**Міністерство освіти і науки України  
Дніпровський національний університет   
імені Олеся Гончара**

з дисципліни «Інформаційні та комунікаційні технології»

Методичні вказівки до виконання

лабораторних робіт з дисципліни

«Обчислювальна геометрія та комп’ютерна графіка»

**2023Міністерство освіти і науки України  
Дніпровський національний університет   
імені Олеся Гончара**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Кафедра компʼютерних технологій**

Методичні вказівки до виконання

лабораторних робіт з дисципліни

«Обчислювальна геометрія та комп’ютерна графіка»

*Ухвалено на вченій раді*

*протокол №*

*від 2023 р.*

**Дніпро**

**РВВ ДНУ**

**2023**

Наведено стислі теоретичні та практичні відомості з курсу «Обчислювальна геометрія та комп’ютерна графіка». Запропоновано лабораторні роботи та вказівки до їх виконання.

Призначено для здобувачів вищої освіти першого (бакалаврського) рівня спеціальності 113 Прикладна математика.

Темплан 2023, поз.

Навчальне видання

**Методичні вказівки до виконання**

**лабораторних робіт з дисципліни**

**«Обчислювальна геометрія та комп’ютерна графіка»**

Укладачі:

канд. физ.-мат. наук, доц. Н. І. Степанова,

асист. Д.І. Єгошкін

Технічний редактор

Підписано до друку 18.02.2022 Формат 60х84/16. Папір друкарський.

Друк плоский. Ум. друк. арк. 1,4. Ум. Фарбовідб. 1,4. Обл.- вид. арк. 1,5.

Тираж 20 пр. Зам. №

РВВ ДНУ, просп. Гагаріна, 72, м. Дніпро, 49010.

ПП «Ліра ЛТД», вул. Наукова, 5, м. Дніпро, 49107.

Свідоцтво про внесення до Державного реєстру

Серія ДК № 6042 від 26.02.2018

**ЗМІСТ**

[ВСТУП 4](#_Toc132013799)

[ЛАБОРАТОРНА РОБОТА 1 6](#_Toc132013800)

[ЛАБОРАТОРНА РОБОТА 2 11](#_Toc132013801)

[ЛАБОРАТОРНА РОБОТА 3 18](#_Toc132013802)

[ЛАБОРАТОРНА РОБОТА 4 22](#_Toc132013803)

[ДОДАТКИ 30](#_Toc132013804)

[ІНФОРМАЦІЙНІ РЕСУРСИ 33](#_Toc132013805)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 33](#_Toc132013806)

## ВСТУП

З часів появи електронних обчислювальних пристроїв йде постійне зростання можливостей комп’ютерної техніки і удосконалення програмного забезпечення, що суттєво впливає на способи обробки інформації. Комп’ютерні системи все глибше проникають у життя сучасного суспільства і стають незамінним його елементом. З’являються нові методи, алгоритми і технології обробки, в тому числі й такі, без яких вже важко уявити використання обчислювальної техніки. До таких технологій, безумовно, в першу чергу слід віднести методи комп’ютерної графіки: тривимірне моделювання, анімацію, користувацький інтерфейс тощо.

Сьогодні комп’ютерна графіка – складова всіх сучасних комп’ютерних технологій. Графічний інтерфейс став стандартом для програмних систем різних типів і використовується фахівцями різних галузей знань –проектувальниками, інженерами, архітекторами, конструкторами. І безперечно, розуміння базових математичних концепцій, які створюють основу для побудови цифрових зображень, вміння правильно застосувати їх у програмних продуктах, знання сучасних графічних бібліотек – важливі компоненти освіти сучасного ІТ-фахівця.

Комп’ютерна графіка – це галузь знань, в якій, з одного боку, вже накопичено значний багаж теоретичних знань і всіляких практик, з іншого боку, здійснюється постійний розвиток методів, алгоритмів та практичних підходів, це складна і різноманітна дисципліна.

В багатьох випадках потреби в графіці можуть бути забезпечені різними існуючими графічними бібліотеками та системами. Однак, постійно виникає необхідність створювати спеціальні графічні програмні засоби. Зробити це можна, якщо оволодіти практичними навичками розв’язування типових задач КГ та відповідними теоретичними знаннями.

У першу чергу це знання з обчислювальної геометрії, тому що саме ця дисципліна стала математичною основою для побудови того графічного різноманіття, яке ми бачимо на наших моніторах.

Мета даного видання – познайомити здобувачів вищої освіти з різними аспектами і можливостями комп’ютерної графіки, допомогти їм отримати практичні навички використання сучасних графічних бібліотек.

Під час виконання лабораторних робіт необхідно розуміння наступних понять комп’ютерної графіки:

*Цифрове зображення* – сукупність даних, яку отримано за результатами дискретизації (аналого-цифрового перетворення) оригінального зображення.

*Рендеринг (візуалізація)* – процес створення фінального зображення на основі вхідних геометричних даних (двомірних або тривимірних) з використанням обчислювальної системи і комп’ютерних програм, які містять великий обсяг математичних обчислень.

*Графічний процесор* – окремий пристрій комп’ютера, який виконує графічний рендеринг.

*Графічний конвеєр –* концептуальна модель, яка описує послідовність дій графічної системи для відображення тривимірної сцени на двовимірному екрані. Окремі етапи графічного конвеєра реалізовані апаратно, що дозволяє оптимізувати процес візуалізації за часом виконання.

# ЛАБОРАТОРНА РОБОТА 1

**Знайомство з графічною бібліотекою OpenGL та запуск шаблону програми Rainbow**

OpenGL (англ. Open Graphics Library) - це кросплатформове програмне забезпечення для реалізації графічних додатків. Воно використовується для створення комп'ютерних ігор, візуалізації даних, моделювання і анімації 3D об'єктів, а також для створення інтерфейсів користувача і програм для обробки зображень.

OpenGL надає програмістам набір функцій для відображення 2D та 3D графіки на екрані, включаючи роботу з тривимірними об'єктами, текстурами, освітленням, тіньовими ефектами та іншими графічними ефектами.

OpenGL є відкритим стандартом, що означає, що його можуть використовувати різні розробники програмного забезпечення та виробники апаратного забезпечення без необхідності платити за ліцензії. Це дозволяє створювати графічні додатки, які можуть працювати на різних операційних системах та на різних пристроях з різною апаратною конфігурацією.

GLFW (англ. Graphics Library Framework) - це кросплатформова бібліотека для створення віконних інтерфейсів та обробки подій в графічних додатках. GLFW дозволяє створювати вікна, обробляти клавіші, мишу та інші події вводу, створювати OpenGL контексти, обробляти помилки та інші завдання, пов'язані з розробкою графічних додатків.

GLEW та GLAD – ці бібліотеки є взаємозамінними. Тому потрібно використовувати тільки одну з них виходячи з архітектури комп’ютера на якому буде збиратися проект.

GLEW (англ. OpenGL Extension Wrangler Library) - це кросплатформова бібліотека, яка допомагає розробникам програмного забезпечення взаємодіяти з розширеннями OpenGL. Розширення OpenGL - це функції, які не входять до основної специфікації OpenGL, але дозволяють розширювати можливості цієї бібліотеки.

GLEW дозволяє розробникам програмного забезпечення легко завантажувати і використовувати розширення OpenGL на різних платформах. Вона надає API для завантаження функцій розширень OpenGL, перевірки підтримки розширень та інших завдань, пов'язаних з взаємодією з розширеннями OpenGL.

GLEW підтримує різні версії OpenGL, включаючи OpenGL 4.x, а також деякі інші графічні API, такі як WGL, GLX та EGL. GLEW дозволяє розробникам програмного забезпечення максимально використовувати можливості графічних процесорів та інших пристроїв для графіки, що забезпечує покращення продуктивності та якості графічних додатків.

GLAD (англ. OpenGL Loader Generator) - це генератор завантажувачів OpenGL, який допомагає розробникам програмного забезпечення створювати код для завантаження розширень OpenGL. Розширення OpenGL - це функції, які не входять до основної специфікації OpenGL, але дозволяють розширювати можливості цієї бібліотеки.

GLAD генерує код для завантаження розширень OpenGL на основі відомостей про наявність розширень на конкретній системі. Це дозволяє розробникам програмного забезпечення отримувати доступ до всіх можливостей OpenGL на різних платформах, включаючи ті, які не мають підтримки для певних розширень.

GLAD може генерувати код для завантаження розширень OpenGL для різних мов програмування, включаючи C, C++, Python та Java. Вона також підтримує різні версії OpenGL, включаючи OpenGL 4.x. GLAD допомагає розробникам програмного забезпечення ефективно використовувати розширення OpenGL для поліпшення продуктивності та якості графічних додатків.

У додатку 1 можна отримати довідкову інформацію щодо функцій OpenGL, потрібних для виконання лабораторної роботи.

**Мета роботи:** знайомство з кросплатформовим програмним інтерфейсом для реалізації графічних додатків OpenGL

**Завдання до виконання:**

1. Завантажити OpenGL бібліотеки:
   1. GLFW - <https://www.glfw.org/download.html>

GLEW або GLAD:

* 1. GLEW - <http://glew.sourceforge.net/>
  2. GLAD - <https://glad.dav1d.de/>

1. Ознайомитись з можливостями графічної бібліотеки OpenGL. Розібратися з особливостями підключення бібліотеки GLFW.
2. За допомогою шаблону програми (RAINBOW) [<https://github.com/KnightDanila/GraphicProjects_OpenGL_Shaders_GLSL/tree/master/Lesson1/Task01Src>] - розглянути та запустити код.
3. Додати у консоль вивід:

|  |
| --- |
| Hello OpenGL - I am Glad to see you  Author: The student of the faculty of Applied Mathematics (Harry James Potter/Hermione Jean Granger) |

1. Створити вікно glfwCreateWindow(...) розміром 640х480, та назвою "Lesson 01 - RAINBOW - (Harry James Potter/Hermione Jean Granger) ПК-13-03".
2. Оформити звіт, в якому описати хід виконання завдань та результат.

# 

**Доповнення:**

**Unix|Linux** build commands:

|  |
| --- |
| g++ main.cpp -o rainbow -lGLEW -lGL –lglfw |

**Windows** MinGW g++ commands:

|  |
| --- |
| g++ --static-libgcc -static-libstdc++ -lglew32s -lglew32 -l:glfw3.a -lglut32 -lopengl32 -lgdi32 |

Лістинг коду шаблону програми:

|  |
| --- |
| /\*  \* File: main.cpp  \*/  #include <stdio.h>  #include <iostream>  #include <cstdlib>  #include <windows.h>  #include <conio.h>  #include <locale.h>  #include <cmath>  #include <string>  #include <vector>  // OpenGL  #define GLUT\_DISABLE\_ATEXIT\_HACK  #define PI 3.14159265  //#define GLFW\_DLL  //#define GLEW\_STATIC  #pragma comment(lib, "libs\\GL\_AL\\glfw3.lib")  #pragma comment(lib, "libs\\GL\_AL\\glut32.lib")  //#pragma comment(lib, "libs\\GL\_AL\\glut32.dll")  #pragma comment(lib, "libs\\GL\_AL\\alut.lib")  #pragma comment(lib, "libs\\GL\_AL\\glew32.lib")  #pragma comment(lib, "libs\\GL\_AL\\glew32s.lib")  // VS2013+ - bugfix - Thanks to Bolsunov Dmitry  #pragma comment(lib, "msvcrt.lib")  #pragma comment(lib, "msvcmrt.lib")  #pragma comment(lib, "legacy\_stdio\_definitions.lib")  #include "libs\GL\_AL\glew.h"  //#include <glad/glad.h> // I will add it soon  #include "libs\GL\_AL\glfw3.h"  #include "libs\GL\_AL\glut.h"  using namespace std;  namespace print\_lib  {  template <typename T>  void print(T i, string endline = "\n") {  cout << i << endline;  }  void argsEcho(int argc, char\*\* argv) {  print("\_\_\_\_ARGS\_\_\_\_");  if (argc == 0) {  print("NO ARGS");  }  else {  for (int i = 0; i < argc; i++) {  if (i != argc - 1) {  cout << i + 1 << ") ";  print(argv[i], ";\n");  }  else {  cout << i + 1 << ") ";  print(argv[i], ".\n");  }  }  }  print("\_\_\_\_\_\_\_\_\_\_\_\_\n");  }  }  using namespace print\_lib;  /\*  \* The main part of the code  \*/  int main(int argc, char\*\* argv) {  argsEcho(argc, argv);  print("Hello OpenGL");  if (!glfwInit())  {  // Ошибка при инициализации GLFW  fprintf(stderr, "GLFW - Initialization error\n");  return -1;  }  glfwWindowHint(GLFW\_SAMPLES, 4); // 4x Anti-Aliasing - Згладжування 4x  glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3); // Ми хочемо використати OpenGL 3.3  glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);  glfwWindowHint(GLFW\_OPENGL\_FORWARD\_COMPAT, GL\_TRUE); // To make MacOS happy; should not be necessary  //glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);  /\*  GLFWmonitor\* monitor = glfwGetPrimaryMonitor();  const GLFWvidmode\* mode = glfwGetVideoMode(monitor);  glfwWindowHint(GLFW\_RED\_BITS, mode->redBits);  glfwWindowHint(GLFW\_GREEN\_BITS, mode->greenBits);  glfwWindowHint(GLFW\_BLUE\_BITS, mode->blueBits);  glfwWindowHint(GLFW\_REFRESH\_RATE, mode->refreshRate);  GLFWwindow\* window;  window = glfwCreateWindow(mode->width, mode->height, "Lesson 01", monitor, NULL);  \*/  // Відкрити вікно та створити в ньому контекст OpenGL  GLFWwindow\* window;  window = glfwCreateWindow(640, 480, "Lesson 01", NULL, NULL);  if (window == NULL) {  fprintf(stderr, "It is not possible to open the window GLFW. If you have Intel GPU, then it does not support the OpenGL 3.3. Try OpenGL 2.1.\n");  glfwTerminate();  return -1;  }  glfwMakeContextCurrent(window);  // Ініціалізуємо GLEW  glewExperimental = true; // Прапор необхідний у Core-режимі OpenGL  if (glewInit() != GLEW\_OK) {  fprintf(stderr, "Unable to initialize GLEW\n");  return -1;  }  // Увімкнемо режим відстеження натискання клавіш для перевірки нижче  glfwSetInputMode(window, GLFW\_STICKY\_KEYS, GL\_TRUE);  float colorRGB = 0.0;  // Перевіряємо натискання клавіші Escape або закриття вікна  while (glfwGetKey(window, GLFW\_KEY\_ESCAPE) != GLFW\_PRESS &&  glfwWindowShouldClose(window) == 0)  {  glClearColor(sin(colorRGB \* PI / 180), abs(cos(colorRGB \* PI / 180)), abs(sin(colorRGB \* PI / 180) + cos(colorRGB \* PI / 180)), 1.0f);  glClear(GL\_COLOR\_BUFFER\_BIT);  // Скидаємо буфери  glfwSwapBuffers(window);  glfwPollEvents();  {  colorRGB <= 180 ? colorRGB += 0.1 : colorRGB = 0;  }  }  glfwTerminate();  system("pause");  return 0;  } |

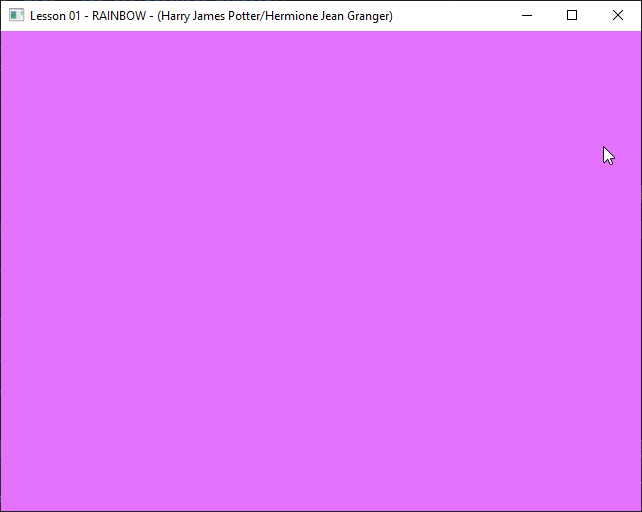


Рис. 1 – Результат виконання лабораторної роботи №1: Вікно OpenGL - Rainbow

# ЛАБОРАТОРНА РОБОТА 2

**Робота з Context, Renderer, VBO, VAO, EBO та алгоритмами малювання**

Контекст (Context) - це об'єкт, який визначає параметри та налаштування графічного контексту для програми. У більшості випадків контекст пов'язаний з вікном програми, яке використовується для відображення графіки. Контекст містить в собі інформацію про обладнання та драйвери, що потрібні для обробки графіки, та дозволяє програмі взаємодіяти з цим обладнанням.

Рендерер (Renderer) - це компонент, який відповідає за відображення графіки на екрані. Він містить в собі реалізацію алгоритмів малювання, які дозволяють створювати зображення з графічних об'єктів, які можуть бути відображені на екрані. Рендерер використовує контекст для взаємодії з обладнанням та драйверами, що потрібні для обробки графіки.

У більшості випадків контекст та рендерер пов'язані між собою та взаємодіють, щоб забезпечити відображення графіки на екрані. Контекст встановлюється перед початком роботи з рендерером, та може бути налаштований залежно від потреб програми. Рендерер же використовує контекст для взаємодії з обладнанням та відображення графіки на екрані відповідно до параметрів, які були встановлені в контексті.

**Мета роботи:** знайомство з об’єктами та компонентами графічної бібліотеки OpenGL

**Завдання до виконання:**

1. Створіть своє унікальне ім'я графічного рушія OpenGL - як абревіатуру вашого імені. Наприклад GSC - Григорович Сергій Константинович (GSC Game World)

|  |
| --- |
| namespace GSC{  } |

1. Використовуючи шаблон програми Task02Src [<https://github.com/KnightDanila/GraphicProjects_OpenGL_Shaders_GLSL/tree/master/Lesson2/Task02Src>] - запишіть та запустіть код з вашим простором імен.
2. Використовуйте діаграму UML для довідки:

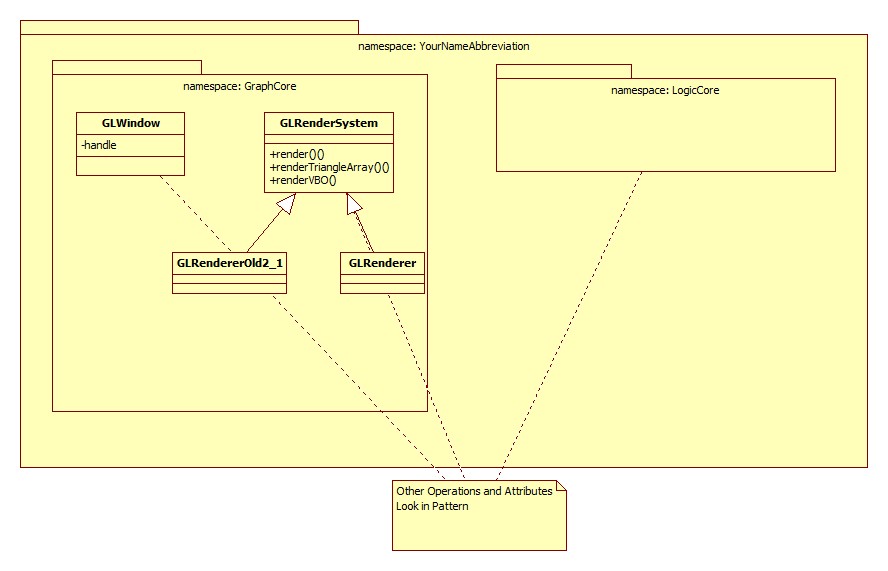


Рис. 2 – UML діаграма архітектури основних класів проекту

1. Додати у консоль вивід:

|  |
| --- |
| Task 2  Author: The student of the faculty of Applied Mathematics (Harry James Potter/Hermione Jean Granger) |

1. Створіть 3 вікна за допомогою класу GLWindow
2. Намалюйте об'єкт VBO (без текстури)
3. Намалюйте простий масив точок glDrawArrays()
4. Покажіть, що ви можете використовувати дві рендерні машини GLRender() та GLRendererOld2\_1().
5. Оформити звіт, в якому описати хід виконання завдань та результат.

**Лістинг коду шаблону програми:**

**main.cpp**

|  |
| --- |
| #include "GLWindow.h"  #include "GLRenderSystem.h"  namespace Cube {  float vertices[] = {  -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,  0.5f, -0.5f, -0.5f, 1.0f, 0.0f,  0.5f, 0.5f, -0.5f, 1.0f, 1.0f,  0.5f, 0.5f, -0.5f, 1.0f, 1.0f,  -0.5f, 0.5f, -0.5f, 0.0f, 1.0f,  -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,  -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,  0.5f, -0.5f, 0.5f, 1.0f, 0.0f,  0.5f, 0.5f, 0.5f, 1.0f, 1.0f,  0.5f, 0.5f, 0.5f, 1.0f, 1.0f,  -0.5f, 0.5f, 0.5f, 0.0f, 1.0f,  -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,  -0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  -0.5f, 0.5f, -0.5f, 1.0f, 1.0f,  -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,  -0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  0.5f, 0.5f, -0.5f, 1.0f, 1.0f,  0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  0.5f, -0.5f, 0.5f, 0.0f, 0.0f,  0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  0.5f, -0.5f, -0.5f, 1.0f, 1.0f,  0.5f, -0.5f, 0.5f, 1.0f, 0.0f,  0.5f, -0.5f, 0.5f, 1.0f, 0.0f,  -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,  -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,  -0.5f, 0.5f, -0.5f, 0.0f, 1.0f,  0.5f, 0.5f, -0.5f, 1.0f, 1.0f,  0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  0.5f, 0.5f, 0.5f, 1.0f, 0.0f,  -0.5f, 0.5f, 0.5f, 0.0f, 0.0f,  -0.5f, 0.5f, -0.5f, 0.0f, 1.0f  };  unsigned int VBO;  void init() {  {  }  }  };  namespace Squares {  GLfloat vertices[] = {  0, 0, 0,  1, 0, 0,  1, 1, 0,  0, 1, 0,  0, 0, 0,  0, 1, 0,  -1, 1, 0,  -1, 0, 0  };  GLfloat colors[] = {  255, 0, 0,  255, 0, 0,  255, 0, 0,  255, 0, 0,  0, 0, 255,  0, 0, 255,  0, 0, 255,  0, 0, 255  };  }  using namespace std;  void resize(GLFWwindow \* window, int width, int height) {  println((string) "Width:" + width + "-Height:" + height);  float ratio = width / (float) height;  glViewport(0, 0, width, height);  glClear(GL\_COLOR\_BUFFER\_BIT);  glMatrixMode(GL\_PROJECTION);  glLoadIdentity();  glOrtho(-ratio, ratio, -1.f, 1.f, 1.f, -1.f);  glMatrixMode(GL\_MODELVIEW);  glLoadIdentity();  }  int main(int argc, char\*\* argv) {  Knight3D::GraphCore::GLRenderSystem\* renderer;  renderer->init();  Knight3D::GLWindow\* Win1 = new Knight3D::GLWindow("Lesson 021", 320, 240);  Knight3D::GLWindow\* Win2 = new Knight3D::GLWindow("Lesson 022", 640, 480);  Cube::init();  // Проверяем нажатие клавиши Escape или закрытие окна  while (glfwGetKey(window, GLFW\_KEY\_ESCAPE) != GLFW\_PRESS &&  glfwWindowShouldClose(window) == 0) {    glfwMakeContextCurrent(Win1->getGLFWHandle());  renderer->render(Win1->getGLFWHandle());  ...  glfwSwapBuffers(Win1->getGLFWHandle());  glfwMakeContextCurrent(Win2->getGLFWHandle());  renderer->render(Win2->getGLFWHandle());  ...  glfwSwapBuffers(Win2->getGLFWHandle());  glfwPollEvents();  glfwMakeContextCurrent(window);  }  glDeleteBuffers(1, &Cube::VBO);  glfwDestroyWindow(...);  glfwTerminate();  return 0;  } |

**GLWindow.h**

|  |
| --- |
| #ifndef GLWINDOW\_H  #define GLWINDOW\_H  namespace Knight3D {  class GLWindow {  public:  GLWindow(const std::string& title, uint32\_t width, uint32\_t height);  GLWindow(const std::string& title, uint32\_t width, uint32\_t height, GLFWwindow\* share);  ~GLWindow() {  // glfwDestroyWindow(handle); // only for one-single window app  glfwSetWindowShouldClose(\_handle, true); // for multiple windows app  };  uint32\_t getWidth() const;  uint32\_t getHeight() const;  // ...  GLFWwindow\* getGLFWHandle() const;  void makeCurrent();      private:  // TODO  };  }  #endif /\* GLWINDOW\_H \*/ |

**GLRenderSystem.h**

|  |
| --- |
| #ifndef GLRENDERSYSTEM\_H  #define GLRENDERSYSTEM\_H  namespace Knight3D {  namespace GraphCore {  class GLRenderSystem {  public:  virtual void init() {  }  virtual void render(GLFWwindow \* window) {  }  virtual void renderTriangleArray(GLfloat vertices[], GLfloat colors[]) {  }  virtual void renderVBO() {  }  };  class GLRender : public GLRenderSystem {  void init() {  }  void render(GLFWwindow \* window) {  }  void renderTriangleArray(GLfloat vertices[], GLfloat colors[]) {  }  };  class GLRendererOld2\_1 : public GLRenderSystem {  void init() {  if (!glfwInit()) {  fprintf(stderr, "GLFW - Initialization error \n");  }  glfwWindowHint(GLFW\_SAMPLES, 4);  glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 2);  glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 1);  }  void render(GLFWwindow \* window) {  glClearColor(sin(colorRGB \* PI / 180), abs(cos(colorRGB \* PI / 180)), abs(sin(colorRGB \* PI / 180) + cos(colorRGB \* PI / 180)), 1.0f);  glClear(GL\_COLOR\_BUFFER\_BIT);  {  colorRGB <= 180 ? colorRGB += 0.1 : colorRGB = 0;  }  glLoadIdentity();  glRotatef((float) glfwGetTime() \* 50.f, 0.f, 0.f, 1.f);  glBegin(GL\_TRIANGLES);  glColor3f(1.f, 0.f, 0.f);  glVertex3f(-0.6f, -0.4f, 0.f);  glColor3f(0.f, 1.f, 0.f);  glVertex3f(0.6f, -0.4f, 0.f);  glColor3f(0.f, 0.f, 1.f);  glVertex3f(0.f, 0.6f, 0.f);  glEnd();  }  void renderTriangleArray(GLfloat vertices[], GLfloat colors[]) {  glClear(GL\_COLOR\_BUFFER\_BIT);  glEnableClientState(GL\_VERTEX\_ARRAY);  glEnableClientState(GL\_COLOR\_ARRAY);  glVertexPointer(3, GL\_FLOAT, 0, vertices);  glColorPointer(3, GL\_FLOAT, 0, colors);  glDrawArrays(GL\_QUADS, 0, 8);  glDisableClientState(GL\_COLOR\_ARRAY);  glDisableClientState(GL\_VERTEX\_ARRAY);  }  };  }  }  #endif /\* GLRENDERSYSTEM\_H \*/ |

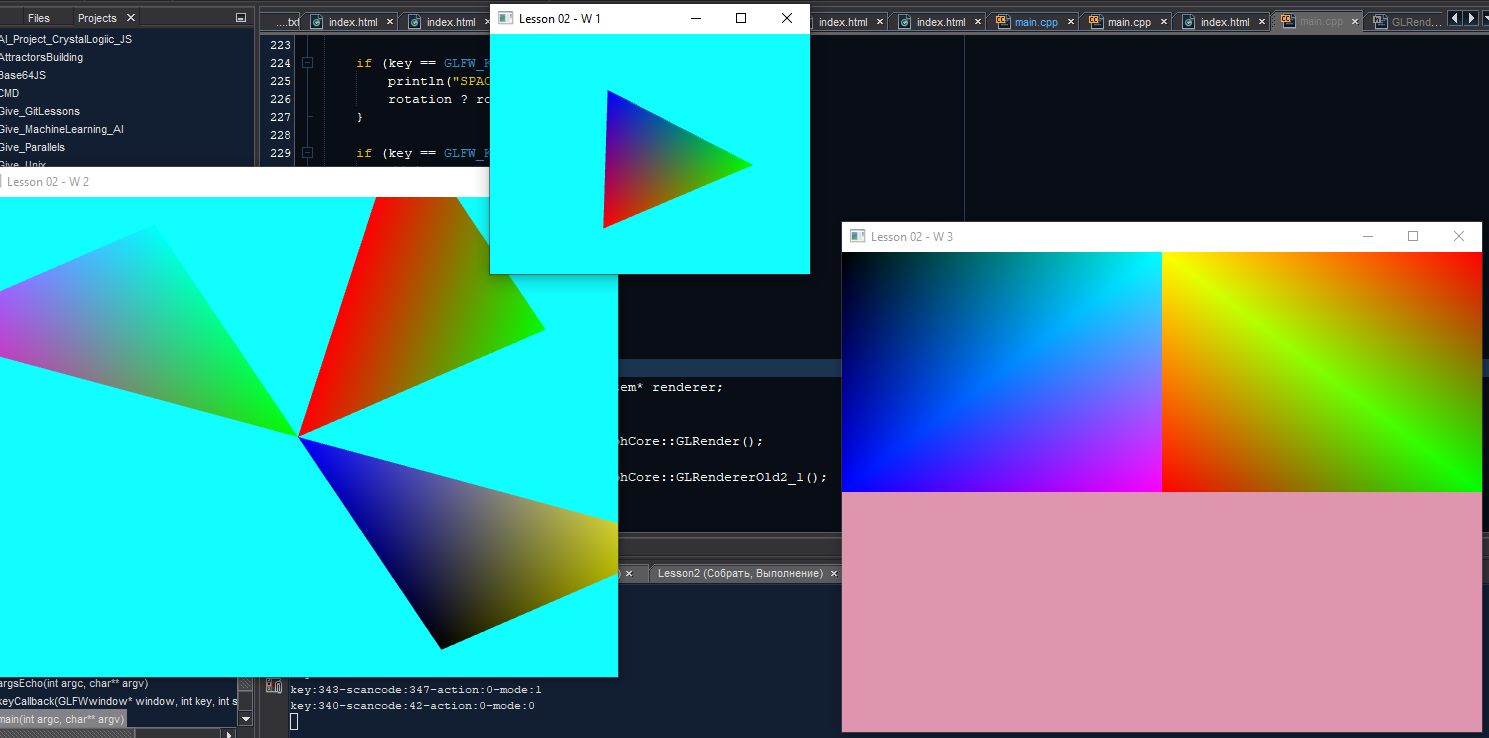


Рис. 3 – Результат виконання лабораторної роботи №2

# ЛАБОРАТОРНА РОБОТА 3

**Робота з камерою в OpenGL та FOV, Aspect**

Камера в OpenGL - це віртуальний об'єкт, який відображає точку спостереження відносно сцени, тобто області, яку потрібно відобразити. Камера дозволяє переміщатись в тривимірному просторі та змінювати кут огляду, щоб змінювати положення та орієнтацію спостерігача відносно сцени.

В OpenGL камера відображається за допомогою матриці перетворень - матриці моделювання та матриці проекції. Матриця моделювання задає положення та орієнтацію камери у світі, тобто координати та напрямки взгляду та вектора "вгору". Матриця проекції відповідає за проекцію тривимірної сцени на двовимірний екран.

Для переміщення камери в OpenGL можна використовувати різні команди трансформації, такі як зсування, поворот та масштабування. Також можна використовувати матриці перетворень для визначення огляду сцени з різних кутів та під різними кутами.

Узагальнюючи, камера в OpenGL - це важливий інструмент для відображення тривимірних сцен, який дозволяє переміщуватись в просторі та змінювати кут огляду, щоб забезпечити користувачеві більш реалістичний та інтерактивний досвід.

**Мета роботи:** отримати практичні навички користування камерою у графічній бібліотеці OpenGL

**Завдання до виконання:**

1. Ознайомитись з можливостями графічної бібліотеки GLM. Розібратися з особливостями підключення бібліотеки GLM.
2. Використовуючи шаблон програми Task03Src [<https://github.com/KnightDanila/GraphicProjects_OpenGL_Shaders_GLSL/tree/master/Lesson3/Task03Src>]  - запишіть та запустіть код з вашим простором імен.
3. Використовуйте діаграму UML для довідки:

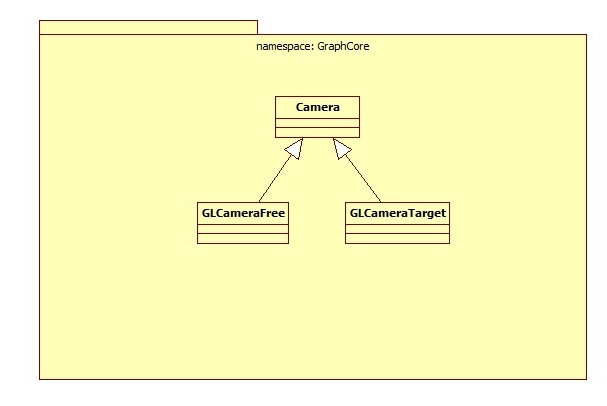


Рис. 4 – UML діаграма архітектури класів камери

1. Додайте вихід до консолі:

|  |
| --- |
| Task 3  Author: The student of the faculty of Applied Mathematics (Harry James Potter/Hermione Jean Granger) |

1. Створіть камеру за допомогою класу ...::GraphCore::Camera\* CamFree = new ...::GraphCore::GLCameraFree();
2. Встановіть перспективу CamFree->setPerspective(...);
3. Намалюйте об'єкт VBO (з лабораторної 2)
4. І перемістіть камеру за допомогою траекторії {x = r \* cos(df); y = r \* sin(df);}
5. Оформити звіт, в якому описати хід виконання завдань та результат.

**Лістинг коду шаблону програми:**

**GLCamera.h**

|  |
| --- |
| #ifndef GLCAMERA\_H  #define GLCAMERA\_H  namespace ... {  namespace GraphCore {  class Camera {  public:  virtual void setPerspective(float fov, float aspect, float near, float far) {  };  virtual void setPos(glm::vec3 pos) {  };  virtual void setTarget(glm::vec3 pos) {  };  virtual glm::vec3 getPos() {  };  private:  /...  };  class GLCameraFree : public Camera {  public:  void setPerspective(float fov, float aspect, float near1, float far1) {  /\*  // Calculate The Aspect Ratio Of The Window  // The parameters are:  // (view angle, aspect ration of the width to the height,  // The closest distance to the camera before it clips,  // FOV, Ratio, The farthest distance before it stops drawing)  \*/  glMatrixMode(...);  glm::mat4 modelproj = glm::perspective(...);  glLoadMatrixf(...); // Підказка - Ми маємо брати покажчик    };  void setPos(glm::vec3 pos) {  glm::vec3 target = ...  glm::vec3 direction = glm::normalize(...);  glm::mat4 modelview = glm::lookAt(...);  glMatrixMode(...);  glLoadMatrixf(...);    }  glm::vec3 getPos() {  ...  };  };  class GLCameraTarget : public Camera {  ...  };  };  }  #endif /\* GLCAMERA\_H \*/ |

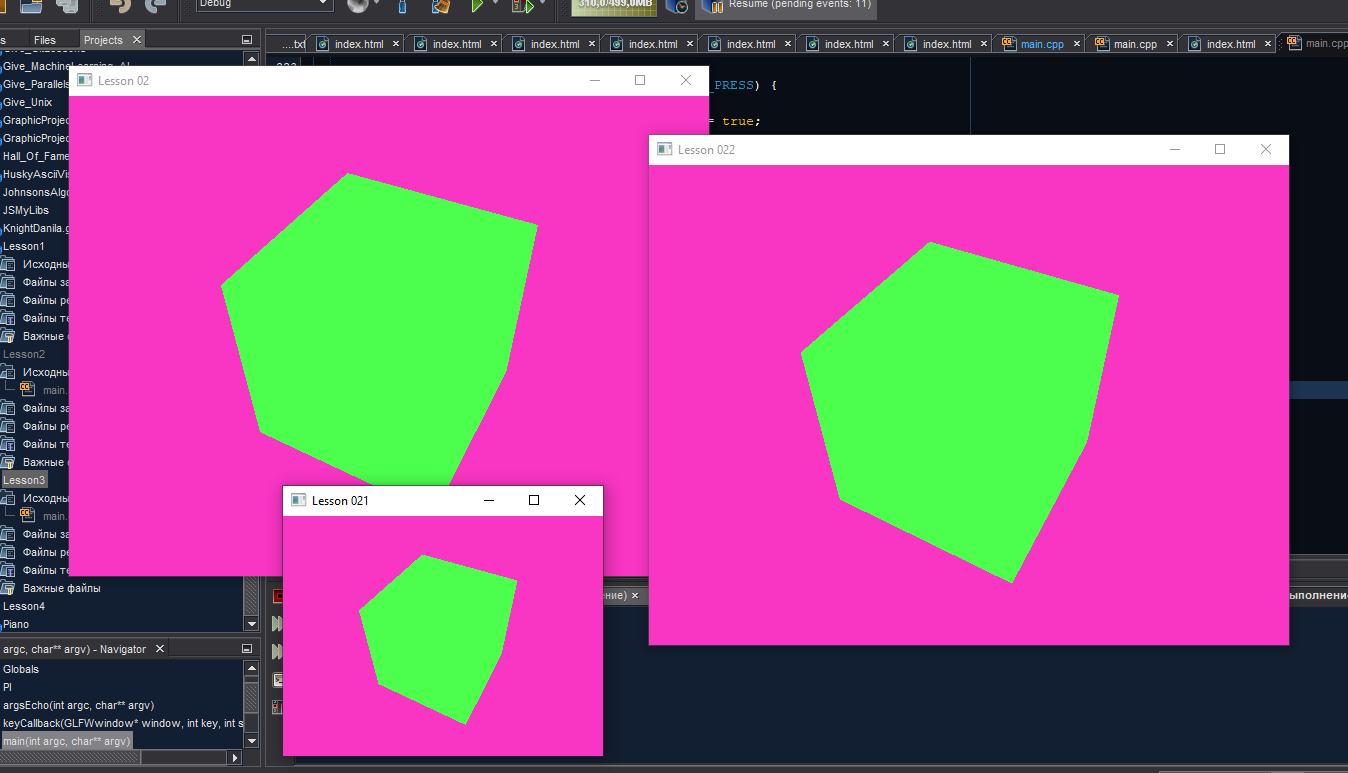


Рис. 5 – Результат виконання лабораторної роботи №3

# ЛАБОРАТОРНА РОБОТА 4

**Шейдер на мові GLSL – “Яскравий та Тьмяний”**

Шейдер - це комп'ютерна програма, яка виконується на графічному процесорі (GPU) та призначена для обробки графічних об'єктів. В OpenGL шейдери використовуються для створення реалістичних та деталізованих зображень, обчислення кольорів, освітлення та ефектів.

Шейдери виконуються на графічному процесорі та оперують зі спеціальною мовою програмування, такою як OpenGL Shading Language (GLSL). GLSL - це мова програмування, яка дозволяє програмістам створювати шейдери для OpenGL та визначати, які ефекти повинні бути застосовані до графічних об'єктів.

Шейдери використовуються для обробки графічних об'єктів на різних етапах графічного пайплайну. Наприклад, вершинний шейдер використовується для обчислення нових координат вершин трикутників, геометричний шейдер - для генерації нових геометричних форм на основі вхідних об'єктів, а фрагментний шейдер - для обчислення кольорів пікселів, які пізніше будуть відображені на екрані.

Узагальнюючи, шейдер - це програма, яка дозволяє програмістам створювати динамічні та вражаючі візуальні ефекти для графічних об'єктів у OpenGL, що дозволяє створювати реалістичні та деталізовані зображення.

**Мета роботи:** отримати практичні навички використання шейдерів у графічній бібліотеці OpenGL

**Завдання до виконання:**

1. Ознайомитись з можливостями "GLShader.h". Розібратися з особливостями "GLShader.h" класу.
2. Використовуючи шаблон програми Task04Src [<https://github.com/KnightDanila/GraphicProjects_OpenGL_Shaders_GLSL/tree/master/Lesson4/Task04Src>] - запишіть та запустить код з вашим простором імен.
3. Завантажити "BrightAndDim\_VertexShader.vs", "BrightAndDim\_FragmentShader.fs" до робочої папки;
4. Додайте вивід до консолі:

|  |
| --- |
| Task 4  Author: The student of the faculty of Applied Mathematics (Harry James Potter/Hermione Jean Granger) |

1. Створіть шейдер за допомогою ...::GraphCore::GLShader\* shaderBrightDim = new ...::GraphCore::GLShader("BrightAndDim\_VertexShader.vs", "BrightAndDim\_FragmentShader.fs");
2. Намалюйте об'єкт VBO (з лабораторної 2) - та додайте шейдер до нього
3. Додайте дію до клавіатури, щоб змінить колір у шейдері - додайте у main glfwSetKeyCallback(window, keyCallback); та напишіть :

|  |
| --- |
| void keyCallback(GLFWwindow \* window, int key, int scancode, int action, int mode) {  println((string) "key:" + key + "-scancode:" + scancode + "-action:" + action + "-mode:" + mode);  if (key == GLFW\_KEY\_SPACE && action == GLFW\_PRESS) {  println("SPACE");  //TO DO  }  } |

1. Оформити звіт, в якому описати хід виконання завдань та результат.

**Лістинг коду шаблону програми:**

**GLShader.h**

|  |
| --- |
| #ifndef GLSHADER\_H  #define GLSHADER\_H  //#include <glad/glad.h>  //#include <glm/glm.hpp>  #include <string>  #include <fstream>  #include <sstream>  #include <iostream>  namespace Knight3D {  namespace GraphCore {  class GLShader {  public:  unsigned int ID;  // constructor generates the shader on the fly  // ------------------------------------------------------------------------  GLShader() {  }  GLShader(const char\* vertexPath, const char\* fragmentPath, const char\* geometryPath = nullptr) {  // 1. retrieve the vertex/fragment source code from filePath  std::string vertexCode;  std::string fragmentCode;  std::string geometryCode;  std::ifstream vShaderFile;  std::ifstream fShaderFile;  std::ifstream gShaderFile;  // ensure ifstream objects can throw exceptions:  vShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);  fShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);  gShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);  try {  // open files  vShaderFile.open(vertexPath);  fShaderFile.open(fragmentPath);  std::stringstream vShaderStream, fShaderStream;  // read file's buffer contents into streams  vShaderStream << vShaderFile.rdbuf();  fShaderStream << fShaderFile.rdbuf();  // close file handlers  vShaderFile.close();  fShaderFile.close();  // convert stream into string  vertexCode = vShaderStream.str();  fragmentCode = fShaderStream.str();  // if geometry shader path is present, also load a geometry shader  if (geometryPath != nullptr) {  gShaderFile.open(geometryPath);  std::stringstream gShaderStream;  gShaderStream << gShaderFile.rdbuf();  gShaderFile.close();  geometryCode = gShaderStream.str();  }  } catch (std::ifstream::failure e) {  std::cout << "ERROR::SHADER::FILE\_NOT\_SUCCESFULLY\_READ" << std::endl;  }  const char\* vShaderCode = vertexCode.c\_str();  const char \* fShaderCode = fragmentCode.c\_str();  // 2. compile shaders  unsigned int vertex, fragment;  // vertex shader  vertex = glCreateShader(GL\_VERTEX\_SHADER);  glShaderSource(vertex, 1, &vShaderCode, NULL);  glCompileShader(vertex);  checkCompileErrors(vertex, "VERTEX");  // fragment Shader  fragment = glCreateShader(GL\_FRAGMENT\_SHADER);  glShaderSource(fragment, 1, &fShaderCode, NULL);  glCompileShader(fragment);  checkCompileErrors(fragment, "FRAGMENT");  // if geometry shader is given, compile geometry shader  unsigned int geometry;  if (geometryPath != nullptr) {  const char \* gShaderCode = geometryCode.c\_str();  geometry = glCreateShader(GL\_GEOMETRY\_SHADER);  glShaderSource(geometry, 1, &gShaderCode, NULL);  glCompileShader(geometry);  checkCompileErrors(geometry, "GEOMETRY");  }  // shader Program  ID = glCreateProgram();  glAttachShader(ID, vertex);  glAttachShader(ID, fragment);  if (geometryPath != nullptr)  glAttachShader(ID, geometry);  glLinkProgram(ID);  checkCompileErrors(ID, "PROGRAM");  // delete the shaders as they're linked into our program now and no longer necessery  glDeleteShader(vertex);  glDeleteShader(fragment);  if (geometryPath != nullptr)  glDeleteShader(geometry);  }  // activate the shader  // ------------------------------------------------------------------------  void use() {  glUseProgram(ID);  }  // utility uniform functions  // ------------------------------------------------------------------------  void setBool(const std::string &name, bool value) const {  glUniform1i(glGetUniformLocation(ID, name.c\_str()), (int) value);  }  // ------------------------------------------------------------------------  void setInt(const std::string &name, int value) const {  glUniform1i(glGetUniformLocation(ID, name.c\_str()), value);  }  // ------------------------------------------------------------------------  void setFloat(const std::string &name, float value) const {  glUniform1f(glGetUniformLocation(ID, name.c\_str()), value);  }  // ------------------------------------------------------------------------  void setVec2(const std::string &name, const glm::vec2 &value) const {  glUniform2fv(glGetUniformLocation(ID, name.c\_str()), 1, &value[0]);  }  void setVec2(const std::string &name, float x, float y) const {  glUniform2f(glGetUniformLocation(ID, name.c\_str()), x, y);  }  // ------------------------------------------------------------------------  void setVec3(const std::string &name, const glm::vec3 &value) const {  glUniform3fv(glGetUniformLocation(ID, name.c\_str()), 1, &value[0]);  }  void setVec3(const std::string &name, float x, float y, float z) const {  glUniform3f(glGetUniformLocation(ID, name.c\_str()), x, y, z);  }  // ------------------------------------------------------------------------  void setVec4(const std::string &name, const glm::vec4 &value) const {  glUniform4fv(glGetUniformLocation(ID, name.c\_str()), 1, &value[0]);  }  void setVec4(const std::string &name, float x, float y, float z, float w) {  glUniform4f(glGetUniformLocation(ID, name.c\_str()), x, y, z, w);  }  // ------------------------------------------------------------------------  void setMat2(const std::string &name, const glm::mat2 &mat) const {  glUniformMatrix2fv(glGetUniformLocation(ID, name.c\_str()), 1, GL\_FALSE, &mat[0][0]);  }  // ------------------------------------------------------------------------  void setMat3(const std::string &name, const glm::mat3 &mat) const {  glUniformMatrix3fv(glGetUniformLocation(ID, name.c\_str()), 1, GL\_FALSE, &mat[0][0]);  }  // ------------------------------------------------------------------------  void setMat4(const std::string &name, const glm::mat4 &mat) const {  glUniformMatrix4fv(glGetUniformLocation(ID, name.c\_str()), 1, GL\_FALSE, &mat[0][0]);  }  private:  // utility function for checking shader compilation/linking errors.  // ------------------------------------------------------------------------  void checkCompileErrors(GLuint shader, std::string type) {  GLint success;  GLchar infoLog[1024];  if (type != "PROGRAM") {  glGetShaderiv(shader, GL\_COMPILE\_STATUS, &success);  if (!success) {  glGetShaderInfoLog(shader, 1024, NULL, infoLog);  std::cout << "ERROR::SHADER\_COMPILATION\_ERROR of type: " << type << "\n" << infoLog << "\n -- --------------------------------------------------- -- " << std::endl;  }  } else {  glGetProgramiv(shader, GL\_LINK\_STATUS, &success);  if (!success) {  glGetProgramInfoLog(shader, 1024, NULL, infoLog);  std::cout << "ERROR::PROGRAM\_LINKING\_ERROR of type: " << type << "\n" << infoLog << "\n -- --------------------------------------------------- -- " << std::endl;  }  }  }  };  };  };  #endif |

**BrightAndDim\_VertexShader.vs**

|  |
| --- |
| #version 330 core  layout (location = 0) in vec3 aPos; // the position variable has attribute position 0    out vec4 vertexColor; // specify a color output to the fragment shader  void main()  {  gl\_Position = vec4(aPos, 1.0); // see how we directly give a vec3 to vec4's constructor  vertexColor = vec4(0.5, 0.0, 0.0, 1.0); // set the output variable to a dark-red color  } |

**BrightAndDim\_FragmentShader.fs**

|  |
| --- |
| #version 330 core  out vec4 FragColor;    in vec4 vertexColor; // the input variable from the vertex shader (same name and same type)  void main()  {  FragColor = vertexColor;  } |

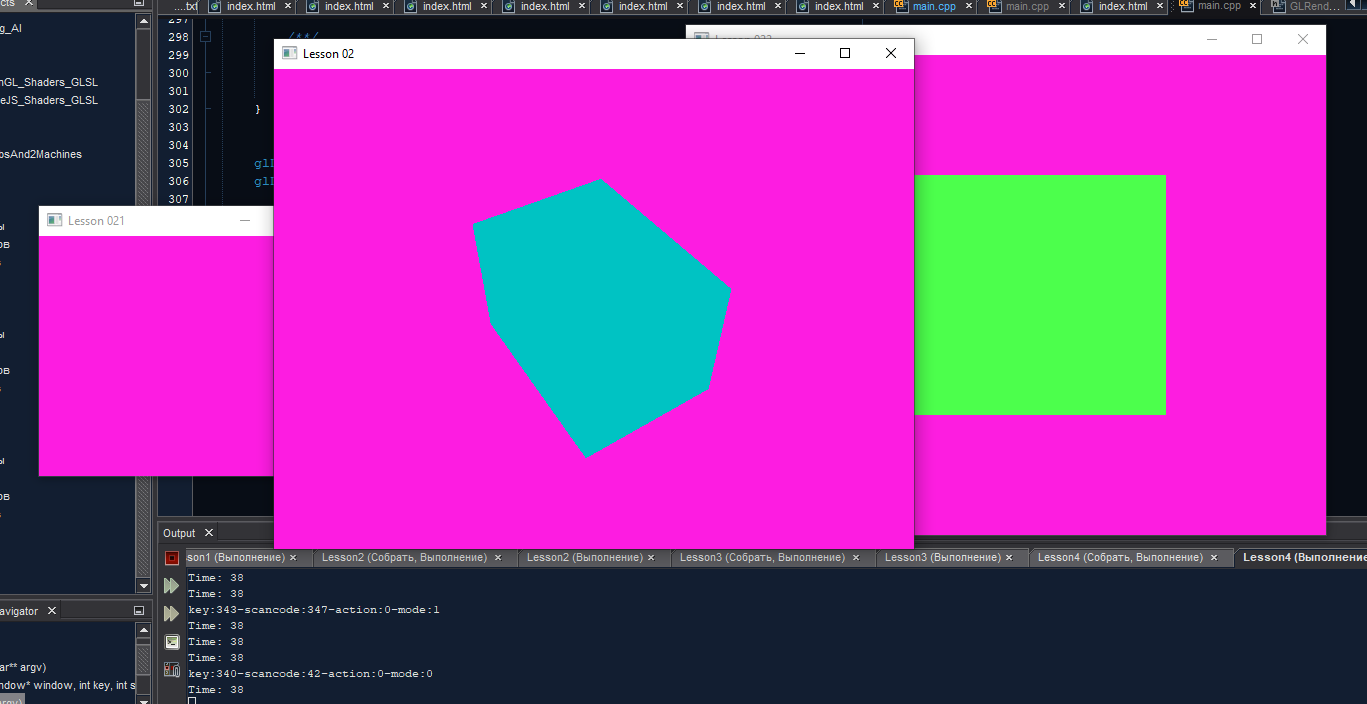


Рис. 6 – Результат виконання лабораторної роботи №4

# ДОДАТКИ

**Додаток 1.** Довідкова інформація щодо функцій OpenGL

**glfwWindowHint**

- це функція з бібліотеки GLFW, яка використовується для встановлення параметрів вікна GLFW до його створення. Ці параметри визначають розміри вікна, режим відображення, наявність буфера глибини та інші характеристики вікна.

Функція приймає два аргументи: ім'я параметра та його значення. Наприклад, для встановлення розмірів вікна за замовчуванням можна використовувати наступний код:

|  |
| --- |
| glfwWindowHint(GLFW\_RESIZABLE, GLFW\_TRUE);  glfwWindowHint(GLFW\_CLIENT\_API, GLFW\_OPENGL\_API);  glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);  glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3); |

У цьому прикладі ми встановлюємо параметри вікна для того, щоб вікно можна було змінювати (значення GLFW\_TRUE параметру GLFW\_RESIZABLE), щоб вікно використовувало API OpenGL (значення GLFW\_OPENGL\_API параметру GLFW\_CLIENT\_API), та мало контекст OpenGL версії 3.3 (значення 3 та 3 для параметрів GLFW\_CONTEXT\_VERSION\_MAJOR та GLFW\_CONTEXT\_VERSION\_MINOR відповідно).

Ці параметри можуть бути встановлені до створення вікна за допомогою функції glfwWindowHint. Деякі параметри можуть бути змінені після створення вікна, але для більшості параметрів зміна після створення вікна не дає ефекту.

**glfwCreateWindow**

- це функція з бібліотеки GLFW, яка використовується для створення вікна GLFW з відповідним контекстом OpenGL. Функція приймає кілька аргументів, які визначають параметри вікна, такі як розміри, заголовок вікна та параметри контексту OpenGL.

Ось приклад використання функції glfwCreateWindow для створення вікна розміром 800 на 600 пікселів з назвою "My GLFW Window":

|  |
| --- |
| GLFWwindow\* window = glfwCreateWindow(800, 600, "My GLFW Window", NULL, NULL);  if (!window)  {  // Обробка помилок створення вікна  } |

У цьому прикладі функція створює вікно розміром 800 на 600 пікселів з назвою "My GLFW Window". Параметри, які передаються як NULL, вказують на використання значень за замовчуванням для параметрів контексту OpenGL та монітора.

Функція повертає вказівник на створене вікно, який можна використовувати для подальшої роботи з вікном, наприклад, для відображення графіки, обробки подій вікна, тощо. Якщо створення вікна не вдалося, функція повертає значення NULL, і програміст повинен обробити цю помилку.

**glfwTerminate**

- це функція з бібліотеки GLFW, яка використовується для завершення роботи GLFW та вивільнення всіх асоційованих з ним ресурсів. Ця функція має бути викликана перед завершенням програми, щоб впевнитися, що всі створені вікна та інші ресурси були правильно вивільнені та щоб уникнути витоку пам'яті.

У цьому прикладі функція glfwTerminate викликається без передачі будь-яких параметрів. Після виконання цієї функції всі створені вікна будуть закриті, контексти OpenGL будуть видалені та всі інші асоційовані з GLFW ресурси будуть вивільнені.

Важливо викликати цю функцію в кінці програми після закриття всіх створених вікон. Якщо ця функція не буде викликана, то програма може не коректно вивільнити пам'ять та інші ресурси, що може призвести до витоку пам'яті та інших проблем з продуктивністю.

**glClearColor**

- це функція з бібліотеки OpenGL, яка використовується для встановлення значення колірного буфера очищення (clear color) для наступної операції очищення екрана (clear screen). Колірний буфер очищення - це буфер, який використовується для зберігання кольорів, які потім будуть відображені на екрані.

Ось приклад використання функції glClearColor для встановлення колірного буфера очищення в чорний колір (RGB: 0, 0, 0):

|  |
| --- |
| glClearColor(0.0f, 0.0f, 0.0f, 1.0f); |

У цьому прикладі функція glClearColor викликається з чотирма параметрами: червоним (R), зеленим (G) та синім (B) кольорами відповідно (в даному випадку всі вони дорівнюють 0), а також альфа-каналом (A), який визначає прозорість (в даному випадку дорівнює 1, що означає повну непрозорість).

Після виклику цієї функції колірний буфер очищення буде встановлений в вказаний колір, і наступна операція очищення екрана буде видаляти колір екрана до заданого значення.

**glClear**

- це функція з бібліотеки OpenGL, яка використовується для очищення колірного буфера (color buffer) та глибинного буфера (depth buffer) відповідно до встановленого значення колірного буфера очищення (clear color) та глибинного буфера очищення (clear depth).

Ось приклад використання функції glClear для очищення колірного буфера та глибинного буфера:

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

У цьому прикладі функція glClear викликається з параметром, який вказує, які буфери потрібно очистити. В даному випадку використовується комбінація флагів GL\_COLOR\_BUFFER\_BIT та GL\_DEPTH\_BUFFER\_BIT, що означає, що необхідно очистити колірний буфер та глибинний буфер відповідно.

Ця функція зазвичай використовується на початку кожного кадру, щоб очистити екран перед наступною операцією малювання (rendering). Очищення колірного буфера дозволяє видалити попередні кадри, щоб малювати нові кадри, а очищення глибинного буфера дозволяє визначити, які об'єкти потрібно відобразити на передньому плані та які на задньому плані.

# ІНФОРМАЦІЙНІ РЕСУРСИ

Презентації-<https://knightdanila.github.io/GraphicProjects_OpenGL_Shaders_GLSL>

GitHb - <https://github.com/KnightDanila/GraphicProjects_OpenGL_Shaders_GLSL>

OpenGLBook - A Free OpenGL Programming Book - <http://openglbook.com/>

The OpenGL® Programming Guide 9th Edition - (<http://www.opengl-redbook.com/>

Khronos-OpenGL-Refpages - <https://www.khronos.org/registry/OpenGL-Refpages/gl4/>

LearnOpenGL Book - https://learnopengl.com/book/book\_pdf.pdf

[Learning Modern 3D Graphics Programming - <https://www.cse.chalmers.se/edu/year/2018/course/TDA361/LearningModern3DGraphicsProgramming.pdf>

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Dunn, Fletcher, and Ian Parberry. 3D Math Primer for Graphics and Game Development, 2nd Ed. Jones & Bartlett Learning, ISBN 978-1-4496-0086-4. 2011, 846 pages.
2. Mark de Berg Marc van Kreveld. Mark Overmars Otfried. Computational Geometry: Algorithms and Applications, Third Edition. Springer, 2008. ISBN 978-3-540-77973-5. 396 сторінок.
3. Lengyel, Eric. Mathematics for 3D Game Programming and Computer Graphics, Third Edition. Cengage Learning PTR, 2011. ISBN 978-1-4354-5886-4. 624 pages.
4. Gunn, Christopher. Mathematics for Game Developers. A K Peters/CRC Press, 2005. ISBN 1-56881-227-7. 603 pages.
5. Nystrom, Robert. Game Programming Patterns. Genever Benning, 2014. ISBN-13: 978-0990582908, 355 pages.
6. McKesson, Jason L. Learning Modern 3D Graphics Programming: Techniques and Tools Used in the Game Industry. Addison-Wesley Professional, 2012, 576 pages.
7. Marrin, John. The OpenGL® Programming Guide: The Official Guide to Learning OpenGL®, Version 4.5 with SPIR-V, 9th Edition. Addison-Wesley Professional, 2016.