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

I. Мета:

Дослідити методологію і технології створення доповненої реальності.

II. Завдання:

Завдання II

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.

III. Результати виконання лабораторної роботи.

3.1. 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.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

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

$$\begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

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

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

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.

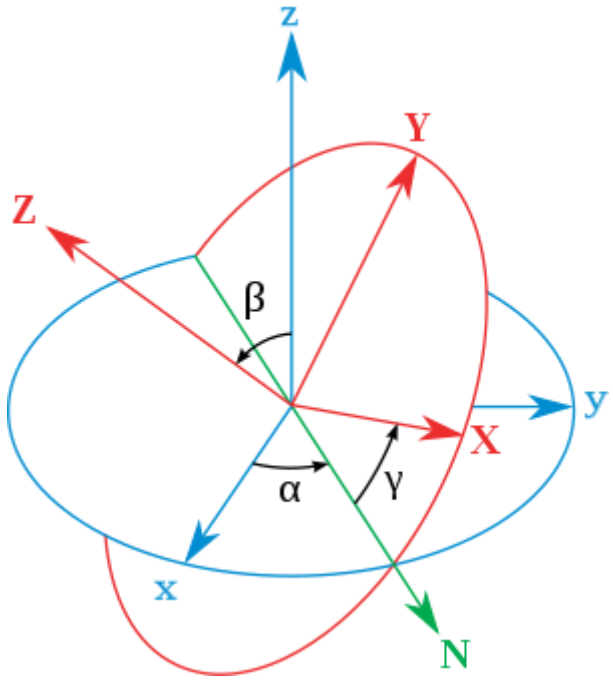

Рисунок 1.1 – Представлення кутів Ейлера

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.

3.2. Результати архітектурного проектування та їх опис.

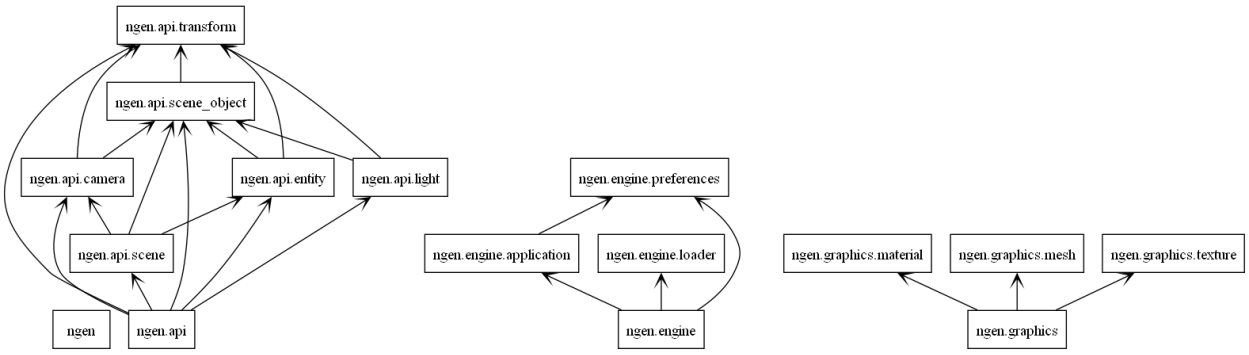

Рисунок 1.2 – Архітектура рушія

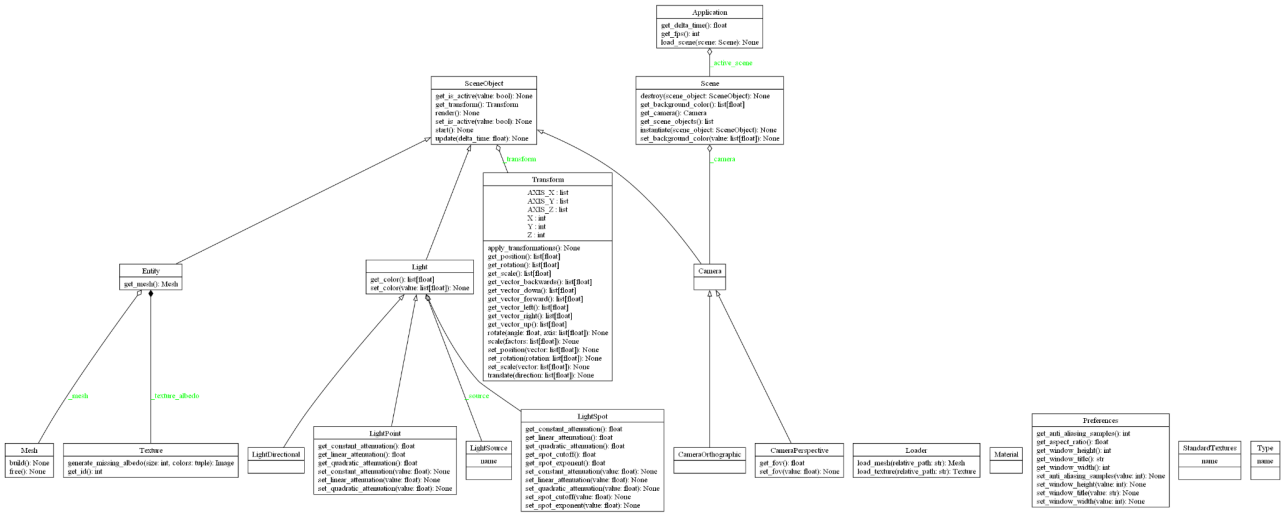

Рисунок 1.3 – UML-діаграма класів

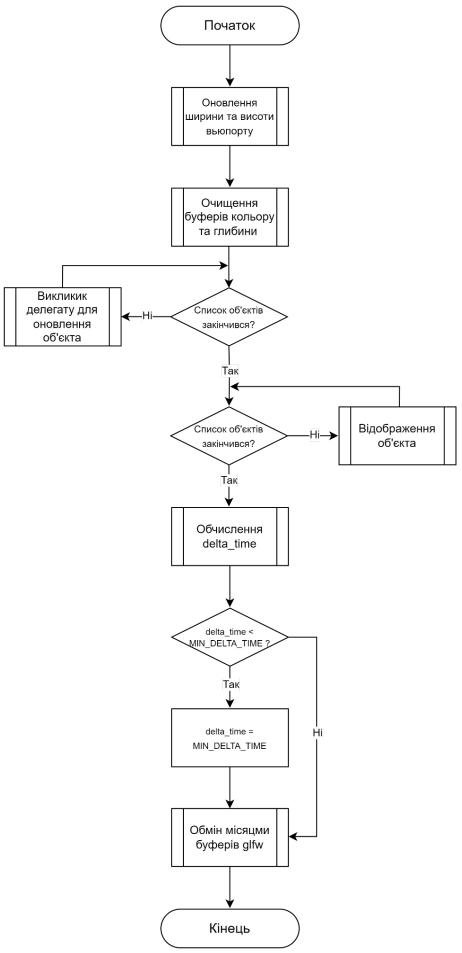

Рисунок 1.4 – Діаграма головного рантайм-циклу застосунку

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.

3.3. Опис структури проекту програми.

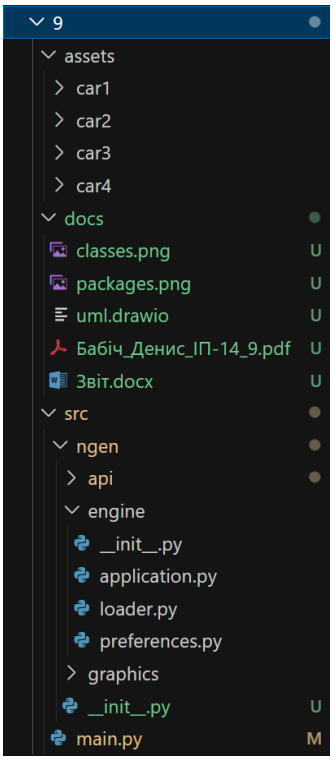

Рисунок 1.5 – Структура проєкту

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.

3.4. Результати роботи програми відповідно до завдання.

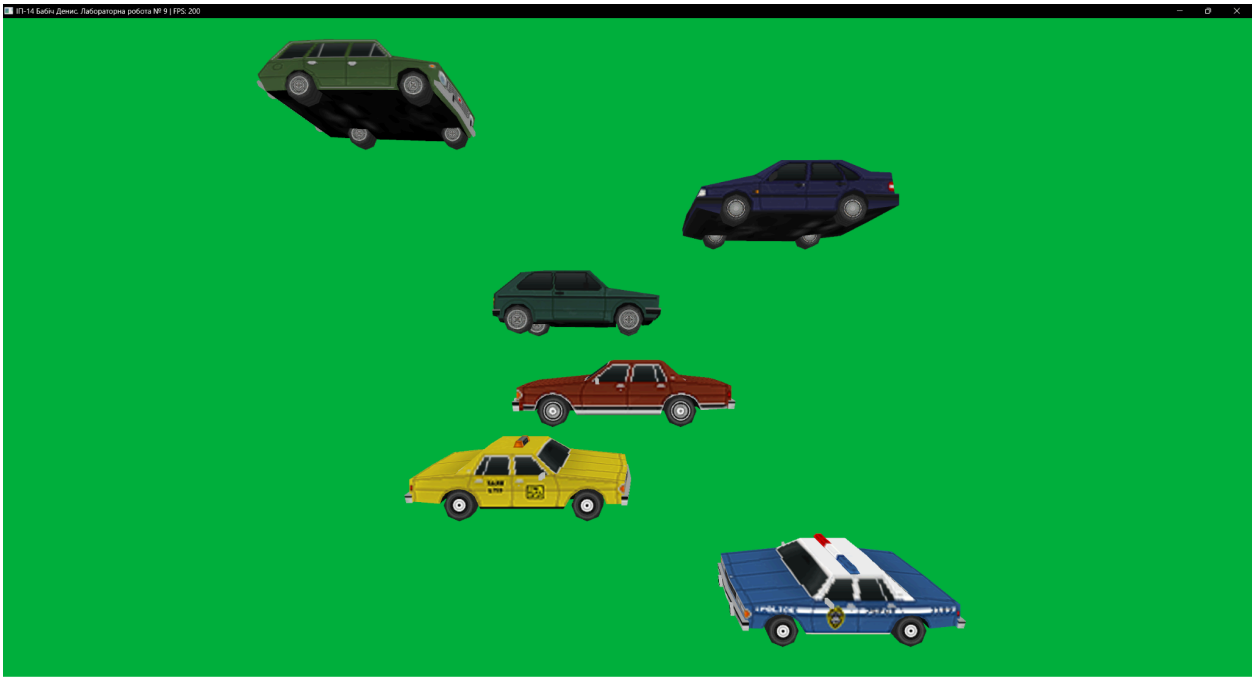

Рисунок 1.6 – Робота створеного застосунку

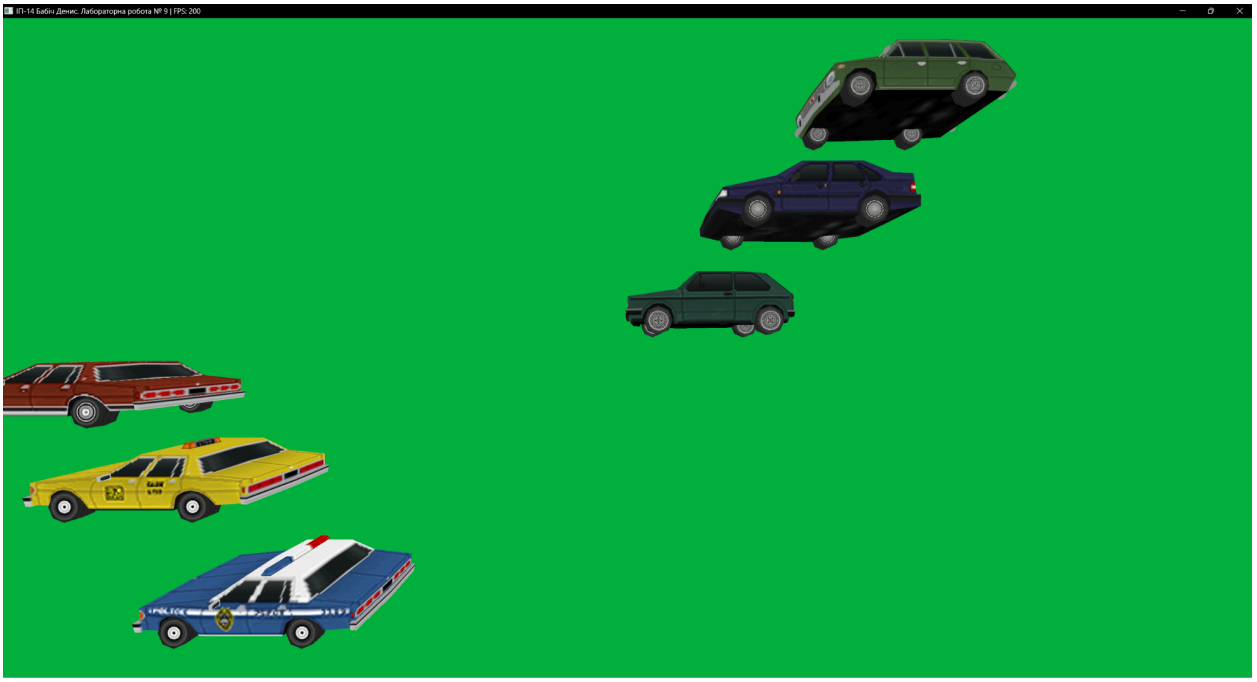

Рисунок 1.7 – Демонстрація руху машин

3.5. 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()
```

IV. 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.