* sinY

self.\_up[Transform.Y] = cosP \* cosR

self.\_up[Transform.Z] = -sinR \* sinY + cosR \* sinP \* cosY

self.\_forward[Transform.X] = -cosP \* sinY

self.\_forward[Transform.Y] = sinP

self.\_forward[Transform.Z] = cosP \* cosY

from .scene import Scene

from .entity import Entity

from .transform import Transform

from .scene\_object import SceneObject

from .camera import CameraPerspective, CameraOrthographic

from .light import LightSource, LightDirectional, LightPoint, LightSpot

import glfw

from OpenGL.GL import \*

from ..api.scene import Scene

from ..api.entity import Entity

from .preferences import Preferences

class Application:

\_instance = None

\_VIEWPORT\_OFFSET\_X = 0

\_VIEWPORT\_OFFSET\_Y = 0

\_MIN\_DELTA\_TIME = 0.005

\_delta\_time = \_MIN\_DELTA\_TIME

def \_\_new\_\_(cls):

if not cls.\_instance:

cls.\_instance = super(Application, cls).\_\_new\_\_(cls)

return cls.\_instance

def \_\_init\_\_(self) -> None:

if not glfw.init():

return

glfw.window\_hint(glfw.DOUBLEBUFFER, glfw.TRUE)

glfw.window\_hint(glfw.SAMPLES, Preferences.get\_anti\_aliasing\_samples())

self.\_window = glfw.create\_window(Preferences.get\_window\_width(), Preferences.get\_window\_height(), Preferences.get\_window\_title(), None, None)

if not self.\_window:

self.\_destroy()

return

glfw.make\_context\_current(self.\_window)

glfw.swap\_interval(0)

glEnable(GL\_LIGHTING)

glEnable(GL\_DEPTH\_TEST)

glEnable(GL\_MULTISAMPLE)

glEnable(GL\_COLOR\_MATERIAL)

def get\_fps(self) -> int:

return int(1 / self.\_delta\_time)

def get\_delta\_time(self) -> float:

return self.\_delta\_time

def load\_scene(self, scene: Scene) -> None:

self.\_active\_scene = scene

for scene\_object in self.\_active\_scene.get\_scene\_objects():

scene\_object.start()

while not glfw.window\_should\_close(self.\_window):

self.\_update()

glfw.poll\_events()

def \_update(self) -> None:

width, height = glfw.get\_framebuffer\_size(self.\_window)

glViewport(self.\_VIEWPORT\_OFFSET\_X, self.\_VIEWPORT\_OFFSET\_Y, width, height)

Preferences.set\_window\_width(width)

Preferences.set\_window\_height(height)

glfw.set\_window\_title(self.\_window, f"{Preferences.get\_window\_title()} | FPS: {self.get\_fps()}")

start\_time = glfw.get\_time()

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT)

self.\_active\_scene.get\_camera().update(self.\_delta\_time)

for scene\_object in self.\_active\_scene.get\_scene\_objects():

scene\_object.update(self.\_delta\_time)

glClearColor(\*self.\_active\_scene.get\_background\_color())

self.\_active\_scene.get\_camera().render()

for scene\_object in self.\_active\_scene.get\_scene\_objects():

scene\_object.render()

finish\_time = glfw.get\_time()

self.\_delta\_time = (finish\_time - start\_time)

if self.\_delta\_time < Application.\_MIN\_DELTA\_TIME:

self.\_delta\_time = Application.\_MIN\_DELTA\_TIME

glfw.swap\_buffers(self.\_window)

def \_destroy(self) -> None:

for scene\_object in self.\_active\_scene.get\_scene\_objects():

if isinstance(scene\_object, Entity):

scene\_object.get\_mesh().free()

glfw.terminate()

from PIL import Image

from ..graphics.mesh import Mesh

from ..graphics.texture import Texture

class Loader:

@staticmethod

def load\_mesh(relative\_path: str) -> Mesh:

CHAR\_FACE = 'f'

CHAR\_VERTEX = 'v'

CHAR\_NORMAL = 'vn'

CHAR\_COMMENT = '#'

CHAR\_SEPARATOR = '/'

CHAR\_TEXCOORDS = 'vt'

INDEX\_VERTEX = 0

INDEX\_TEXCOORD = 1

INDEX\_NORMAL = 2

faces = []

normals = []

vertices = []

texcoords = []

with open(relative\_path, 'r') as file\_stream:

for line in file\_stream:

if not line.startswith(CHAR\_COMMENT):

values = line.split()

if values:

if values[0] == CHAR\_VERTEX:

vertices.append(list(map(float, values[1:4])))

elif values[0] == CHAR\_NORMAL:

normals.append(list(map(float, values[1:4])))

elif values[0] == CHAR\_TEXCOORDS:

texcoords.append(list(map(float, values[1:3])))

elif values[0] == CHAR\_FACE:

face = []

norms = []

texcoords\_face = []

for value in values[1:]:

face\_components = value.split(CHAR\_SEPARATOR)

face.append(int(face\_components[INDEX\_VERTEX]))

norms.append((int(face\_components[INDEX\_NORMAL]) if len(face\_components) >= 3 and face\_components[INDEX\_NORMAL] else 0))

texcoords\_face.append((int(face\_components[INDEX\_TEXCOORD]) if len(face\_components) >= 2 and face\_components[INDEX\_TEXCOORD] else 0))

faces.append((face, norms, texcoords\_face))

return Mesh(faces, normals, vertices, texcoords)

@staticmethod

def load\_texture(relative\_path: str) -> Texture:

return Texture(Image.open(relative\_path))

class Preferences:

\_window\_title = ""

\_window\_width = 1280

\_window\_height = 720

\_anti\_aliasing\_samples = 1

@classmethod

def get\_window\_title(cls) -> str:

return cls.\_window\_title

@classmethod

def set\_window\_title(cls, value: str) -> None:

cls.\_window\_title = value

@classmethod

def get\_window\_width(cls) -> int:

return cls.\_window\_width

@classmethod

def set\_window\_width(cls, value: int) -> None:

cls.\_window\_width = value

@classmethod

def get\_window\_height(cls) -> int:

return cls.\_window\_height

@classmethod

def set\_window\_height(cls, value: int) -> None:

cls.\_window\_height = value

@classmethod

def get\_aspect\_ratio(cls) -> float:

return cls.\_window\_width / cls.\_window\_height

@classmethod

def get\_anti\_aliasing\_samples(self) -> int:

return self.\_anti\_aliasing\_samples

def set\_anti\_aliasing\_samples(self, value: int) -> None:

self.\_anti\_aliasing\_samples = value

from .loader import Loader

from .preferences import Preferences

from .application import Application

class Material:

pass

from OpenGL.GL import \*

class Mesh:

def \_\_init\_\_(self, faces: list[tuple], normals: list[float], vertices: list[float], texcoords: list[float]) -> None:

self.\_faces = faces

self.\_normals = normals

self.\_vertices = vertices

self.\_texcoords = texcoords

self.\_display\_list = glGenLists(1)

glNewList(self.\_display\_list, GL\_COMPILE)

glFrontFace(GL\_CCW)

glEnable(GL\_TEXTURE\_2D)

for face in self.\_faces:

vertices, normals, texture\_coords = face

glBegin(GL\_POLYGON)

for i in range(len(vertices)):

if normals[i] > 0:

glNormal3fv(self.\_normals[normals[i] - 1])

if texture\_coords[i] > 0:

glTexCoord2fv(self.\_texcoords[texture\_coords[i] - 1])

glVertex3fv(self.\_vertices[vertices[i] - 1])

glEnd()

glDisable(GL\_TEXTURE\_2D)

glEndList()

def build(self) -> None:

glCallList(self.\_display\_list)

def free(self) -> None:

if self.\_display\_list is not None:

glDeleteLists([self.\_display\_list])

from enum import Enum

from OpenGL.GL import \*

from PIL import Image, ImageDraw

class Texture:

def \_\_init\_\_(self, texture: Image) -> None:

self.\_texture = texture

self.\_texture\_id = glGenTextures(1)

glBindTexture(GL\_TEXTURE\_2D, self.\_texture\_id)

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR)

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR)

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA, texture.width, texture.height, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, texture.tobytes("raw", "RGBA", 0, -1))

glGenerateMipmap(GL\_TEXTURE\_2D)

def get\_id(self) -> int:

return self.\_texture\_id

@staticmethod

def generate\_missing\_albedo(size: int = 256, colors: tuple = ("purple", "black")) -> Image:

IMAGE = Image.new("RGBA", (size, size), colors[0])

DRAW = ImageDraw.Draw(IMAGE)

BLOCK\_SIZE = size // 8

for i in range(0, size, BLOCK\_SIZE \* 2):

for j in range(0, size, BLOCK\_SIZE \* 2):

DRAW.rectangle([i, j, i + BLOCK\_SIZE, j + BLOCK\_SIZE], fill = colors[1])

DRAW.rectangle([i + BLOCK\_SIZE, j + BLOCK\_SIZE, i + BLOCK\_SIZE \* 2, j + BLOCK\_SIZE \* 2], fill = colors[1])

return IMAGE

class StandardTextures(Enum):

MISSING\_ALBEDO = Texture.generate\_missing\_albedo()

from .mesh import Mesh

from .texture import Texture

from .material import Material

from ngen.engine import Loader, Application, Preferences

from ngen.api import CameraPerspective, LightDirectional, LightSource, Scene, Entity, Transform

INITIAL\_POSITION\_OFFSET = 20.0

car1\_velocity = 1.5

car2\_velocity = 2.0

car3\_velocity = 2.5

car4\_velocity = 2.75

car5\_velocity = 2.25

car6\_velocity = 1.75

def callback\_car1\_update\_movement(entity: Entity, delta\_time: float) -> None:

global car1\_velocity

entity.get\_transform().translate([car1\_velocity \* delta\_time, 0, 0])

if abs(entity.get\_transform().get\_position()[Transform.X]) > INITIAL\_POSITION\_OFFSET:

car1\_velocity \*= -1

entity.get\_transform().set\_rotation([0.0, -(entity.get\_transform().get\_rotation()[Transform.Y]), 0.0])

if entity.get\_transform().get\_position()[Transform.X] > 0:

entity.get\_transform().set\_position([INITIAL\_POSITION\_OFFSET, entity.get\_transform().get\_position()[Transform.Y], 0.0])

else:

entity.get\_transform().set\_position([-INITIAL\_POSITION\_OFFSET, entity.get\_transform().get\_position()[Transform.Y], 0.0])

def callback\_car2\_update\_movement(entity: Entity, delta\_time: float) -> None:

global car2\_velocity

entity.get\_transform().translate([car2\_velocity \* delta\_time, 0, 0])

if abs(entity.get\_transform().get\_position()[Transform.X]) > INITIAL\_POSITION\_OFFSET:

car2\_velocity \*= -1

entity.get\_transform().set\_rotation([0.0, -(entity.get\_transform().get\_rotation()[Transform.Y]), 0.0])

if entity.get\_transform().get\_position()[Transform.X] > 0:

entity.get\_transform().set\_position([INITIAL\_POSITION\_OFFSET, entity.get\_transform().get\_position()[Transform.Y], 0.0])

else:

entity.get\_transform().set\_position([-INITIAL\_POSITION\_OFFSET, entity.get\_transform().get\_position()[Transform.Y], 0.0])

def callback\_car3\_update\_movement(entity: Entity, delta\_time: float) -> None:

global car3\_velocity

entity.get\_transform().translate([car3\_velocity \* delta\_time, 0, 0])

if abs(entity.get\_transform().get\_position()[Transform.X]) > INITIAL\_POSITION\_OFFSET:

car3\_velocity \*= -1

entity.get\_transform().set\_rotation([0.0, -(entity.get\_transform().get\_rotation()[Transform.Y]), 0.0])

if entity.get\_transform().get\_position()[Transform.X] > 0:

entity.get\_transform().set\_position([INITIAL\_POSITION\_OFFSET, entity.get\_transform().get\_position()[Transform.Y], 0.0])

else:

entity.get\_transform().set\_position([-INITIAL\_POSITION\_OFFSET, entity.get\_transform().get\_position()[Transform.Y], 0.0])

def callback\_car4\_update\_movement(entity: Entity, delta\_time: float) -> None:

global car4\_velocity

entity.get\_transform().translate([car4\_velocity \* delta\_time, 0, 0])

if abs(entity.get\_transform().get\_position()[Transform.X]) > INITIAL\_POSITION\_OFFSET:

car4\_velocity \*= -1

entity.get\_transform().set\_rotation([0.0, -(entity.get\_transform().get\_rotation()[Transform.Y]), 0.0])

if entity.get\_transform().get\_position()[Transform.X] > 0:

entity.get\_transform().set\_position([INITIAL\_POSITION\_OFFSET, entity.get\_transform().get\_position()[Transform.Y], 0.0])

else:

entity.get\_transform().set\_position([-INITIAL\_POSITION\_OFFSET, entity.get\_transform().get\_position()[Transform.Y], 0.0])

def callback\_car5\_update\_movement(entity: Entity, delta\_time: float) -> None:

global car5\_velocity

entity.get\_transform().translate([car5\_velocity \* delta\_time, 0, 0])

if abs(entity.get\_transform().get\_position()[Transform.X]) > INITIAL\_POSITION\_OFFSET:

car5\_velocity \*= -1

entity.get\_transform().set\_rotation([0.0, -(entity.get\_transform().get\_rotation()[Transform.Y]), 0.0])

if entity.get\_transform().get\_position()[Transform.X] > 0:

entity.get\_transform().set\_position([INITIAL\_POSITION\_OFFSET, entity.get\_transform().get\_position()[Transform.Y], 0.0])

else:

entity.get\_transform().set\_position([-INITIAL\_POSITION\_OFFSET, entity.get\_transform().get\_position()[Transform.Y], 0.0])

def callback\_car6\_update\_movement(entity: Entity, delta\_time: float) -> None:

global car6\_velocity

entity.get\_transform().translate([car6\_velocity \* delta\_time, 0, 0])

if abs(entity.get\_transform().get\_position()[Transform.X]) > INITIAL\_POSITION\_OFFSET:

car6\_velocity \*= -1

entity.get\_transform().set\_rotation([0.0, -(entity.get\_transform().get\_rotation()[Transform.Y]), 0.0])

if entity.get\_transform().get\_position()[Transform.X] > 0:

entity.get\_transform().set\_position([INITIAL\_POSITION\_OFFSET, entity.get\_transform().get\_position()[Transform.Y], 0.0])

else:

entity.get\_transform().set\_position([-INITIAL\_POSITION\_OFFSET, entity.get\_transform().get\_position()[Transform.Y], 0.0])

def main() -> None:

application = Application()

Preferences.set\_window\_title("ІП-14 Бабіч Денис. Лабораторна робота № 9")

directional\_light = LightDirectional(Transform([0, 0, 0], [0, 0, 0], [1, 1, 1]), LightSource.LIGHT0, [1.0, 1.0, 1.0])

camera = CameraPerspective(Transform([0.0, 0.0, -10.0], [0, 0, 0], [1, 1, 1]), 100.0, 0.01, 100.0, None)

car1 = Entity(Transform([INITIAL\_POSITION\_OFFSET, 8.0, 0], [0, 90, 0], [1.0, 1.0, 1.0]), Loader.load\_mesh("9/assets/car1/car1.obj"), Loader.load\_texture("9/assets/car1/car1.png"), None, callback\_car1\_update\_movement)

car2 = Entity(Transform([-INITIAL\_POSITION\_OFFSET, 4, 0], [0, 90, 0], [1.0, 1.0, 1.0]), Loader.load\_mesh("9/assets/car2/car2.obj"), Loader.load\_texture("9/assets/car2/car2.png"), None, callback\_car2\_update\_movement)

car3 = Entity(Transform([INITIAL\_POSITION\_OFFSET, 0.5, 0], [0, 90, 0], [1.0, 1.0, 1.0]), Loader.load\_mesh("9/assets/car3/car3.obj"), Loader.load\_texture("9/assets/car3/car3.png"), None, callback\_car3\_update\_movement)

car4 = Entity(Transform([-INITIAL\_POSITION\_OFFSET, -2.5, 0], [0, 90, 0], [1.0, 1.0, 1.0]), Loader.load\_mesh("9/assets/car4/car4.obj"), Loader.load\_texture("9/assets/car4/car4.png"), None, callback\_car4\_update\_movement)

car5 = Entity(Transform([INITIAL\_POSITION\_OFFSET, -5.5, 0], [0, 90, 0], [1.0, 1.0, 1.0]), Loader.load\_mesh("9/assets/car4/car4\_taxi.obj"), Loader.load\_texture("9/assets/car4/car4\_taxi.png"), None, callback\_car5\_update\_movement)

car6 = Entity(Transform([-INITIAL\_POSITION\_OFFSET, -9.5, 0], [0, 90, 0], [1.0, 1.0, 1.0]), Loader.load\_mesh("9/assets/car4/car4\_police.obj"), Loader.load\_texture("9/assets/car4/car4\_police.png"), None, callback\_car6\_update\_movement)

scene = Scene([0.0, 0.694, 0.251, 1.0], camera, directional\_light, car1, car2, car3, car4, car5, car6)

application.load\_scene(scene)

if \_\_name\_\_ == "\_\_main\_\_":

main()

1. **Висновки.**

У даній лабораторній роботі було створено простий рушій для роботи з 3Д графікою, який має базовий функціонал для скриптингу, завантаження моделей, роботи з освітленням та віртуальними камерами. Для демонстрації роботи було створено сцену з машинами, яка може використовуватися для створення навчального датасету. Варто зазначити, що дана реалізація рушія є базовою, де використовуються застарілі підходи для роботи зі світлом і відсутня реалізації таких функціональних особливостей, як підтримка шейдерів та матеріалів, проте надає розуміння влаштування подібних розробок.

Виконав: ІП-14 Бабіч Д. В.