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

М Е Т О Д И Ч Н І В К А З І В К И
ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ
ЧАСТИНА 2. OPENGL

Дисципліна: Комп’ютерна графіка

Розробник:
Державний вищий навчальний заклад
«Донецький національний технічний університет»
факультет комп’ютерних наук і технологій
кафедра прикладної математики і інформатики

2018

УДК 004.032.6 : 004.92

Методичні вказівки до виконання лабораторних робіт за модулем «OpenGL» дисципліни «Комп’ютерна графіка» підготовки бакалаврів спеціальності 121 «Інженерія програмного забезпечення» / Укладачі: Є.О. Башков, П.О. Ануфрієв. – Покровськ: ДонНТУ, 2018. – 74 с.

Укладачі:
Є.О. Башков, професор кафедри ПМІ, д.т.н., професор;
П.О. Ануфрієв, асистент кафедри ПМІ.

ЗМІСТ

1. Вступ. Загальні положення

Комплекс лабораторних робот модуля «2D графіка в середовищі OpenGL» виконується відповідно навчально-освітній програмі має на меті відпрацювання навичок і вмінь використання базових можливостей середовища OpenGL [2] для генерації двовимірних зображень та передбачає послідовне виконання низки практичних завдань.

Самостійна робота студентів передбачає вивчення додаткової літератури [3, 4], підготовку до лабораторних робіт, а також підготовку та оформлення звітів за результатами виконання лабораторних робіт.

1.1. Графік виконання лабораторних робіт

Номер тижня	Виконання	Оцінювання (максимальний бал)
9-10	Лабораторна робота № 5	**3**
11-12	Лабораторна робота № 6	**3**
12-14	Лабораторна робота № 7	**3**
15-16	Лабораторна робота № 8	**3**
Максимальний сумарний бал		**12**

1.2. Оцінка виконання комплексу лабораторних робіт

Критерії оцінювання виконання лабораторної роботи наведені в додатку А.

Критерії оцінювання заключного звіту наведені в додатку Б.

1.3. Контактні дані для on-line допомоги та консультування

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

1.4. Необхідне обладнання

Для виконання лабораторних робіт бажано використовувати ПК із встановленою системою Windows 10 x86-64 та інтегрованим середовищем розробки Microsoft Visual Studio 2017 з встановленим пакетом Visual C++.

Мінімальні вимоги до технічних характеристик ПК:

- Процесор Intel Core 2 Duo, x86-64;
- Оперативна пам'ять 4 Gb;
- Не менше 10 Gb вільного простору на HDD або SSD (для встановлення Visual Studio);
- Графічна карта, яка підтримує не менше, ніж OpenGL 3.0;
- Інтернет з’єднання.

2. Попередні підготовчі кроки до виконання комплексу лабораторних робіт

2.1. Збірка бібліотек

Перше, що потрібно зробити для створення графічних об’єктів та графіки в цілому - це створення контексту OpenGL та вікна програми, в яких малюється графіка. Однак операції створення контексту та вікон специфічні для кожної операційної системи, і OpenGL цілеспрямовано намагається абстрагуватися від цих операцій. Це означає, що треба створити вікно, визначити контекст і працювати з користувацьким вводом власноруч [2, 3].

Існує безліч бібліотек, які забезпечують подані функціональності. Ці бібліотеки зберігають всю специфічну роботу з операційною системою та надають вікно та контекст OpenGL для відтворення. Деякі з найбільш популярних бібліотек - GLUT, SDL, SFML та GLFW. Для лабораторних робіт буде використовуватися бібліотека GLFW [2, 3].

Перед тим, як розпочати роботу, створіть директорії для завантаження і роботи з бібліотеками, а також для майбутніх лабораторних робіт (рис. 1.1). Всі лабораторні роботи будуть зберігатися в одному рішенні Visual Studio. Кожне завдання лабораторної роботи буде як новий проект Visual Studio. Рекомендуємо одразу переключити вашу версію Visual Studio на англійську мову. Пам’ятайте, ви робите лабораторні роботи для ознайомлення з роботою OpenGl.

Лабораторні роботи № 5 – 7 базуються на матеріалах з сайту https://learnopengl.com/

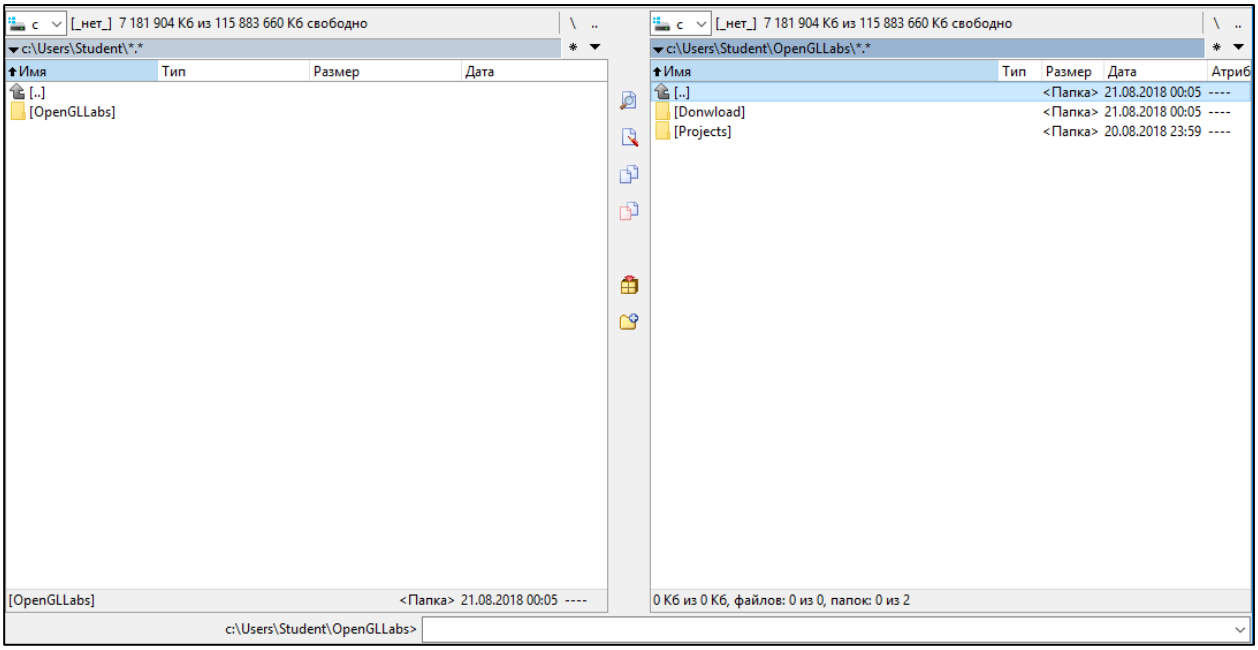


Рисунок 1.1 – Директорії проекту

2.2. Бібліотека GLFW

GLFW - це відкрита бібліотека, написана на C, спеціально орієнтована для надання OpenGL самого необхідного для відтворення на екрані. Вона дозволяє нам створювати контекст OpenGL, створювати та відкривати вікна, визначати параметри вікна та обробляти вхід користувача [2].

2.2.1. Завантаження GLFW

- Перейдіть на офіційну сторінку завантаження бібліотеки http://www.glfw.org/download.html;
- Обрати «***Source package***» (рис. 1.2)**;**
- Завантажити та розпакувати в папку …\Donwload;
- В папці …\Donwload\glfw-3.2.1\... створити папку …\build;

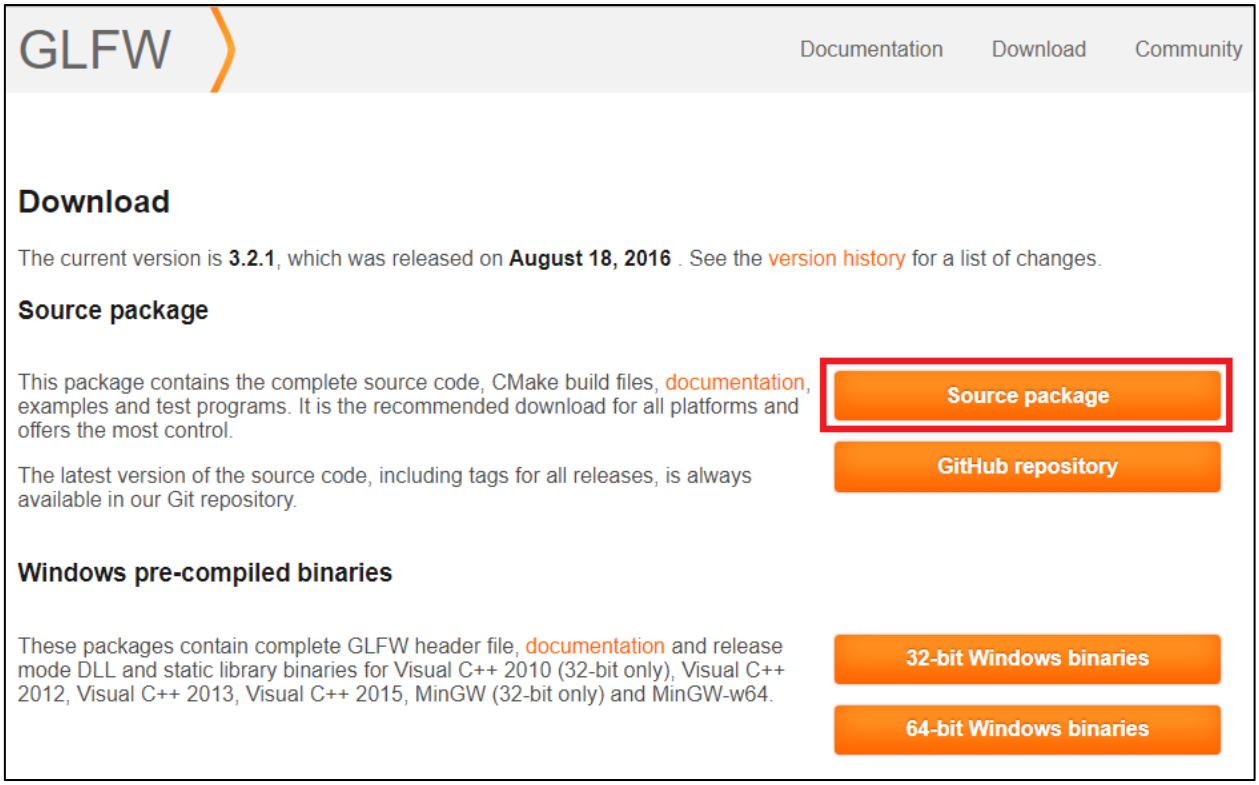


Рисунок 1.2 – Завантаження бібліотеки GLFW

2.2.2. Завантаження CMake

CMake - це інструмент, який генерує файли проекту/рішення за вибором користувача (наприклад, Visual Studio, Code :: Blocks, Eclipse) з колекції файлів вихідного коду за допомогою заздалегідь визначених сценаріїв CMake. Це дозволяє генерувати файл проекту Visual Studio 2017 із вихідного пакета GLFW, який використовується для складання бібліотеки[2, 3].

Для завантаження перейдіть на офіційну сторінку завантажувача CMake https://cmake.org/download/ та оберіть Win32 версію (рис. 1.3).

Далі встановіть CMake на свій ПК.

Примітка: Рекомендуємо під час виконання лабораторних робіт використовувати 32-бітні версії бібліотек.

Рисунок 1.3 – Завантаження CMake

2.2.3. Збірка бібліотеки GLFW

- Запустіть CMake;
- В якості директорії з вихідним кодом вказати кореневу папку GLFW …\Donwload\glfw-3.2.1\... (рис. 1.4);
- В якості директорії для бінарних файлів вказати створену папку …\Donwload\glfw-3.2.1\build\... (рис. 1.4);
- Натиснути кнопку ***«Configure»;***
- Натиснути кнопку ***«Finish»;***
- Вікно конфігурації стане червоним, ще раз натиснути ***«Configure»*** (рис. 1.5)***;***
- Далі натиснути кнопку ***«Generate»;***
- Натиснути кнопку ***«Open Project»;***

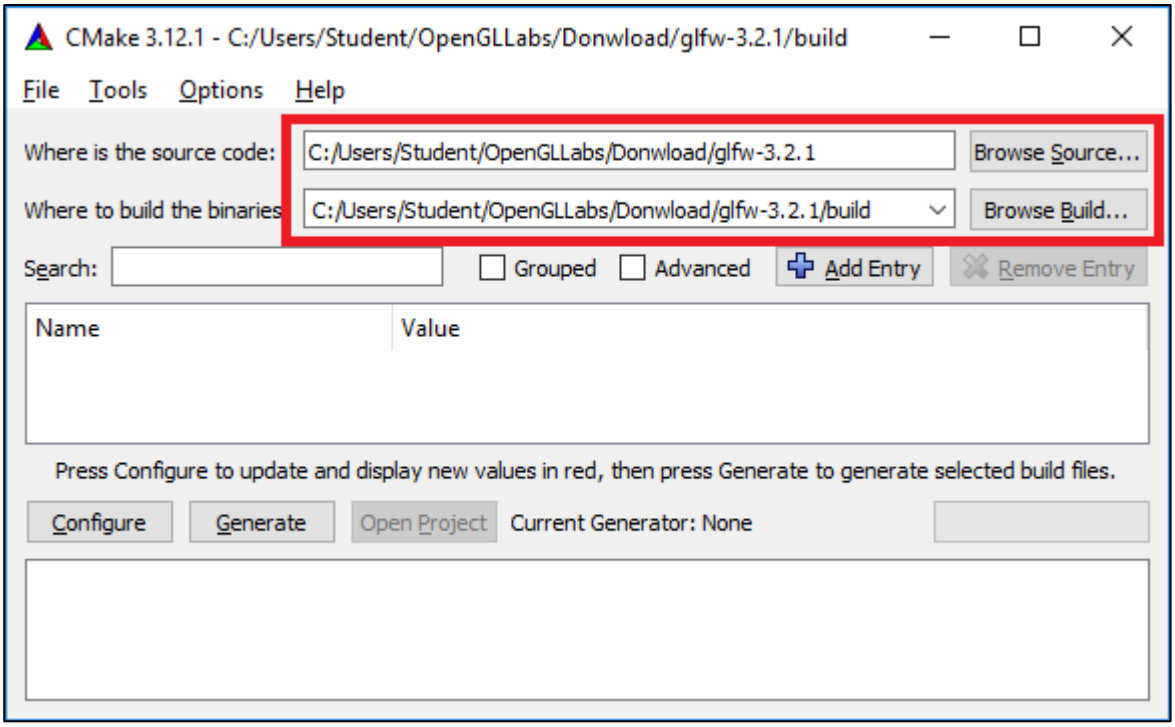


Рисунок 1.4 – Вибір директорій для зборки бібліотеки GLFW

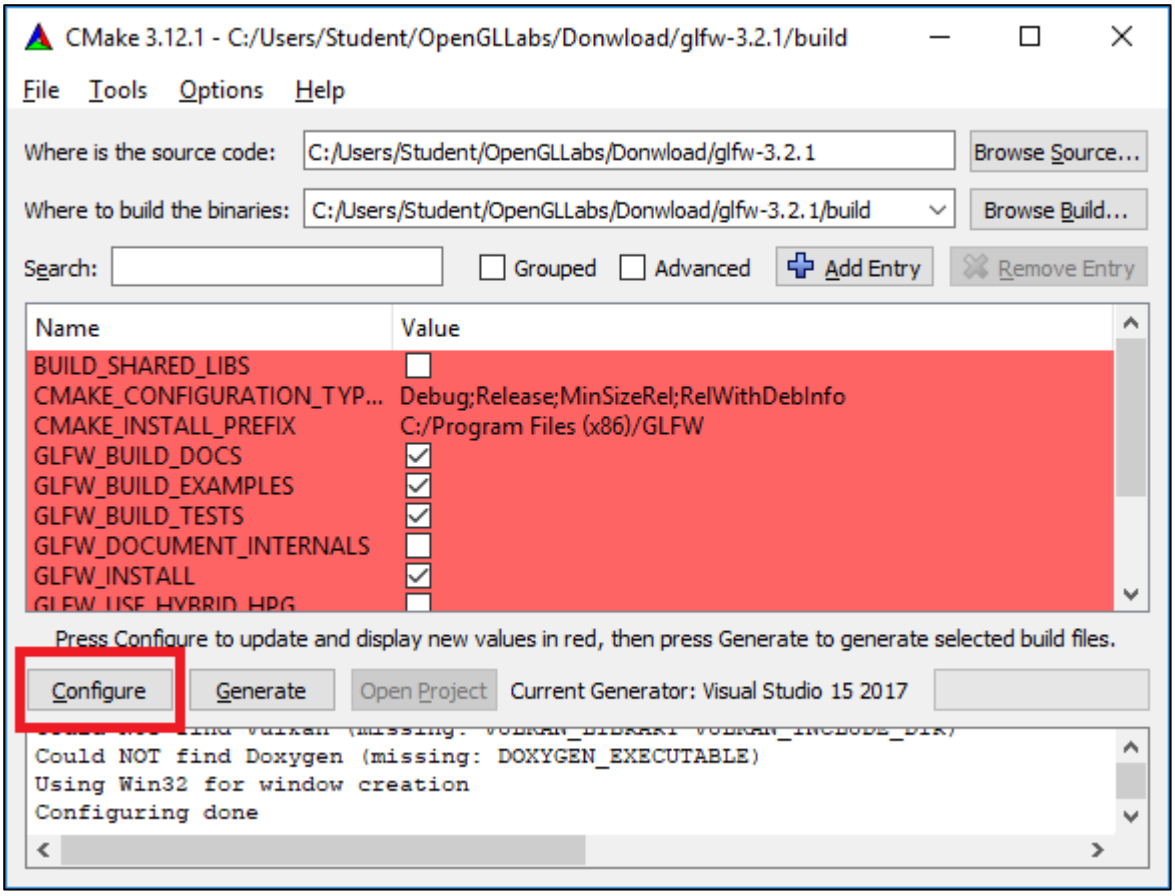


Рисунок 1.5 – Продовження конфігурації CMake

- Відкриється вікно Visual Studio з проектами GLFW. У вкладці «***Solution Explorer»*** оберіть правою кнопкою GLFW3\examples\wave та натисніть «***Set as StartUp Project»;***
- Натисніть «Ctrl+F5» та побудуйте проект;
- Якщо відкрилось вікно **“Wave Simulation”**, то зборка GLFW пройшла успішно (рис. 1.6). Можна закрити Visual Studio;

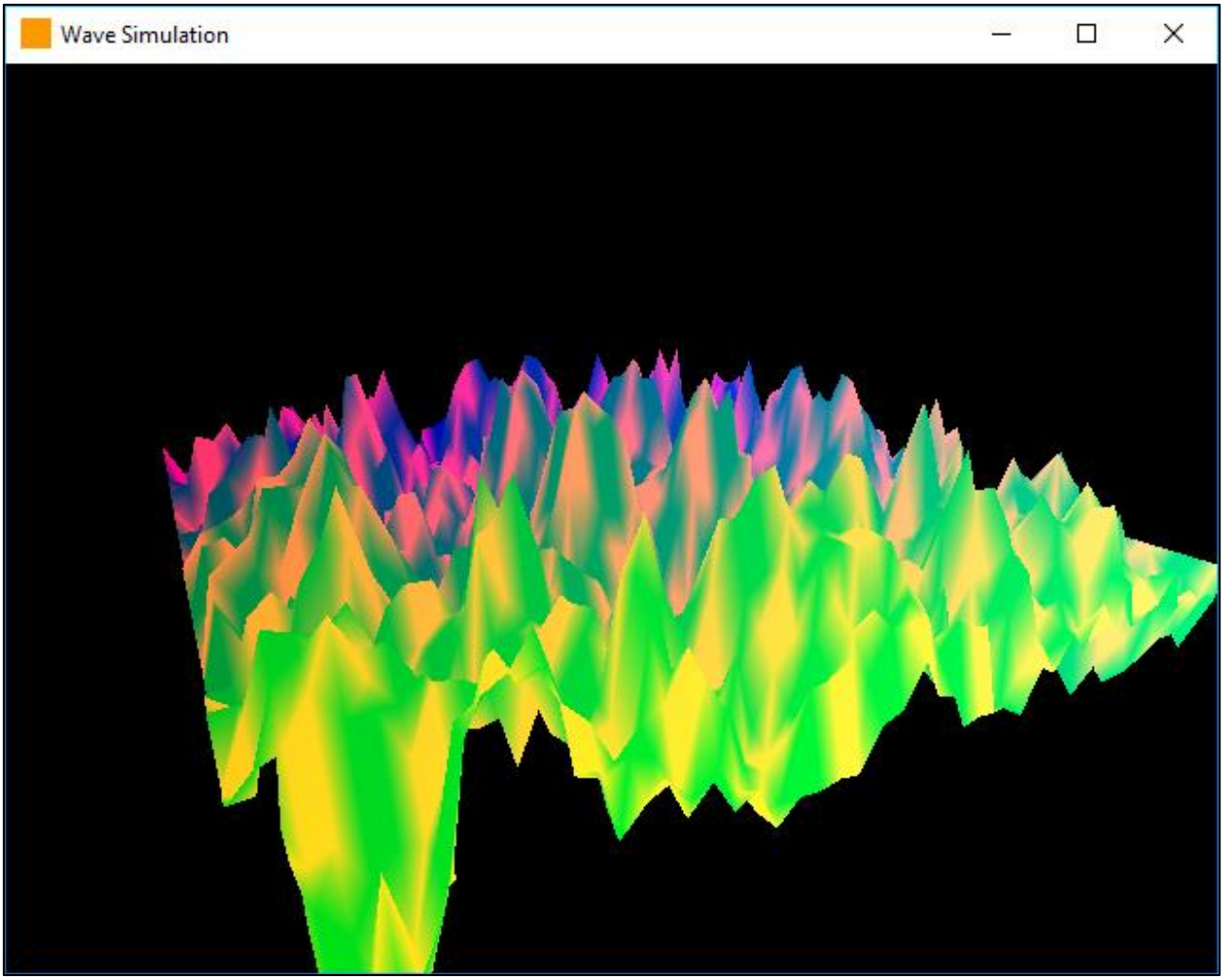


Рисунок 1.6 – Проект “**Wave Simulation”**

- В папці …\Users\Student\OpenGLLabs\... створити папку …\Libraries\..., а в ній створити папки …\Lib\..., …\Include \...,

та …\Src \... (рис. 1.7);

Примітка: Рекомендований підхід - створити набір каталогів у вибраному місці, який містить всі заголовки файлів / бібліотеки / вихідні файли, до яких можна послатися при використанні IDE / компілятора. Це дає змогу організовано розташувати сторонні бібліотеки в одній директорії …\Libraries\..., та посилатися на неї при створенні проектів.

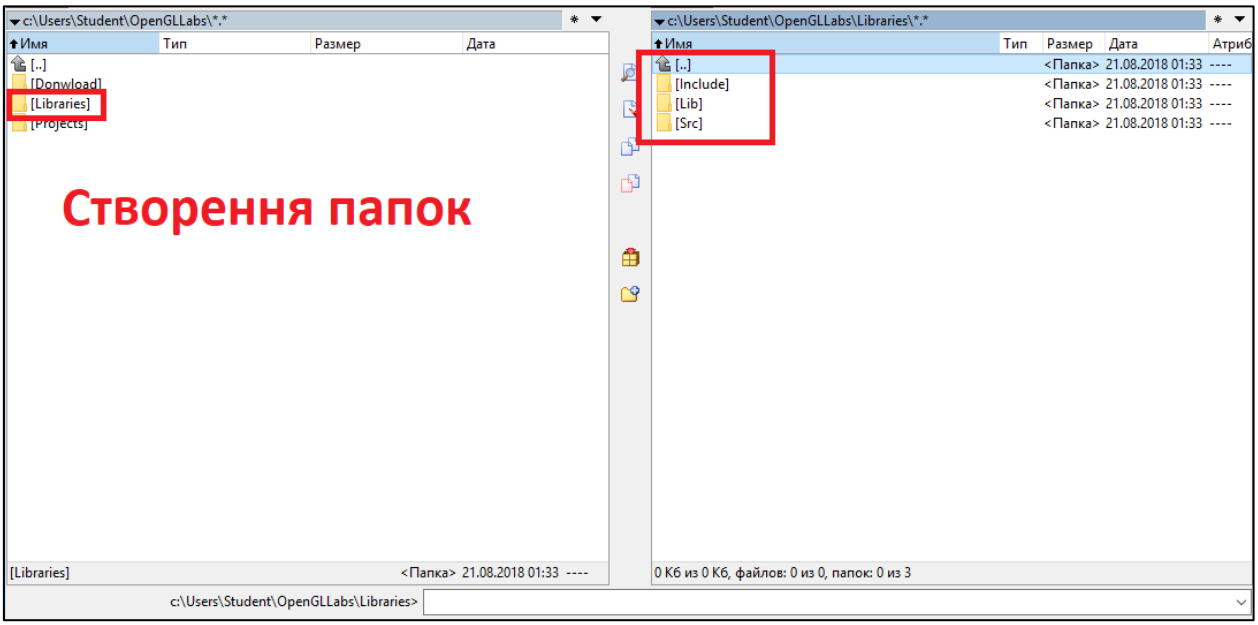


Рисунок 1.7 – Створення директорій для зберігання бібліотек, заголовків та вихідних файлів

- Скопіювати заголовки та бібліотеки до нашої директорії:

А) з папки …\OpenGLLabs\Donwload\glfw-3.2.1\build\src\Debug\... скопіювати …\glfw3.lib до …\OpenGLLabs\Libraries\Lib\...;

Б) з папки …\OpenGLLabs\Donwload\glfw-3.2.1\include\GLFW\... скопіювати всі файли до …\OpenGLLabs\Libraries\Include\....

2.3. Бібліотека GLEW

Оскільки OpenGL - це стандарт / специфікація, виробник драйверів повинен ввести специфікацію драйвера, яку підтримує конкретна графічна

карта. Оскільки існує безліч різних версій драйверів OpenGL, розташування більшості його функцій не відоме під час компіляції та потребує запиту під час виконання. Тоді завдання розробника полягає в отриманні необхідних функцій і збереженні їх у покажчиках функцій для подальшого використання [2, 3]. Отримання цих локацій залежить від ОС, і в Windows це виглядає приблизно так:

```
// Визначаємо прототип функції
typedef void (*GL_GENBUFFERS) (GLsizei, GLuint*);

// Знаходимо цю функцію в реалізації і зберігаємо покажчик на неї
GL_GENBUFFERS                    glGenBuffers                    =
GL_GENBUFFERS)wglGetProcAddress("glGenBuffers");

// Нормально викликаємо функцію
GLuint buffer;
glGenBuffers(1, &buffer);
```

Код виглядає досить непростим, і необхідність робити отримання адреса для кожної OpenGL функції робить цей процес занадто ускладненим. Для поліпшення цього процесу існують бібліотеки, що реалізують цю динамічну зв'язку і одна з найпопулярніших бібліотек – GLEW [2, 3].

GLEW розшифровується як OpenGL Extension Wrangler Library і управляє всією роботою, про яку було сказано. GLEW являє собою бібліотеку, яку потрібно також зібрати.

2.3.1. Завантаження та збірка GLEW

- Завантажити бібліотеку GLEW з офіційного сайту http://glew.sourceforge.net/index.html та розпакувати (рис.1.8);

Примітка: Рекомендуємо під час виконання лабораторних робіт використовувати 32-бітні версії бібліотек.

- Перейти до …\OpenGLLabs\Donwload\glew-2.1.0\build\vc12\...

та запустити файл **“glew.sln”.** Відкриється вікно Visual Studio;

- Якщо відкриється вікно “**Retarget Projects**”, натиснути кнопку ***«OK»*** (рис. 1.9);

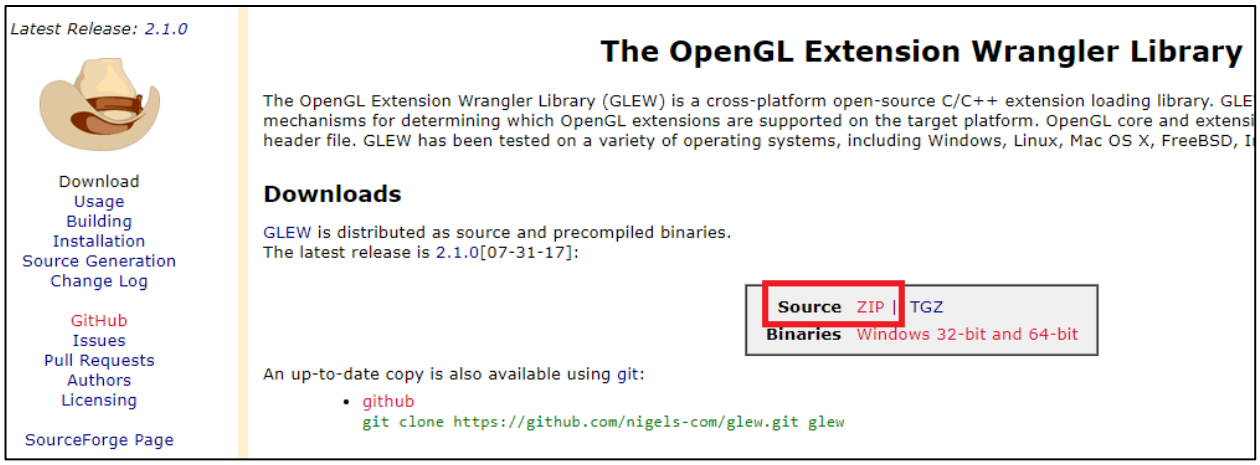


Рисунок 1.8 – Завантаження бібліотеки GLEW

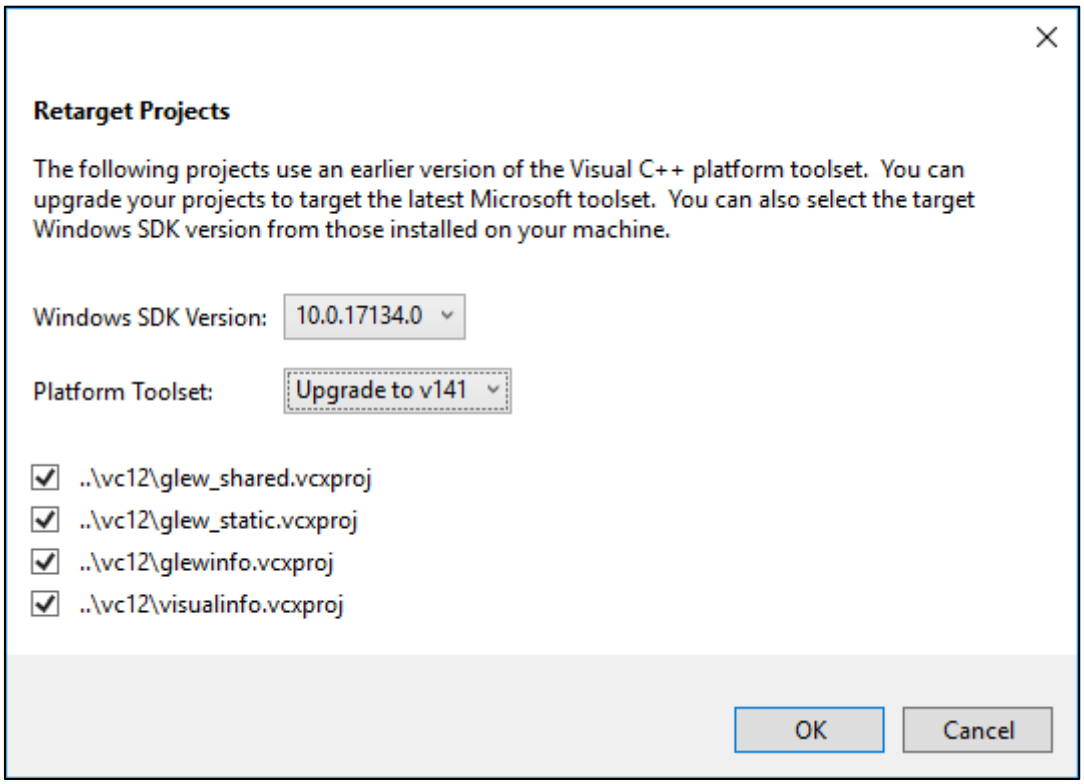


Рисунок 1.9 – Оновлення файлів проекту GLEW

- Натиснути вкладку ***«Build»\«Build solution»*** або клавіши «Ctrl+Shift+B»;

- Якщо у лівому нижньому куті повідомлення **“Build succeeded”,** можна закрити Visual Studio;
- Скопіювати заголовки, бібліотеки та вихідні файли до директорій:

А) з папки …\OpenGLLabs\Donwload\glew-2.1.0\lib\Debug\Win32\... скопіювати всі файли до …\OpenGLLabs\Libraries\Lib\...;

Б) з папки …\OpenGLLabs\Donwload\glew-2.1.0\include\GL\... скопіювати папку …\GL\... до …\OpenGLLabs\Libraries\Include\...;

В) з папки …\OpenGLLabs\Donwload\glew-2.1.0\src\... скопіювати всі файли до …\OpenGLLabs\Libraries\Src\...;

2.4. Створення першого проекту

2.4.1. Зв’язка (лінкування) бібліотек, заголовків та вихідних файлів

Для зв’язки створених нами бібліотек з нашими проектами треба зробити наступні кроки:

- Відкрити Visual Studio і створити новий проект. Для цього треба обрати вкладку «**Visual++»\«Empty Project»;**

Примітка: Рекомендації щодо назв:

- Name: Lab5;
- Location: …\OpenGLLabs\Projects\;
- Solution name: OpenGL;

Після створення проекту перейменуйте рішення на **“OpenGl”.** Одразу визначте ієрархію вашого проекту. У вашому проекті буде тільки одно рішення з одною назвою, а лабораторні роботи будуть проектами, тобто будуть додаватися. Через зміну назв можуть бути помилки.

Важливо: Наступні шляхи будуть обов'язковими при кожному створенні нового проекту:

- Натиснути правою кнопкою на **"Lab5"** та обрати **«Properties»**
- Обрати вкладку **"VC++ Directories"** та натиснути на **"Include Directories",** далі натиснути на випадну строку та натиснути на **«Edit».** Обрати шляхом **"Include Directories"** папку …\OpenGLLabs\Libraries\Include (рис. 2.1);

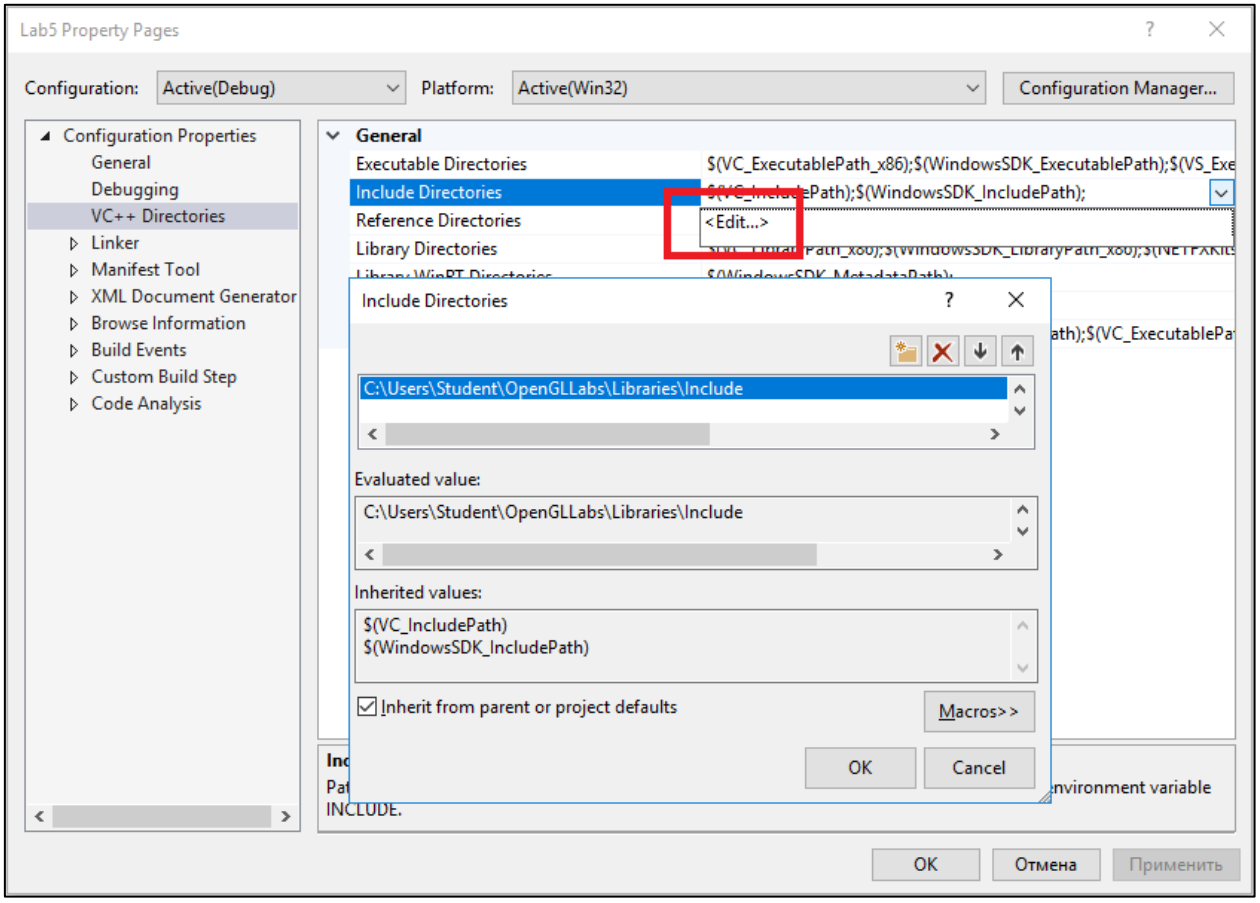


Рисунок 2.1 – Вибір директорій з заголовками

- Ті ж самі шляхи зробити для **"Library Directories"**, проте обрати папку …\OpenGLLabs\Libraries\Lib
- Далі обрати вкладку **"Linker"\"Input"**, натиснути на **"Additional Dependencies"**, далі натиснути на випадну строку та натиснути на **«Edit»**.

Прописати у верхній області вікна **“Additional Dependencies”** наступні бібліотеки: **opengl32.lib, glfw3.lib, glew32sd.lib**, та підтвердити дії (рис. 2.2);

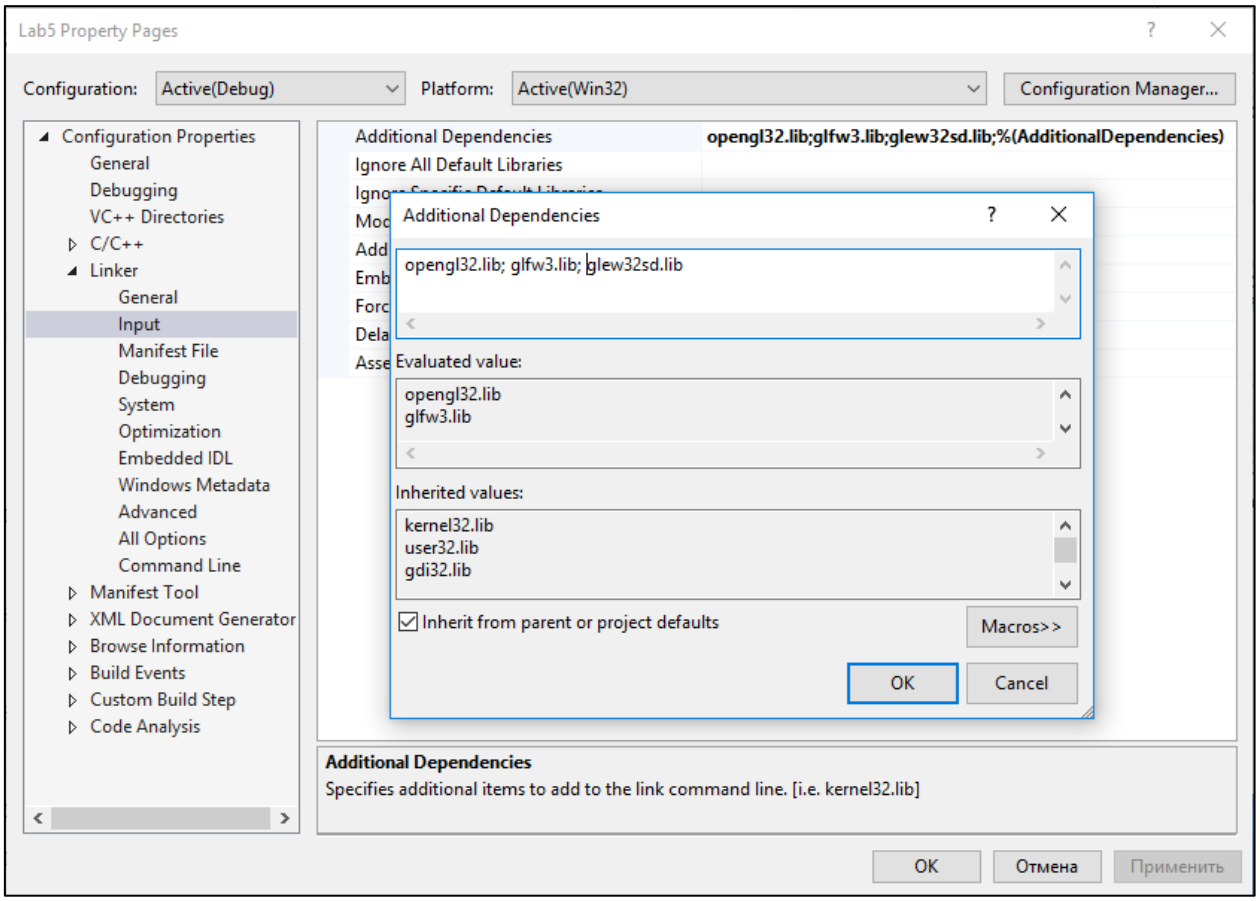


Рисунок 2.2 – Лінкування (зв’язка) бібліотек

- Натиснути правою кнопкою на **“Lab5”**, обрати **«Add»\«Existing Item»**, перейти до …\OpenGLLabs\Libraries\Src\... та обрати **“glew.c”**;
- Натиснути правою кнопкою на **“Lab5”**, обрати **«Add»\«New Item»\ «C++ File (.cpp)»** та назвати **“FirstWindow”**
- Відкрити через **“Solution Explorer”** файл **“FirstWindow.cpp”** та скопіювати наступний код та натиснути «Ctrl+F5»:

```
#define GLEW_STATIC
```

```
#include <GL/glew.h>;
#include <glfw3.h>;

#include <iostream>;

int main()
{

	glfwInit();
	glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
	glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
	GLFWwindow* window = glfwCreateWindow(800, 600, "FirstWindow", nullptr, nullptr);

	glfwMakeContextCurrent(window);
	glewExperimental = GL_TRUE;

	int width, height;
	glfwGetFramebufferSize(window, &width, &height);
	glViewport(0, 0, width, height);

	while (!glfwWindowShouldClose(window))
	{

		glfwPollEvents();
		glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
		glClear(GL_COLOR_BUFFER_BIT);
		glfwSwapBuffers(window);
	}

	glfwTerminate();
	return 0;
}
```

Якщо все було підключено та злінковано правильно, відкриється наступне вікно (рис. 2.3). За допомогою наших зібраних бібліотек було створено перший проект.

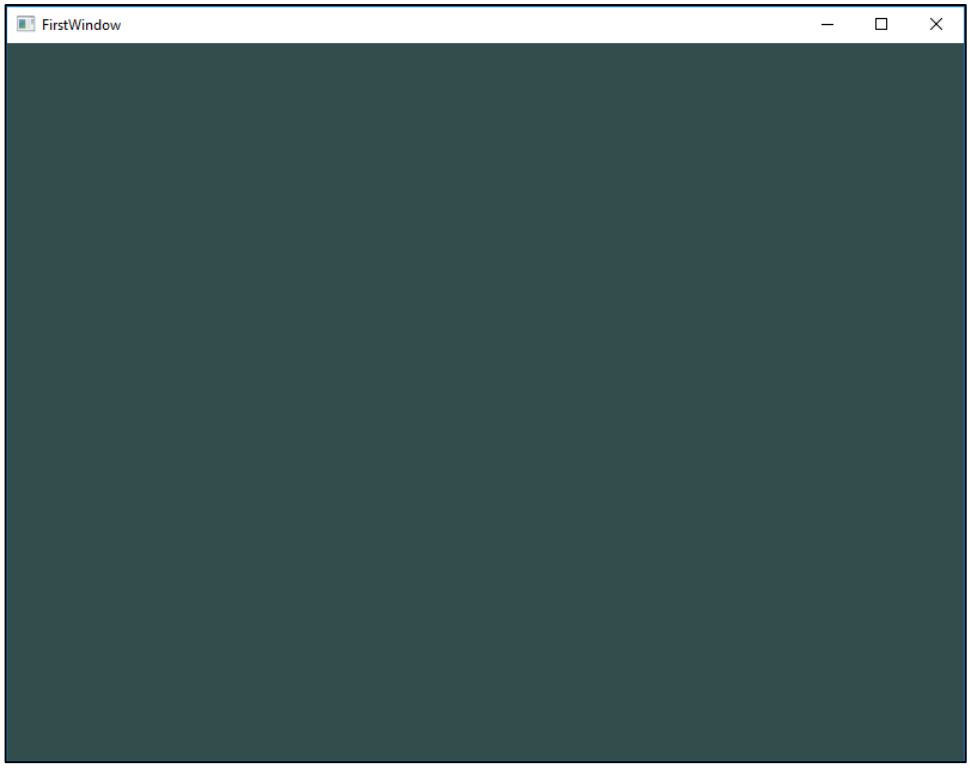


Рисунок 1.3 – Перше вікно лабораторної роботи

2.4.2. Створення вікна

Код, який ми використовували у другому розділі, підключає заголовки, ініціалізує та настроює GLFW, створює об’єкт та контекст вікна, ініціалізує GLEW, визначає розмір вікна для відмалювання, створює ігровий цикл, в якому перевіряються дії з вікном мишею або клавіатурою, в якому є функція подвійної буферізації, функції очистки екрану відповідним кольором та остання функція, яка після виходу з циклу очищує виділені ресурси для роботи.

Завдання №1. Подивиться уважно на код и з’ясуйте, як кожна функція працює. За коментуйте ваші припущення у файлі **“FirstWindow.cpp”** поряд з кожною функцією. Перевірте ваші припущення за допомогою довідки у Додатку 1.

3. Лабораторна робота № 5.
Створення найпростіших фігур за допомогою OpenGL

Мета лабораторної роботи: Ознайомитися з середовищем MS VisualStudio, мовою C++, особливостями створення вікна за допомогою OpenGl, створенням таких фігур як точка, відрізок, трикутник та прямокутник. Опрацювати набуті знання щодо порядку створення вікна та фігур; за допомогою середовища VisualStudio створити фігури у вікні.

3.1. Основний теоретичний матеріал

В OpenGL все знаходиться в 3D-просторі, але при цьому екран і вікно - це 2D матриця з пікселів. Тому більша частина роботи OpenGL - це перетворення 3D координат в 2D простір для малювання на екрані. Процес перетворення 3D координат в 2D координати управляється графічним конвеєром OpenGL. Графічний конвеєр можна розділити на 2 великих частини: перша частина перетворює 3D координати в 2D координати, а друга частина перетворює 2D координати в кольорові пікселі [6, 7].

Графічний конвеєр приймає на вході набір 3D координат і перетворює їх на кольорові 2D пікселі на вашому екрані. Графічний конвеєр можна розділити на кілька етапів, де кожен крок вимагає на вхід результат роботи попереднього кроку. Всі ці кроки є досить спеціалізованими (вони мають одну конкретну функцію) і можуть бути легко виконуватися паралельно. Через їх паралельну структуру більшість графічних карт сьогодні мають тисячі малих процесорів для швидкої обробки даних графічного конвеєра за допомогою запуска великої кількості маленьких програм на кожному етапі графічного конвеєра. Ці невеликі програми називаються шейдерами [6, 7].

На рисунку 4.1 можна побачити приблизне представлення всіх етапів графічного конвеєру. Сині частини описують етапи, для яких ми можемо

ввести особисті шейдери. Як видно, графічний конвеєр містить велику кількість розділів, кожен з яких обробляє одну конкретну частину перетворення ваших вершинних даних у повністю відтворений піксель [6, 7].

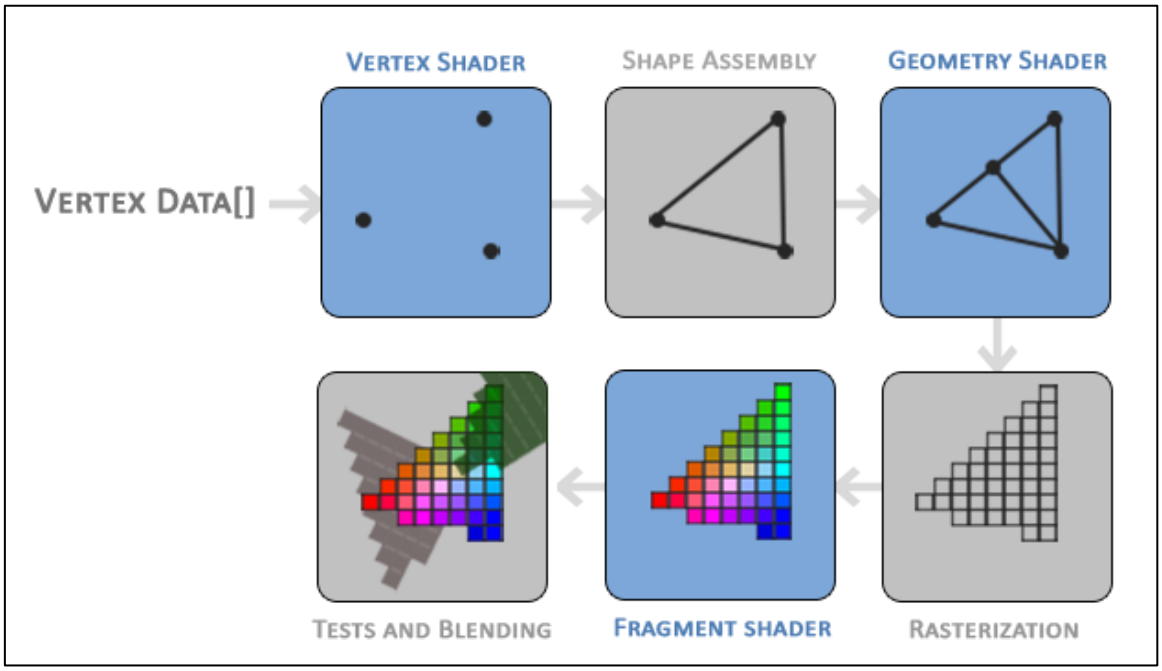


Рисунок 4.1 – Етапи графічного конвеєру

На рисунку 4.2 представлене більш детальне представлення етапів графічного конвеєру, які під час виконання лабораторної роботи ви самі їх послідовно побудуєте. Після створення графічного конвеєру, можна будувати різні геометричні фігури, представляти ці фігури різними кольорами, добавляти текстури та інше. Також ви познайомитесь с шейдерами, етапами їх зборок, буферами зберігання вершинних даних та вершинних атрибутів, та процесом передачі цих даних до графічної карти, також процесом інтерпретації переданих даних.

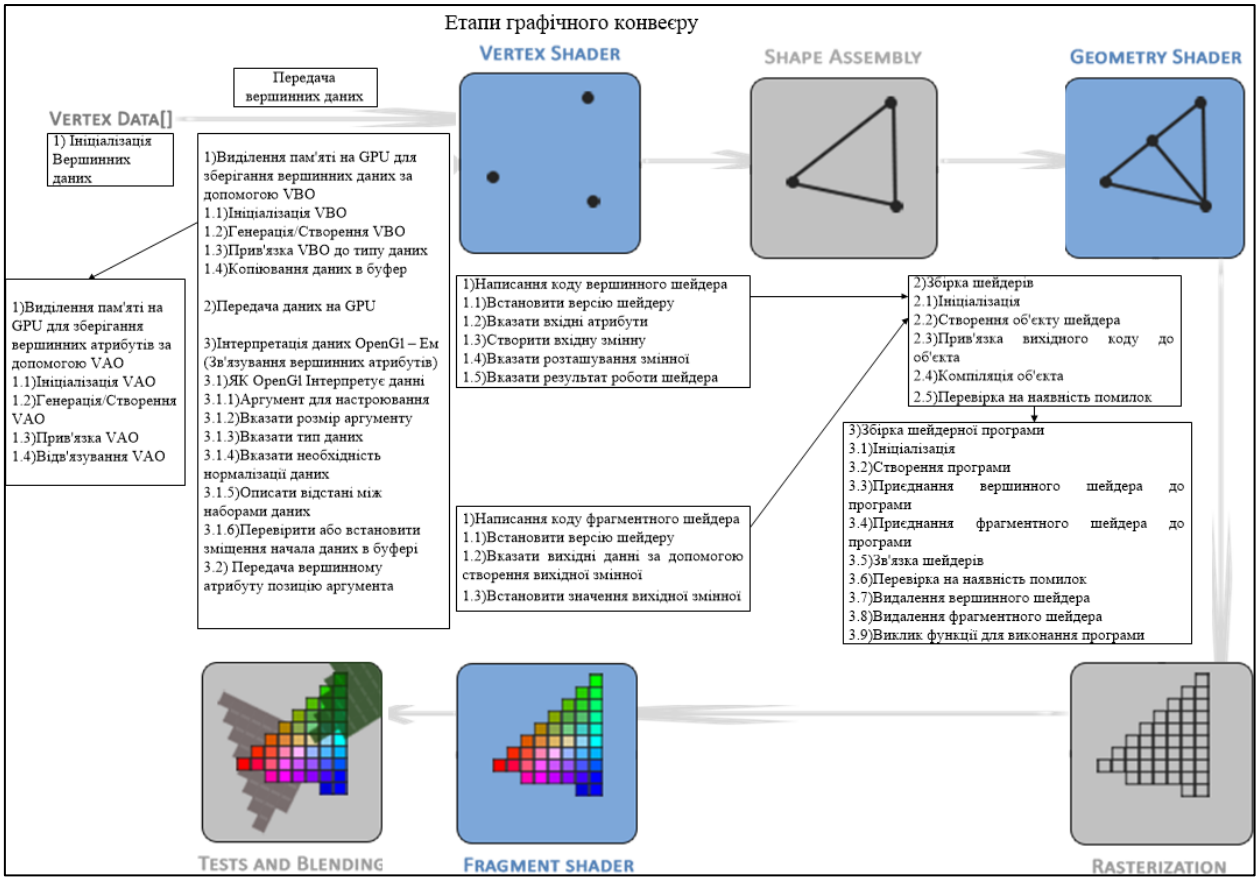


Рисунок 4.2 – Детальне представлення етапів графічного конвеєру

3.2. Завдання до лабораторної роботи

1. Виконати етапи створення графічного конвеєру, описані в розділі 3.5;
2. Відмалювати 2 трикутника, один за одним, за допомогою функції `glDrawArrays();` з використанням більшої кількості вершин. Для першого трикутника використовувати координати вершин з першого етапу. Для другого трикутника додати такі вершини, які не будуть співпадати з вершинами першого трикутника (див. табл. 3.1);
3. Створіть 2 трикутника з використанням 2 різних VBO та VAO;
4. Створить другий фрагментний шейдер, який буде відмальовувати другий трикутник протилежним першому трикутнику кольором.

3.3. Підготовка до лабораторної роботи

- Встановити Visual Studio 2017 (Лабораторна робота №1).
- Зібрати бібліотеки (Розділ 2).
- Ознайомитись з методичними вказівками до виконання лабораторної роботи (Розділ 3.5 – 3.10).
- Вивчити послідовність створення графічного конвеєру.
- Побудувати графічний конвеєр.
- Побудувати трикутник.
- Виконати завдання до лабораторної роботи (Розділ 3.2).
- Відповісти письмово на питання, згідно номеру варіанта.

3.4. Контрольні питання

1. Опишіть процес створення вікна OpenGL згідно Додатку 1.
2. Опишіть загальний процес створення графічного конвеєру
3. Що таке VBO і для чого його використовують?
4. Опишіть процес створення VBO.
5. Що таке шейдери і для чого вони потрібні?
6. Опишіть процес створення фрагментного шейдера.
7. Опишіть процес створення вершинного шейдера.
8. Опишіть процес створення шейдерної програми.
9. Що таке VAO і для чого його використовують?
10. Опишіть процес створення VAO.

3.5. Методичні вказівки до виконання лабораторної роботи №5

3.5.1. Ініціалізація вершинних даних

Для малювання будь чого треба передати OpenGL дані о вершинах. Так як OpenGL – це трьохвимірна бібліотека, то координати вершин фігури, які передаються, повинні знаходитися в тривимірному просторі (x, y, z). OpenGL не займається перетворенням 3D координат в 2D координати, а виконує обробку 3D координат в відповідному проміжку від -1.0 до 1.0 по всім 3 координатам (x, y, z). Ці координати називаються нормалізованими (Лабораторна робота № 1 – 2) [6, 7].

Оскільки малюється один трикутник, треба задати 3 вершини, кожна з яких знаходиться в тривимірному просторі, при цьому координата Z повинна бути рівною 0.0 (для того, щоб трикутник виглядав двомірно). Для цього треба визначити вершини в нормалізованому вигляді в GLfloat масиві [6, 7].

Етап 1: Створення масиву вершинних даних з наступними координатами вершин (табл. 3.1):

Таблиця 3.1 – Вершинні данні для малювання трикутника

Варіант	X1	Y1	X2	Y2	X3	Y3	Z
1	0.71	-0.7	-1.0	-0.8	-0.76	0.66	0.0
2	0.95	0.17	0.64	0.79	-0.63	-0.57	0.0
3	0.32	0.19	0.3	-0.89	0.1	-0.38	0.0
4	-0.72	-0.23	0.89	-0.94	0.52	0.47	0.0
5	-0.85	-0.64	0.67	-0.39	0.65	-0.5	0.0
6	0.8	-0.6	0.44	0.41	0.0	-0.11	0.0
7	0.59	-0.23	-0.5	0.67	-0.47	-0.34	0.0
8	-0.51	-0.62	0.19	0.8	-0.62	-0.19	0.0

9	-0.49	1.0	-0.76	0.95	0.33	0.5	0.0
10	-0.77	-0.5	0.14	-0.36	-0.13	-0.84	0.0

3.5.2. Передача вершинних даних

Після визначення вершинних даних потрібно передати їх в перший етап графічного конвеєра: в верховий шейдер. Це робиться наступним чином: виділяється пам'ять на GPU (розділ 4.2.1), куди зберігаються вершинні дані, вказується OpenGL, як вона повинна інтерпретувати передані їй дані і передається на GPU кількість переданих нами даних. Потім вершинний шейдер (розділ 4.2.2) обробляє таку кількість вершин, яку йому було повідомлено [6, 7].

3.5.3. Виділення пам’яті на GPU

Керування цією пам'яттю здійснюється через, так звані, об'єкти вершинного буфера (vertex buffer objects (VBO)), які можуть зберігати велику кількість вершин в пам'яті GPU. Перевага використання таких об'єктів буферів в тому, що можна послати на відеокарту велику кількість наборів даних за один раз, без необхідності відправляти по одній вершині за раз. Відправлення даних з CPU на GPU досить повільне, тому треба відправляти якомога більше даних за один раз. Але як тільки дані виявляться в GPU, вершинний шейдер отримає їх практично миттєво [6, 7].

Етап 2: Створення VBO

1. Ініціалізація (GLuint);
2. Створення VBO за допомогою функції `glGenBuffers();`
3. Прив’язка VBO до типу об’єктів буферів за допомогою функції `glBindBuffer();`

4. Копіювання вершинних даних в VBO за допомогою функції `glBufferData();`

Примітка: API документація OpenGL (англомовна) знаходиться за наступним посиланням http://docs.gl/ [8]. Там можна знайти список функції OpenGL, їх визначення, призначення, технічні характеристики та параметри. Також список версії OpenGL, в яких та чи інша функція використовується.

Примітка: Під час виконання лабораторної роботи використовується функції третьої та четвертої версії OpenGL (gl3, gl4).

3.6. Шейдери

3.6.1. Вершинний шейдер та його зборка

OpenGL вимагає, щоб були задані вершинний та фрагментний шейдери для малювання. Таким чином треба побудувати два примітивних шейдера для відмалювання заданого трикутника [6, 7].

Вихідний код шейдера пишеться на спеціальній мові GLSL (OpenGL Shading Language), а далі його потрібно зібрати для того, щоб додаток міг з ним працювати [6, 7]. Шейдерам буде приділятися Лабораторна робота №6.

Найпростіший код вершинного шейдера:

```
1 #version 330 core
2 layout (location = 0) in vec3 position;
3 void main()
4 {
5 gl_Position = vec4(position.x, position.y, position.z, 1.0);
6 }
```

Примітка: так як вихідний код написано в GLchar строчці, для спрощення і запобігання помилок, можна скористатися функцією строкових літералів.

```
const GLchar* vertexShaderSource = R"(

#version 330 core
layout (location = 0) in vec3 position;
```

```
void main(void)
{
	gl_Position = vec4(position.x, position.y, position.z, 1.0);
}

)"
```

На рисунку 4.2 описано алгоритм написання коду вершинного шейдера.

Етап 3: Написання коду та збірка вершинного шейдера

1. Написати код вершинного шейдера, який буде зберігатися в С рядку (GLchar);
2. Ініціалізація (GLuint) і створення об’єкту вершинного шейдера за допомогою функції `glCreateShader();`
3. Прив’язка коду шейдера до об’єкта шейдера за допомогою функції `glShaderSource();`
4. Компіляція шейдера за допомогою функції `glCompileShader();`
5. Перевірка на наявність помилок за допомогою функцій `glGetShaderiv()` та `glGetShaderInfoLog();`

```
	GLint success;
	GLchar infolog[512];
	glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);

	if (!success)
	{
		glGetShaderInfoLog(vertexShader, 512, NULL, infolog);
		std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infolog <<
		std::endl;
	}
```

3.6.2. Фрагментний шейдер та його зборка

Фрагментний шейдер займається обчисленням кольорів пікселів. Колір в комп'ютерній графіці представляється як масив з 4 значень: червоний, зелений, синій і прозорість; ця база називається RGBA. Коли ми задаємо колір в OpenGL або в GLSL ми задаємо величину кожного компонента між 0.0 і 1.0. Якщо наприклад ми встановимо величину червоного і зеленого компонентів в

1.0, то ми отримаємо суміш цих кольорів - жовтий. Комбінація з 3 компонентів дає близько 16 мільйонів різних кольорів [6, 7].

Найпростіший код вершинного шейдера:

```
1 #version 330 core
2 out vec4 color;
3 void main()
4 {
5 color = vec4(0.0, 0.0, 0.0, 0.0);
6 }
```

На рисунку 4.2 описано алгоритм написання коду фрагментного шейдера.

Етап 4: Написання коду та збірка фрагментного шейдера

1. Написати код вершинного шейдера, який буде зберігатися в С рядку (GLchar);
2. Ініціалізація (GLuint) і створення об’єкту вершинного шейдера за допомогою функції `glCreateShader();`
3. Прив’язка коду шейдера до об’єкта шейдера за допомогою функції `glShaderSource();`
4. Компіляція шейдера за допомогою функції `glCompileShader();`
5. Перевірка на наявність помилок за допомогою функцій `glGetShaderiv()` та `glGetShaderInfoLog();`

3.6.3. Шейдерна програма

Шейдерна програма - це об'єкт, який є фінальним результатом комбінації кількох шейдерів. Для того щоб використовувати зібрані шейдери їх потрібно з'єднати в об'єкт шейдерної програми, а потім активувати цю програму при відображенні об'єктів, і ця програма буде використовуватися при виклику команди відтворення (в ігровому циклі) [6, 7].

При з'єднанні шейдерів в програму, вихідні значення одного шейдера (фрагментного) зіставляються з вхідними значеннями іншого шейдера (вершинного). Якщо вхідні і вихідні значення не збігаються, то програма сформує признаки помилок [6, 7].

Етап 5: Збірка шейдерної програми

1. Ініціалізація (GLuint) і створення об’єкту шейдерної програми за допомогою функції `glCreateProgram();`
2. Приєднання вершинного шейдера до програми за допомогою функції `glAttachShader();`
3. Приєднання фрагментного шейдера до програми за допомого функції `glAttachShader();`
4. Зв’язування шейдерів за допомогою функції `glLinkProgram();`
5. Перевірка на наявність помилок за допомогою функцій `glGetShaderiv()` та `glGetShaderInfoLog();`
6. Видалення вершинного шейдерів за допомогою функції `glDeleteShader();`
7. Видалення фрагментного шейдерів за допомогою функції `glDeleteShader();`
8. Використання створеної програми в ігровому циклі за допомогою функції `glUseProgram();`

3.7. Інтерпретація даних OpenGL. Зв’язування вершинних атрибутів

Вершинний шейдер дозволяє вказувати будь-які вхідні дані у вигляді атрибутів вершин, і, хоча це і дає велику гнучкість, це означає, що повинно вручну вказувати, яка частина наших вхідних даних йде до вершинного атрибута вершинного шейдера. Для цього потрібно показати, як OpenGL слід інтерпретувати вершинні дані перед рендерінгом [6, 7].

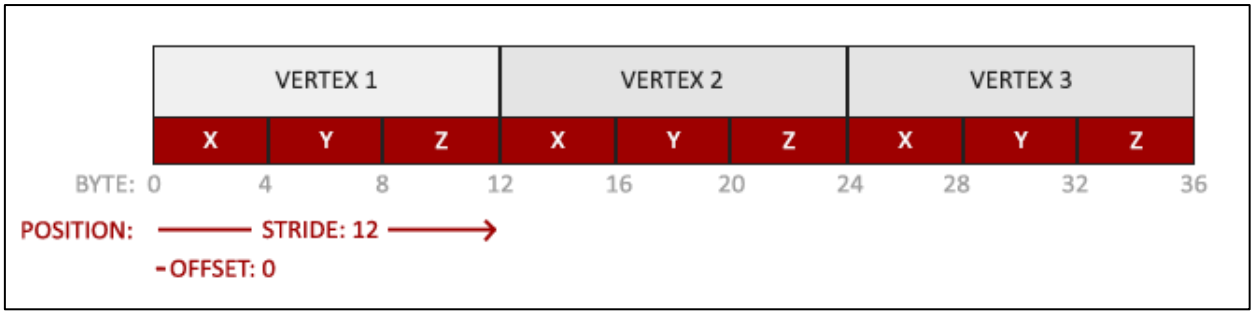


Рисунок 4.3 – Формат вершинного буфера

Формат вершинного буфера має наступні характеристики:

- Інформація зберігається в 32 бітному (4 байти) значенні з плаваючою точкою;
- Кожна позиція формується з трьох значень;
- Не існує ніякого роздільника між наборами з 3 значень. Такий буфер називається щільно упакованим;
- Перше значення в переданих даних це початок буфера.

Виклик функції інтерпретації даних відбувається після функції `glBufferData();`. Інтерпретація даних відбувається за допомогою функції `glVertexAttribPointer();`, і складається с наступних параметрів:

1. Який аргумент шейдера настроюється. В нашому випадку це значення аргументу *position,* позиція котрого була вказана наступним чином: *layout (location = 0);*
2. Розмір аргументу в шейдері. Було використано *vec3,* то вказується 3;
3. Використаний тип даних. Використовуються числа з плаваючою точкою, тому вказується GL_FLOAT;
4. Необхідність нормалізації вхідних даних. Якщо вказується GL_TRUE, то всі дані будуть розташовані між 0 (-1 для знакових значень) і 1. В даному випадку нормалізація не потрібна, тому вказується GL_FALSE;

5. Це шаг, котрий описує відстань між наборами даних. Також можна вказати крок рівний 0 і тоді OpenGL розрахує цей крок (працює тільки з щільно упакованими наборами даних).

6. Останній параметр має тип GLvoid * і тому вимагає такого дивного приведення типів. Це зміщення початку даних в буфері. Цей буфер не має зміщення і тому вказується 0.

Після повідомлення OpenGL, як він повинен інтерпретувати вершинні дані, повинно включити атрибут за допомогою `glEnableVertexAttribArray();`

Цей процес повинно повторювати при будь-якому відображенні об'єкта. Проте його потрібно повторювати навіть тоді, коли відображається, наприклад, 100 об’єктів і 10 вершинних атрибутів. Постійна установка цих параметрів для кожного об'єкта стає дуже проблематичною.

3.8. Об’єкти вершинного масиву (Vertex Array Objects)

Об'єкт вершинного масиву (VAO) може бути також прив'язаний як і VBO і після цього всі наступні виклики вершинних атрибутів будуть зберігатися в VAO. Перевага цього методу в тому, що налаштувати треба тільки один раз, а всі наступні рази буде використана конфігурація VAO. Також такий метод спрощує зміну вершинних даних і конфігурацій атрибутів простим прив'язкою різних VAO [6, 7].

Крім того Core OpenGL потребує використання VAO для того, щоб OpenGL знав, як працювати з нашими вхідними вершинами. Якщо не буде вказано VAO, OpenGL може відмовитися відображати об’єкти [6, 7].

VAO зберігає наступні виклики:

- Виклики `glEnableVertexAttribArray()` та `glDisableVertexAttribArray();`
- Конфігурації атрибутів, котрі виконані через `glVertexAttribPointer();`
- VBO зв’язується з вершинними атрибутами за допомогою виклика `glVertexAttribPointer();`

Етап 6: Створення VAO

1. Ініціалізація (GLuint);
2. Створення VAO за допомогою функції `glGenVertexArrays();`
3. Прив’язка VAO за допомогою функції `glBindVertexArray();`
4. Настроювання та прив’язка необхідного VBO (розділ 4.2.1., етап 2);
5. Інтерпретація вершинних даних (розділ 4.2.5);
6. Відв’язування VBO за допомогою функції `glBindBuffer();`
7. Відв’язування VAO за допомогою функції `glBindVertexArray();`

Примітка: В OpenGL відв’язування це стандартна операція. Бажано відв’язувати все, що було прив’язано, для того, щоб не зіпсувати конфігурацію.

3.9. Відображення трикутника

Останній етап, який нам допоможе відобразити об’єкт, це виклик функції `glDrawArrays();`в ігровому циклі. Ця функція використовує активний шейдер і встановлений VAO для відмалювання вказаних примітивів.

Етап 7: Малювання трикутника

1. Використання активного шейдеру за допомогою функції `glUseProgram();`
2. Прив’язка VAO за допомогою функції `glBindVertexArray();`
3. Виклик функції `glDrawArrays();`для відмалювання примітивів;
4. Відв’язування VAO;

Примітка: Після виконання відображення примітивів (трикутників) доцільно видалити використані буфери, та масиви об’єктів за допомогою функцій `glDeleteVertexArrays(), glDeleteBuffers().`

4. Лабораторна робота № 6.
Створення та робота з шейдерами в OpenGL

Мета лабораторної роботи: Ознайомитися з середовищем MS VisualStudio, мовою C++, особливостями OpenGl Shading Language, роботи з кольором за допомогою даної мови. Опрацювати набуті знання в написанні особистих шейдерів та їх редагуванні; за допомогою середовища VisualStudio створити провести малювання трикутника з минулої лабораторної роботи.

4.1. Основний теоретичний матеріал

В лабораторній роботі №5 було вказано, що шейдери - це маленькі програм, які виконуються на графічному процесорі. Ці програми виконуються для кожного конкретного розділу графічного конвеєра. В основному, шейдери - це не що інше, як програми, що перетворюють вхідні данні на вихідні. Важливо, що шейдери є ізольованими програми, які виконуються паралельно, тому їм не дозволяється спілкуватися один з одним; єдина можливість комунікації, яку вони мають, - це їхні входи та виходи [9, 10].

Для написання шейдерів використовується С-подібна мова - GLSL (OpenGl Shading Language) [11]. GLSL призначений для використання в графіці та містить корисні функції, спеціально орієнтовані на векторні та матричні маніпуляції .

Шейдер завжди починається з опису його версії, за яким слідує перелік вхідних і вихідних змінних, уніформи та їх основна функція. Кожна точка входу шейдера знаходиться в його основній функції, коли обробляється будь-які вхідні змінні та виводиться результати на його вихідні змінні [9, 10].

Шейдер, як правило, має таку структуру:

```
#version version_number

in type in_variable_name;
in type in_variable_name;
out type out_variable_name;
uniform type uniform_name;
void main()
{
    // Процес обчислення, виконання алгоритму, експерименти, і т.п.
    ... // Надається результат роботи шейдера вихідній змінній
    out_variable_name = weird_stuff_we_processed;
}
```

Вхідні змінні вершинного шейдера називаються вершинами атрибутів. Існує максимальна кількість вершин, яку можна передавати в шейдер, таке обмеження накладається обмеженими можливостями технічного забезпечення. OpenGL гарантує можливість передачі принаймні 16 4-х компонентних вершин, інакше кажучи, в шейдер можна передавати як мінімум 64 значення [9, 10].

```
GLint nrAttributes;
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &nrAttributes);
std::cout << "Maximum nr of vertex attributes supported: " <<
nrAttributes << std::endl;
```

4.1.1. Типи даних

GLSL, як і будь-яка інша мова програмування, надає певний перелік типів змінних, до яких належать такі примітивні типи: int, float, double, uint, bool. Також GLSL надає два типи-контейнера: vector і matrix [9, 10].

4.1.2. Vector

Vector в GLSL - це контейнер, що містить від 1 до 4 значень будь-якого примітивного типу. Оголошення контейнера vector може мати такий вигляд (n - це кількість елементів вектору) [9, 10]:

- vecn (наприклад vec4) - це стандартний vector, що містить в собі n значень типу float;
- bvecn (наприклад, bvec4) - це vector, що містить в собі n значень типу boolean;
- ivecn (наприклад, ivec4) - це vector, що містить в собі n значень типу integer;
- uvecn (наприклад, uvecn) - це vector, що містить в собі n значень типу unsigned integer;
- dvecn (наприклад dvecn) - це vector, що містить в собі n значень типу double.

У більшості випадків використовується стандартний vector vecn.

Компоненти вектору можна отримати через vec.x, де x - перший компонент вектору. Можна використовувати .x, .y, .z та .w для доступу до їх першого, другого, третього та четвертого компонентів, відповідно. GLSL також дозволяє використовувати RGBA для кольорів або STPQ для текстурних координат [9, 10].

З вектору, при зверненні до даних через точку, можна отримати не тільки одне значення, а й цілий вектор, використовуючи наступний синтаксис, який називається swizzling (настроювання по адресам) [9, 10]:

```
vec2 someVec;
vec4 differentVec = someVec.xyxx;
vec3 anotherVec = differentVec.zyw;
vec4 otherVec = someVec.xxxx + anotherVec.yxzy;
```

Для створення нового вектору також можна використовувати до 4 літералів одного типу або вектор, правило тільки одне - в сумі потрібно отримати необхідну кількість елементів, наприклад: для створення вектору з 4 елементів ми можемо використовувати два вектору довжиною в 2 елементи, або один вектор довжиною в 3 елементи і один літерал. Також для створення вектор з n елементів допускається вказівка одного значення, в цьому випадку всі елементи вектору візьмуть це значення. Також допускається використання змінних примітивних типів [9, 10].

Вектор дуже гнучкий тип даних і його можна використовувати як для вхідних, так і для вихідних змінних.

```
vec2 vect = vec2(0.5f, 0.7f);
vec4 result = vec4(vect, 0.0f, 0.0f);
vec4 otherResult = vec4(result.xyz, 1.0f);
```

4.1.3. In/Out Змінні

Хоча шейдери - це маленькі програми, але в більшості випадків вони є частиною більшої програми або додатку, тому в GLSL є in і out змінні, що дозволяють створити "інтерфейс" шейдера, який дозволяє отримати дані для обробки і передати результати викликаючій змінній. Таким чином кожен шейдер може визначити для себе вхідні і вихідні змінні використовуючи ключові слова in і out [9, 10].

Вершинний шейдер був би вкрай неефективним, якщо б він не приймав ніяких вхідних даних. Сам по собі цей шейдер відрізняється від інших шейдерів тим, що приймає вхідні значення безпосередньо з вершинних даних. Для того, щоб вказати OpenGL, як організовані аргументи, ми використовуємо метадані розташування, для того, щоб ми могли налаштовувати атрибути на CPU. Ми вже бачили цей прийом раніше: 'layout (location = 0). Вершинний

шейдер, в свою чергу, вимагає додаткових специфікацій для того, щоб ми могли зв'язатися аргументи з вершинних даними [9, 10].

Примітка: Можна опустити layout (location = 0) та використовувати для визначення розташування атрибутів у вашому коді функцію glGetAttribLocation, проте для розуміння того, що робиться під час роботи конвеєра краще його залишити [9, 10].

Фрагментний шейдер (Fragment shader) повинен мати на виході vec4, інакше кажучи фрагментний шейдер повинен надати у вигляді результату колір в форматі RGBA. Якщо цього не буде зроблено, то об'єкт буде підмальовуватися чорним або білим [9, 10].

Таким чином, якщо стоїть задача передачі інформації від одного шейдера до іншого, то необхідно визначити в передавальному шейдера out змінну такого типу як і у in змінної в приймаючому шейдера. Таким чином, якщо типи і імена змінних будуть однакові по обидва боки, то OpenGL з'єднає ці змінні разом, що дасть можливість обміном інформацією між шейдерами (це робиться на етапі компонування шейдерної програми). Для демонстрації цього на практиці треба змінити шейдери з попередньої лабораторної роботи таким чином, щоб вершинний шейдер надавав колір для фрагментного шейдера. Це буде першим етапом лабораторної роботи.

4.1.4. Uniforms

Uniforms (Уніформи) – це спосіб передачі інформації від додатка, який працює на CPU, до шейдеру, який працює на GPU. Уніформи трохи відрізняються від атрибутів вершин. По перше, ***уніформи є глобальними.*** Глобальна змінна для GLSL означає наступне: глобальна змінна буде унікальною для кожної шейдерної програми, і доступ до неї є у кожного шейдера на будь-якому етапі виконання цієї програмі. По друге, ***значення***

уніформи зберігається до тих пір, поки воно не буде скинуто або оновлено [9, 10].

Для оголошення форми в GLSL використовується специфікатор змінної unifrom. Після оголошення форми його можна використовувати в шейдера [9, 10]. Наступним чином встановлюється колір нашого трикутника з використанням форми:

```
        #version 330 core
        out vec4 color;
        uniform vec4 ourColor; // Значення змінної задається в коде OpenGL.
        void main()
{
        color = ourColor;
}
```

Було оголошено змінну уніформи outColor типу вектору з 4 елементів в фрагментному шейдері і вона використовується для установки вихідного значення фрагментного шейдера. Оскільки форма є глобальної змінної, то її оголошення можна робити в будь-якому шейдері. Також це значить, що не потрібно передавати щось з вершинного шейдера у фрагментний. Уніформа не оголошується в вершинному шейдери, тому що вона там не використовуємо [9, 10].

Важливо: Якщо уніформа оголошується і не використовується в шейдерної програмі, то компілятор видалить її з компільованої версії, що може провести до деяких помилок [9, 10].

В поданому коді уніформа не містить в собі корисних даних. Для початку потрібно дізнатися індекс/розташування потрібного атрибуту уніформи в шейдері. Отримавши значення індексу атрибуту можна помістити туди необхідні дані. Це буде другим етапом лабораторної роботи.

4.1.5. Задання кольору через атрибути

Уніформи це дуже зручний спосіб обміну даними між шейдерами та програмою. Проте що робити, якщо є завдання задати колір для кожної вершини? Для цього треба оголосити стільки уніформ, скільки є вершин. Найбільш вдалим рішенням для цієї задачі буде використання атрибутів вершин [9, 10].

Лабораторна робота №5 була присвячена створенню та заповненню VBO, налаштуванню покажчиків на атрибути вершин і зберіганню цієї інформації на VAO. В наступному етапі потрібно буде додати інформацію про кольори на даних вершинах. Для цього треба створити вектор з трьох елементів float. Інформація

4.2. Завдання до лабораторної роботи

1. Замалювати трикутник з лабораторної роботи №5 (етап 1) відповідно етапам лабораторної роботи №6:

1.1. Етап 1: Змінити колір та шейдери згідно етапу: замість бордового використати дані таблиці 4.1;

1.2. Етап 2: Для встановлення кольору використати дані з таблиці 4.1, змінюючи значення одного з елементів вектору (RGB);

1.3. Етап 3: Для кожної вершини додати колір, використовуючи дані таблиці 4.2.

2. Модифікувати вершинний шейдер так, щоб в результаті трикутник перевернувся.

3. Передати горизонтальне зміщення за допомогою уніформи і перемістити трикутник до правого боку вікна за допомогою вершинного шейдера.

4. Передати фрагментного шейдеру позицію вершини і встановити значення кольору, яке рівне значенню позиції (подивіться, як позиція вершини інтерполюється по всьому трикутнику). Чому нижня ліва частина трикутника чорна?

4.3. Підготовка до лабораторної роботи

- Виконати всі завдання та етапи лабораторної роботи №5.
- Ознайомитись з теоретичним матеріалом до лабораторної роботи (Розділ 4.1).
- Виконати три етапи, які приводяться в лабораторній роботі №6.
- Виконати завдання до лабораторної роботи (Розділ 5.2)
- Відповісти письмово на питання, згідно номеру варіанта

4.4. Контрольні питання

1. Дайте визначення шейдеру. Опишіть його функції, цілі та характеристики різним типам шейдерів.
2. Що таке OpenGl Shading Language? Для чого його використовують, опишіть його функцій, цілі та області застосування.
3. Опишіть загальну структуру шейдера.
4. Які основні типи даних використовуються в шейдерах?
5. Опишіть тип – конвеєр: vector. Які типи він в собі має?
6. Опишіть способи використання типу vector з прикладами.
7. Опишіть застосування in/out змінних.
8. Що таке Уніформи? Опишіть застосування уніформ.
9. Опишіть застосування функції glUniform(). Для чого вона потрібна?
10. Опишіть процес задання кольорів вершин через атрибути вершин.

4.5. Методичні вказівки до виконання лабораторної роботи № 6

4.5.1. Використання In/Out Змінних

Задачею цього етапу є передача інформації від одного шейдера до іншого за допомогою In/Out змінних. Для цього треба змінити шейдери з попередньої лабораторної роботи таким чином, щоб вершинний шейдер надавав колір для фрагментного шейдера.

В наступному коді буде оголошено вихідний вектор з 4 елементів з ім'ям vertexColor в вершинному шейдері і ідентичний вектор з назвою vertexColor, але тільки як вхідний у фрагментного шейдера. В результаті вихідний vertexColor з вершинного шейдера і вхідний vertexColor з фрагментного шейдера були з'єднані. Буде встановлено значення vertexColor в вершинному шейдері, відповідне непрозорого бордовому (темно-червоному кольору), і застосування шейдера до об'єкта робить його бордового кольору.

Етап 1: Передавання інформації від одного шейдера до іншого

1. Змінити код вершинного шейдера наступним чином:

```
#version 330 core
layout (location = 0) in vec3 position; // Встановлення позиції атрибута
out vec4 vertexColor; // Передача кольору до фрагментного шейдеру
void main()
{
gl_Position = vec4(position, 1.0); // Безпосередня передача vec3 в vec4
vertexColor = vec4(0.5f, 0.0f, 0.0f, 1.0f); // Встановлення значення вихідної змінної в темно-червоний колір.
}
```

Таблиця 4.1. – Дані для замальовування трикутника

	R	G	B	Елемент вектору для заміни (етап 2)
1	0.47	0.91	0.71	R
2	0.35	0.01	0.80	G
3	0.84	0.98	0.76	B
4	1.00	0.43	0.60	G
5	0.75	0.61	0.74	B
6	0.85	0.49	0.55	R
7	0.36	0.91	0.99	G
8	0.78	0.91	0.71	B
9	0.59	0.53	0.86	G
10	0.22	0.77	0.17	R

2. Змінити код фрагментного шейдера наступним чином:

```
#version 330 core
in vec4 vertexColor;
// Вхідна змінна з вершинного шейдера (та же назва і той же тип)
out vec4 color;
void main()
{
color = vertexColor;
}
```

Примітка: так як вихідний код написано в GLchar строці, для спрощення і запобігання помилок при побудові, можна скористатися C++ функцією строкових літералів.

```
const GLchar* vertexShaderSource = R"(
#version 330 core
void main(void)
{
	gl_Position = vec4(position.x, position.y, position.z, 1.0);
}
)"
```

4.5.2. Використання уніформ для відображення ефектів

Етап 2: Застосування уніформ.

1. Замінити вихідний код фрагментного шейдера з застосуванням уніформ:

```
#version 330 core
out vec4 color;
uniform vec4 ourColor; // Встановлення значення цієї змінної в коді OpenGL.
void main()
{
color = ourColor;
}
```

2. Додати в ігровий цикл наступний код, який буде змінювати колір трикутника з плином часу:

```
glUseProgram(shaderProgram);
GLfloat timeValue = glfwGetTime();
GLfloat greenValue = (sin(timeValue) / 2) + 0.5;
GLint vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ...
//підмальовування трикутника (лабораторна робота №5)
```

В таблиці 4.1 знаходяться інформація про елемент (RGB) вектору, який потрібно замінити, щоб він змінювався з плином часу.

Примітка: в поданому коді ми застосовували наступні функції:

`glfwGetTime()` – отримання часу роботи в секундах;

`sin` – за допомогою функції зміна значення кольору від 0.0 до 1.0;

`glGetUniformLocation()` – запит на індекс (розташування) уніформи, тобто виконується пошук уніформи;

`glUniform4f()` – встановлення значення уніформи (значення кольору);

`glUseProgram()` – доцільно викликати цю функцію ДО пошуку уніформи, щоб уніформа була прив’язана до шейдерної програми. Для запобігання помилок виклик цієї функції треба провести до виконання всіх операцій.

Примітка 2: Оскільки OpenGL в своїй основі є бібліотекою C, вона не має власної підтримки для перевантаження типів, тому там, де можна викликати функцію з різними типами, OpenGL визначає нові функції для кожного потрібного типу; glUniform є прекрасним прикладом цього. Функція вимагає певного постфіксу для типу уніформи, яка встановлюється. Нижче приведені деякі з можливих постфікс:

- f: функція очікує, що значення буде типу float;
- i: функція очікує, що значення буде типу int;
- ui: функція очікує, що значення буде типу unsigned int;
- 3f: функція очікує, що значення буде з 3 типів float;
- fv: функція очікує, що значення буде з float вектором або масивом.

Кожного разу, коли ви хочете налаштувати параметр OpenGL, просто виберіть перевантажену функцію, яка відповідає вашому типу. У нашому випадку ми хочемо встановити 4 float значень окремо, тому ми передаємо наші дані через glUniform4f (зверніть увагу, що ми також могли б використати версію fv).

4.5.3. Використання атрибутів для відображення

Використання поданого методу доцільно, коли треба додати до вершин відповідний колір. Для цього треба створити вектор з трьох елементів float. Для кожної вершини буде свій колір: червоний, зелений та синій.

Етап 3: Додавання кольорів до вершин.

1. Додати інформацію в масив вершин, щодо кольорів кожної вершини (табл. 4.2):

Таблиця 4.2 – Значення кольору для кожної вершини

	Нижній правий кут			Нижній лівий кут			Верхній кут		
	R	G	B	R	G	B	R	G	B
1	0.28	0.15	0.60	0.93	0.85	0.62	0.39	0.44	0.90
2	0.04	0.30	0.20	0.31	0.57	0.34	0.99	0.08	0.62
3	0.50	0.82	0.73	0.36	0.28	0.56	0.18	0.84	0.50
4	0.88	0.18	0.78	0.63	0.49	0.77	0.33	0.39	0.73
5	0.46	0.38	0.46	0.72	0.36	0.41	0.21	0.33	0.47
6	0.93	0.0	0.10	0.63	0.70	0.50	0.51	0.48	0.56
7	0.93	0.01	0.51	0.93	0.16	0.21	0.35	0.16	0.85
8	0.58	0.50	0.51	0.84	0.76	0.03	0.22	0.98	0.81
9	0.83	0.41	0.92	0.68	0.08	0.63	0.93	0.54	0.95
10	0.30	0.96	0.42	0.58	0.36	0.81	0.03	0.56	0.40

```
GLfloat vertices[] = {
// Позиції            // Кольори
0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 0.0f, // Нижній правий кут
-0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f, // Нижній лівий кут
0.0f, 0.5f, 0.0f, 0.0f, 0.0f, 1.0f // Верхній кут
};
```

2. Редагування вершинного шейдеру таким чином, щоб він отримував і вершини, і кольори:

```
#version 330 core
layout (location = 0) in vec3 position; // Встановлення позиції змінної с координатами в 0
layout (location = 1) in vec3 color; // Встановлення позиції змінної с кольором в 1
out vec3 ourColor; // Передача кольору до фрагментного шейдеру
void main()
{
gl_Position = vec4(position, 1.0);
ourColor = color; // Встановлення значення кольору, який отримали від вершинних даних
}
```

3. Редагування фрагментного шейдеру:

```
#version 330 core
in vec3 ourColor;
out vec4 color;
void main()

{
color = vec4(ourColor, 1.0f);
}
```

4. Настроювання вказівників на атрибути вершин:

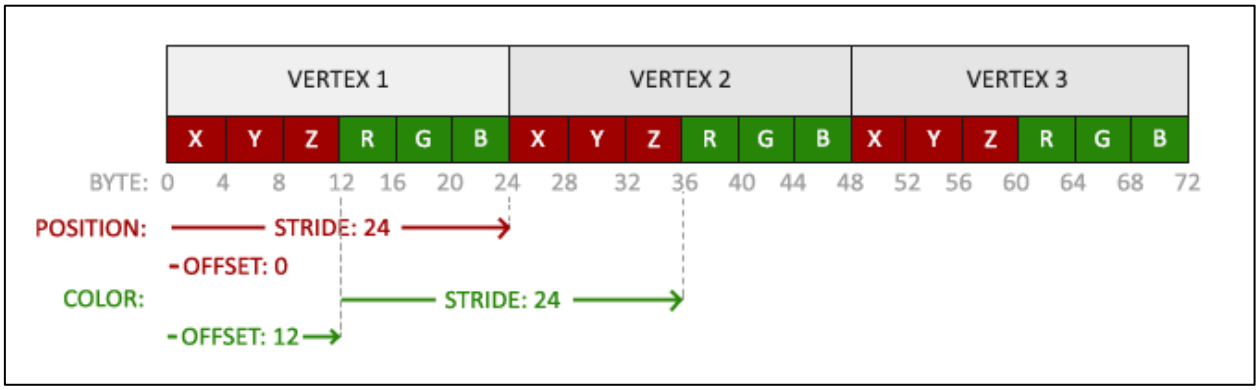


Рис. 6.1. – Зміщення та шаг після додавання кольору

За наступною схемою (рис. 6.1) настроюємо атрибути вершин:

```
// Атрибут с координатами
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(0);
// Атрибут с кольором
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)(3* sizeof(GLfloat)));
glEnableVertexAttribArray(1);
```

5. Поясніть, яким чином проходить зміщення та шаг в даному прикладі (додати пояснення в коментарях до коду)

4.6. Звіт з виконання лабораторної роботи

Звіт повинен містити наступні пункти:

- Вступну частину
- Лістинг програми
- Копію екрану з результатами виконання індивідуального завдання (три копії екрану до завдань 2-4)
- Висновки
- Письмову відповідь на питання для самоконтролю відповідно до номеру студента в групі

5. Лабораторна робота № 7.

Текстурування примітивів в OpenGL

Мета лабораторної роботи: Ознайомитися з базовими прийомами 2D текстурування примітивів в OpenGl. Опрацювати набуті знання в написанні С++ програми текстурування в середовищі VisualStudio, провести текстурування трикутника з лабораторної роботи №5.

5.1. Основний теоретичний матеріал

5.1.1. Загальні відомості.

Текстурування або **накладання текстури** (texture mapping) - це метод зафарбування, за допомогою якого на поверхню примітива (об'єкту) накладається деяке зображення, яке визначається як **текстура** (зображення текстури). Застосування текстур широко використовується для досягнення наступних цілей:

- текстури можна використовувати для того, щоб показати матеріал, з якого зроблений об'єкт. Наприклад, на звичайний паралелепіпед наноситься зерниста картина зрізу дерева і він стає дерев'яним брусом;
- на поверхню може бути нанесена картина зафарбованих повторюваних прямокутників і поверхня перетворюється в стіну, складену з цегли;
- за допомогою текстурування можна наочно уявити фізичні властивості об'єктів в додатках наукової візуалізації. Наприклад, дані про температуру кодуються кольором і наносяться на об'єкт, дозволяючи бачити, як геометрія впливає на перебіг процесів теплопередачі.
- текстури дають можливість моделювати світлові ефекти, наприклад відображення, при створенні фотореалістичних зображень.

Загальну інформацію щодо текстурування можна знайти в [12, 13].

Текстура (зображення текстури) являє собою масив елементів - текселів, дуже схожих на звичайні пікселі, які і мають сенс пікселя текстури.

Цей масив може бути одно-, дво- або, в деяких системах, тривимірним. Виділяють два основних види накладання текстур:

1. MIP-текстурування - метод нанесення текстури, при якому застосовуються численні копії одного зображення текстури, але з різним рівнем деталізації.

2. Рельєфне текстурування – метод, що використовує спеціальні карти висот, нормалей, тощо, що дозволяє надати поверхні 3D-моделі об'єкта насиченості і реалістичності.

В роботі будемо розглядати реалізацію MIP текстурування в середовищі OpenGL. Процес накладання текстури проілюстровано рисунком 7.1. До зображення текстури, за замовчуванням, прив’язана система координат 0UV, в який **прямокутна** текстура довільного розміру (у **текселах)** має нормалізовані розміри, з координатами *U, V,* що змінюються в межах

$$0.0\mathbf{f} \le U \le 1.0\mathbf{f} \ , \quad 0.0\mathbf{f} \le V \le 1.0\mathbf{f} \ .$$

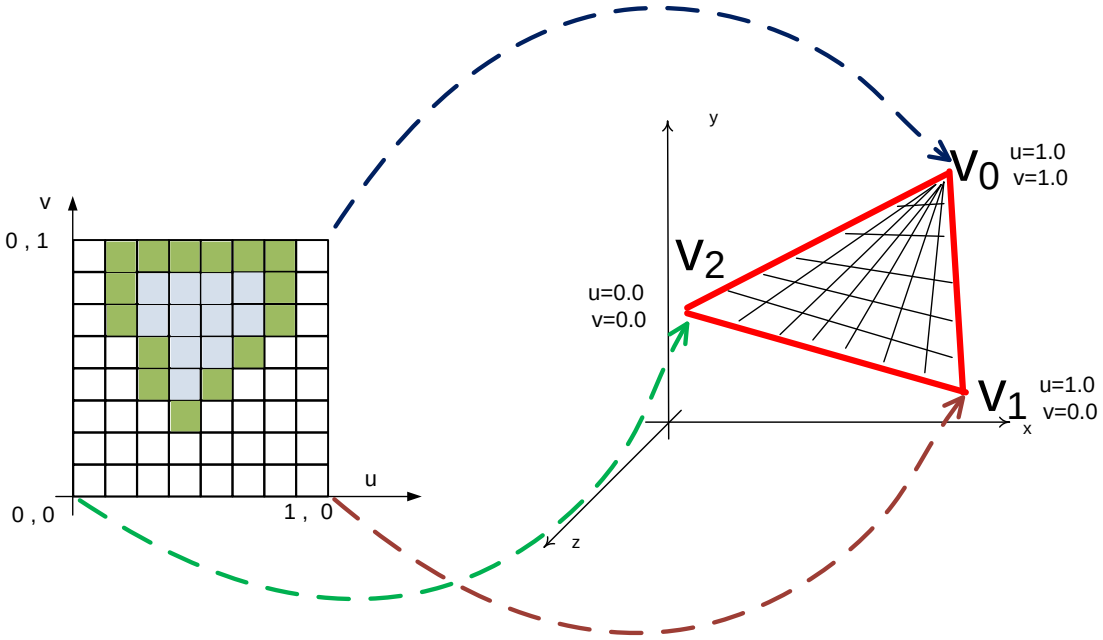


Рис. 7.1. Процес накладання текстури.

В середовищі OpenGL вважається, що *UV*-перетворення в межах одного трикутника є афінним, тому достатньо задати текстурні координати *U* і *V* для кожної вершини кожного з трикутника зображення, що генерується. Для цього, кожній вершині трикутника додатково до атрибутів позиції та кольору додаються значення текстурних координат *U* і *V*, які пов'язують вершину трикутника з відповідним текселем текстури. Відзначимо, що зв'язування цілком виконується розробником (3D-моделером) і визначаються творчими завданнями проекту.

Врахуємо, що текселі в текстурі вибираються (семплуються) за допомогою текстурних координат під час виконання операцій малювання. Ці координати коливаються від 0.0f до 1.0f, де $U = 0.0f$, $V = 0.0f$ є умовно нижній лівий кут і $U = 1.0f$, $V = 1.0f$ - верхній правий кут текстурного зображення. Операція, що використовує ці текстурні координати для отримання кольорової інформації з текселів, називається **вибіркою**. OpenGL пропонує багато варіантів контролю за тим, як ця вибірка буде зроблена.

Також, процес накладання текстури знаходиться в протиріччі з точністю представлення об'єкту в залежності від віддаленості від спостерігача. При віддалені текстура повинна ставати меншою. Коректне представлення текстур в просторовій перспективі забезпечується в OpenGL декількома прийомами. Для представлення текстур на різній відстані від спостерігача і для трикутників різного розміру використовується серія текстурних карт різного розміру, що і називається MIP-мепінгом. В процесі рендерінгу об'єкта використовується та текстурна карта із MIP-мепінга, яка найбільш відповідає розмірам об'єкту і відстанню його до спостерігача. При цьому виконується деяка фільтрація вибраних даних. Білінійна фільтрація передбачає вибір текстурної карти із MIP-мепінга і обчислення зваженої суми найближчих сусідніх текселів для отримання пікселя, який буде виведений на дисплей. Трилінійна фільтрація має більшу обчислювальну складність, оскільки передбачає обробку двох найближчих текстурних карт із MIP-мепінгу. Над

кожною із пари текстурних карт виконується білінійна фільтрація, а із отриманої пари значень береться зважена сума, яка і є результатом.

5.1.2. Послідовність процесу текстурування в OpenGL

У OpenGL текстурування виконується за наступною схемою:

- Підготовка текстури.
- В оперативній пам'яті розміщується масив даних, в якому буде зберігатися образ майбутньої текстури. Вибирається зображення (наприклад, з файлу) і заповнюється масив даними цього зображення.
- Створюється текстура в пам'яті (**glTexImage2D( )**)
- Задається можливі параметри фільтрації текстури для досягнення найкращого сприйняття об'єктів при різних коефіцієнтах масштабування (**glTexParameter ()**).
- Вказується яким чином координати текстури пов'язані з координатами трикутника, що текстурується.

Підготовка текстури. Взагалі зображення текстури може бути згенеровано в будь-якому графічному редакторі (наприклад Microsoft Painter) з використанням довільного графічного зображення. В роботі рекомендуємо використовувати графічні файли в форматі BMP з використанням 24-бітого RGB кольору, з розміром (**обов'язково !**) зображення кратним цілій степені двійки (наприклад 128 X 64, 128 X 256, 512 X 512 та подібне).

Розміщення та загрузка масиву даних текстури. Для завантаження зображення текстури можливо використання стандартних C++ бібліотек, однак для більш докладного розуміння форматів графічних файлів використаємо ручне завантаження. Роз'яснення формату BMP та послідовність завантаження наведені в додатку 1 до лабораторної роботи.

Створення текстури. Послідовність створення текстури в OpenGL подібно створенню буферу вершин. Спочатку генерується ідентифікатор текстури:

```
GLuint textureID;
glGenTextures(1, &textureID);
```

який зв'язується з обробником текстур

```
glBindTexture (GL_TEXTURE_2D, textureID);
```

Параметр GL_TEXTURE_2D вказує, що використовуються 2D текстури. Таким чином усі наступні текстурні операції будуть виконуватися з пов'язаною текстурою.

Далі масив текселей оголошується зображенням текстури шляхом звернення до функції **glTexImage2D ()** , яка має наступну специфікацію:

```
void glTexImage2D (
GLenum target, GLint level, Glint internalformat,
GLsizei width, GLsizei height, Glint border, GLenum format,
GLenum type, const GLvoid * data);
```

Параметри виклику функції:

- ***Target*** - визначає тип текстури, повинен бути G_TEXTURE_2D,
- ***Level*** - задає кількість рівнів деталізації текстури, 0 для базового рівня,
- ***Internalformat*** - визначає кількість кольорових компонент текстури, для RGB повинен бути GL_RGB,
- ***Width*** - задає ширину зображення текстури,
- ***Height*** - задає висоту зображення текстури,
- ***Border*** - визначає ширину межи текстури, повинен бути 0 або 1),
- ***format*** - визначає формат даних тексела, приймаємо GL_RGB,
- ***type*** - *визначає тип даних тексела текстури, повинен бути GL_UNSGNED_BYTE,*

- ***data*** – задає вказівник (*pointer*) на масив даних зображення текстури.

Наприклад:

```
glTexImage2D (GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
GL_RGB, GL_UNSIGNED_BYTE, RGBdata);
```

Тут RGBdata – вказівник на завантажений з BMP файлу масив кольорів текселів (пікселів текстури).

Після виклику **glTexImage2D ()** поточна прив'язана текстура матиме прив'язане до неї зображення. При цьому текстура матиме тільки базове зображення і якщо бажано використовувати MIP-меппінг необхідно додаткво викликати **glGenerateMipmap ( )** після генерації текстури. Ця функція автоматично згенерує всі необхідні MIP-мапи для поточної прив'язаною текстури.

Надалі, якщо це необхідно, встановлюються параметри використання і фільтрації текстури. З кожним зображенням текстури пов'язані параметри текстури, які задають спосіб відображення текселей на пікселі. Вони визначаються за допомогою функції **glTexParameterf**( ) або **glTexParameteri**( ), які мають наступну специфікацію:

```
Void glTexParameterf(GLenum  target, GLenum  pname,
GLfloat  param);
```

Параметри виклику:

- ***target*** – задає тип цільової текстури, для лабораторної роботи повинно бути GL_TEXTURE_2D.
- ***pname*** – вказує символічне ім’я текстурного параметру. Невичерпний список параметрів: GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, GL_TEXTURE_MIN_FILTER, GL_TEXTURE_MAG_FILTER.
- ***param*** – задає значення ***pname***.

Група параметрів TEXTURE_WRAP визначає, що робити, якщо якась із координат текстури (s=*U*, t=*V*) виходить за діапазон [0.0f, 1.0f]. Для кожної з

координат можна задати свій спосіб дії. Можливі два варіанти - CLAMP або REPEAT. У режимі CLAMP (pname = **GL_CLAMP_TO_EDGE, GL_CLAMP_TO_BORDER**), якщо відповідна координата текстури виходить за одиничний діапазон, текстура не накладаються. Цей режим застосовується, коли потрібно накласти зображення на об'єкт в єдиному екземплярі. У другому з можливих режимів – REPEAT (pname = **GL_REPEAT, GL_MIRRORED_REPEAT**), використовується завжди тільки дрібна частина координати і тим самим може бути створена картина, що повторюється.

Друга група параметрів (…**FILTER**) пов'язана з тією обставиною, що розміри пікселів і текселей не обов'язково збігаються. В процесі накладення текстури, виходячи з значень текстурних координат в вершинах полігону, шляхом інтерполяції обчислюються значення текстурних координат в центрі кожного пікселя полігону, отриманого в результаті растерізации, а також визначається область текстурного простору, на яку відображається даний піксель. Залежно від розміру області використовується різні алгоритми фільтрації.

Якщо область більше, ніж один тексел, текстура стискається так, щоб відповідати пікселя екрану, для чого застосовується фільтр стиснення. В іншому випадку, коли область менше тексела, текстура розширюється і використовується фільтр розширення. Докладний опис дивись [14].

Зв’язування текстури із трикутником. Найпростіший спосіб зв'язування вершинних координат з текстурними полягає в **прямому** завданні текстурних координат для кожної вершини, що буде заноситися в VBO. Наприклад:

```
GLfloat vertices[] = {
// Position         // Color          // U   V
-0.7f,-0.7f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f,//Bottom
-0.7f, 0.7f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,// Top
 0.7f, 0.7f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // Top
 0.7f,-0.7f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f, //Bottom
```

```
};
```

Також для зв’язування можна використовувати OpenGL функцію **void glTexCoord (type coord)** різних модифікацій [14].

5.2. Завдання до лабораторної роботи

Базове завдання.

1. Створити довільну текстуру у форматі BMP файлу розміром 512 X 512 текселів та використати її для текстурування трикутнику відповідного варіанту із лабораторної роботи № 5.

Додаткові завдання.

2. За допомогою двох суміжних трикутників завдання лабораторної роботи № 5 згенерувати зображення текстурованого **прямокутника**.
3. Встановити текстурні координати трикутника більш 1.0f та використати функцію **glTexParameterf()** для завдання способу розповсюдження текстури.

5.3. Підготовка до лабораторної роботи

- Ознайомитись з теоретичним матеріалом до лабораторної роботи (Розділ 5.1).
- Виконати базове задання лабораторної роботи **№ 5**.
- Модифікувати програму для реалізації базового завдання лабораторної роботи № 7 (розділ 5.2, завдання 1).
- Відповісти письмово на питання, згідно номеру варіанта

5.4. Контрольні питання

1. Дайте визначення текстури.
2. Що таке текстурні координати? Для чого їх використовують?
3. Опишіть загальну послідовність текстурування.

4. Які розміри текстурного зображення рекомендовані для використання в OpenGL?
5. Надайте призначення і визначить зміст параметрів функції **glTexImage2D ().**
6. Надайте призначення і визначить зміст параметрів функції **glTexParameter**( ) .

5.5. Методичні вказівки до виконання лабораторної роботи № 7

5.5.1. Використання атрибутів при текстуруванні

Передача текстурних даних до вершинного шейдеру потребує завдання додаткового вказівника на текстурні координати в VBO. Вказівник необхідно сформувати відповідно до рис. 7.2.

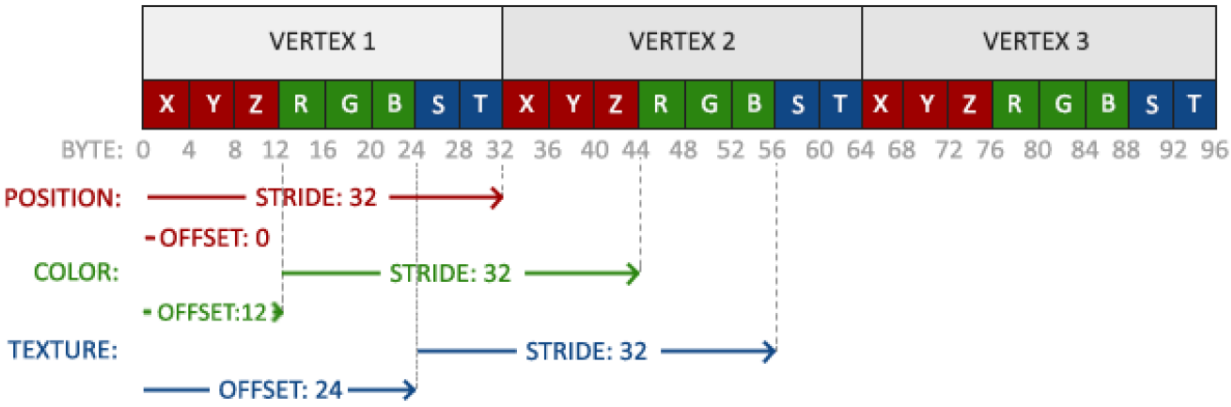


Рис. 7.2. Зміщення та шаг після додавання текстурних координат

Фрагмент коду налаштування текстурного вказівника наведено нижче.

```
// Texture attribute
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
8 * sizeof(GLfloat), (GLvoid*)(6 * sizeof(GLfloat)));
glEnableVertexAttribArray(2);
```

5.5.2. Модифікація вершинного шейдеру

В процесі текстурування графічних примітивів вершинний шейдер не виконую ніяких перетворень з текстурними *UV* координатами, а тільки передає

їх до фрагментного шейдеру. В зв'язку з цим до вершинного шейдеру додаються опис вхідної змінної (з урахуванням вказівника на текстурні координати):

```
layout (location = 2) in vec2 texCord;
```

та опис вихідної змінної:

```
out vec2 ourTexCoord;
```

До основної функції шейдеру додається оператор передачі текстурних координат вершини до фрагментного шейдеру

```
ourTexCoord = texCord;
```

Текст вершинного шейдеру наведено в додатку 2.

5.5.3. Модифікація фрагментного шейдеру

Фрагментний шейдер отримує текстурні *UV* координати від вершинного шейдеру через вхідну змінну

```
in vec2 ourTexCoord;
```

Крім того фрагментний шейдер повинен «знати» який семплер (**sempler**) необхідно використовувати для доступу до текстури. Для цього використовується **uniform** змінна

```
uniform sampler2D textureSampler;
```

Вихідний колір пікселю обчислюється за допомогою вбудованої в OpenGL функції **texture ( )**, яка викликається в **main( )** шейдеру наступним чином

```
fragColor = texture(textureSampler, ourTexCoord);
```

Функція **texture ( )** в якості першого аргументу приймає текстурний **sampler**, а в якості другого аргументу текстурні координати. Функція **texture ( )** під час виконання семплує значення кольору, використовуючи текстурні параметри, які шейдер получив через вхідну змінну ourTexCoord. Результатом роботи фрагментного шейдера буде колір текстури на інтерпольованих текстурних координатах.

Текст фрагментного шейдеру наведено в додатку 2.

5.5.4. Прив’язка текстури до семплеру.

Передача текстури до семплеру фрагментного шейдеру виконується за допомогою виклику

```
glBindTexture(GL_TEXTURE_2D, textureID);
```

який виконується перед викликом **glDrawElements (…)** в основній програмі.

Фрагмент коду наведено в додатку 2.

5.6. Звіт з виконання лабораторної роботи

Звіт повинен містити наступні пункти:

- Вступну частину
- Лістинг програми
- Копію екрану з результатами виконання базового індивідуального завдання
- Копію екрану з результатами виконання додаткових завдань (якщо виконано).
- Висновки .
- Письмову відповідь на питання для самоконтролю.

5.7. Додатки до лабораторної роботи №7

Додаток 1. Формат BMP файлу та шаблон коду для зчитування зображення текстури.

При виконанні лабораторної роботи для зображень текстури рекомендуємо використовувати BMP формат зберігання растрових зображень від компанії Microsoft. Повний офіційний опис формату наведено в [ ].

Взагалі BMP формат зберігає одношарові растри з різною глибиною кольору: 1, 2, 4, 8, 16, 24, 32, 48 и 64 біта на колір, але для лабораторної роботи з текстурування з урахуванням представлення кольорових компонент в OpenGL будемо використовувати зображення з 8-бітовими кольоровими

компонентами RGB , тобто **24 біта** на кольори пікселя, **без використання** таблиці кольорів. Також необхідно урахувати, що для текстурування бажано використовувати зображення у яких ширина та висота зображення кратна цілої ступені двійки, наприклад 256 X 256 пікселів, 256 X 512 пікселів, 512 X 1024 пікселів та подібне.

Спрощена структура BMP файлу (версія 3) наведена на рис. 7.3. BMP файл якнайменш складається з трьох блоків:

- Заголовок (хедер) структури BITMAPFILEHEADER.
- Інформаційний блок BITMAPINFO.
- Піксельні данні RGBARRAY.

Заголовок **BITMAPFILEHEADER** займає перші 14 байт файлу включає наступні поля:

- **bfType** – відмітка типу файлу, завжди два символи **BM** (Hex 42, Hex 4D),
- **bfSize** – розмір файлу в байтах,
- **bfResrved1, bfResrved2** – зарезервовані 4 байту, повинні містити нуль,
- **bfOffBits** - положення (зміщення) масиву піксельних даних (RGBARRAY) щодо початку файлу.

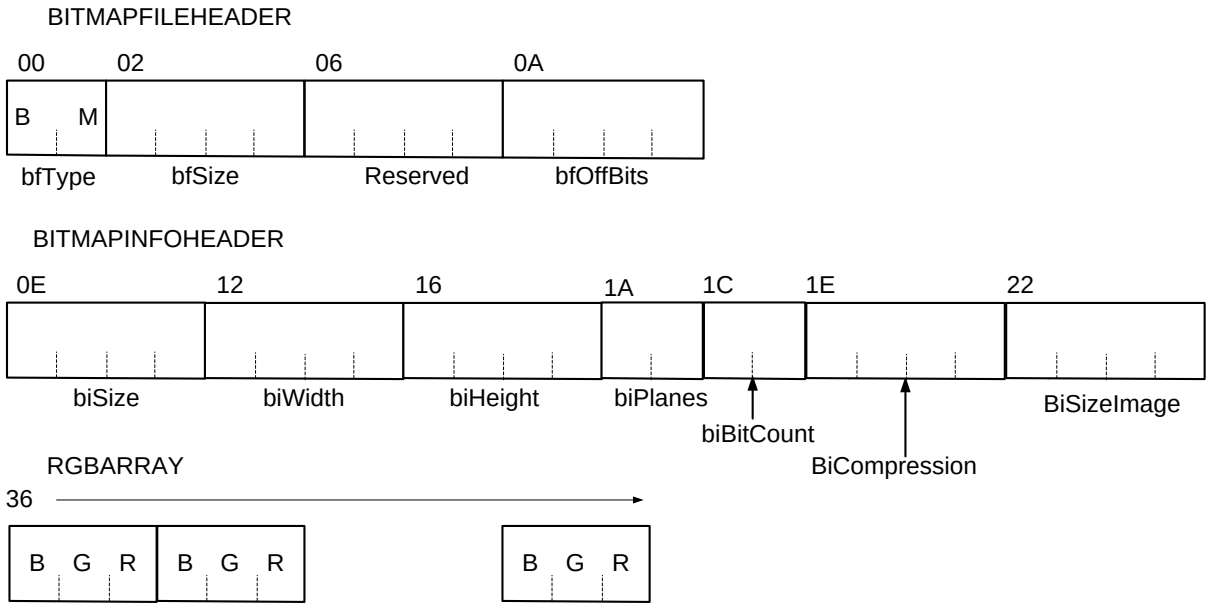


Рис. 7.3. Спрощена структура BMP файлу.

Інформаційний блок **BITMAPINFO** займає наступні 40 байт файлу та містить ширину, висоту і бітність растра, а також формат пікселів, інформацію про кольорову таблицю та інше. Він має наступні поля:

- **biSize** – розмір блоку в байтах сумісно з версією (розглядаємо версію 3),
- **biWidth** – ширина растру в пікселях,
- **biHeight** – висота растру в пікселях,
- **biPlanes** – кількість шарів растру (завжди **1**),
- **biBitCount** – кількість біт на піксель ,
- **biCompression** – вказує на спосіб зберігання пікселів,
- **biSizeImage** – розмір піксельних даних (**RGBARRAY**) в байтах.

Двомірний масив піксельних даних RGBARRAY зберігає інформацію щодо кольорових компонент кожного пікселя. Зображення зберігається однопіксельними горизонтальними смужками (рядками, scans) та записується по-порядку, починаючи з самого нижнього (Bottom-up bitmap) рядку. У кожному горизонтальному ряду пікселі записуються строго від лівого пікселя зображення до правого. При 24-бітому кольорі кожен кольоровий компонент займає 1 байт, колір пікселя - 3 байти. Важливо вказати, що компонент Blue розміщується в молодших бітах / першому байті, Green (зелений) в другому байті, а Red (червоний) старшому (третьому) байті. Тобто у пам’яті кольорові компоненти йдуть в порядку: синій, зелений, червоний (**BGR**) !

З урахуванням формату **BGR** файлу та вказаних обмеженнях , рекомендуємо використовувати наступний шаблон коду для зчитування текстур.

```
// Завдання шляху до BMP  файлу
const char* FilePath = " …  path to BMP texture file";
// Опис хедеру BMP файлу
```

```
unsigned char header[54];    // хедер
unsigned int dataPos;        // зміщення
unsigned int width, height;  // ширина та висота зображення
unsigned int imageSize;      // розмір зображення в байтах
unsigned char * RGBdata;     // масив RGB даних зображення

FILE* BMP_File;
fopen_s(&BMP_File, FilePath, "r"); // відкриття BMP файлу
if (!BMP_File) {// перевірка помилки відкриття BMP файлу
        std::cout << "ERROR::1\n" << infoLog << std::endl;
        return 1;
    }
if (fread(header, 1, 54, BMP_File) != 54) {//перевірка
розміру хедера
        std::cout << "ERROR::2\n" << infoLog << std::endl;
        return 2;
    }
if (header[0] != 'B' || header[1] != 'M') {//перевірка типу
файлу
        std::cout << "ERROR::3\n" << infoLog << std::endl;
        return 3;
    }
// Зчитування позиції BGR даних (зміщення)
dataPos = *(int*)&(header[0x0A]); //RGB date Offset in file
//Зчитування ширини текстури в пікселях
width = *(int*)&(header[0x12]); //Image Width
//Зчитування висоти текстури в пікселях
height = *(int*)&(header[0x16]); // Image Height

// Додаткові перевірки
//Some BMP-files may have ZERRO in imageSize AND(OR)
dataPos
if (dataPos == 0) dataPos = 54; // set offset Image DATA
imageSize = width * height * 3;
```

```
// Створення буферу в пам’яті для BGR  даних
RGBdata = new unsigned char[imageSize];
// Зчитування BGR даних до буферу
fread(RGBdata, 1, imageSize, BMP_File);
// Закриття файлу
fclose(BMP_File);
```

Додаток 2. Фрагменти коду для виконання лабораторної роботи

```
// Шейдер вершин
const GLchar* vertexShaderSource = "#version 330 core\n"
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;
layout (location = 2) in vec2 texCord;
out vec3 ourColor;
out vec2 ourTexCoord;
void main()
{
gl_Position = vec4(position, 1.0);
ourColor = color;
ourTexCoord = texCord;
};

// ШЕЙДЕР фрагментів
const GLchar* fragmentShaderSource = "#version 330 core"
uniform sampler2D textureSampler;
in vec3 ourColor;
in vec2 ourTexCoord;
out vec4 fragColor;
void main()\
{
fragColor = texture(textureSampler, ourTexCoord);
};
```

```
// Фрагмент коду основної програми
glBindTexture(GL_TEXTURE_2D, textureID);
glUseProgram(shaderProgram);
glBindVertexArray(myVAO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, myEBO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, (void*)0
```

6. Лабораторна робота № 8. Побудова геометрії 3D сцен. Базові 3D перетворення в OpenGL

Мета лабораторної роботи: Ознайомитися з базовими прийомами 3D перетворень в процесі синтезу зображення 3D сцени за допомогою OpenGl. Опрацювати набуті знання в написанні C++ програми побудови відображення геометрії 3D сцени для різних параметрів наглядача.

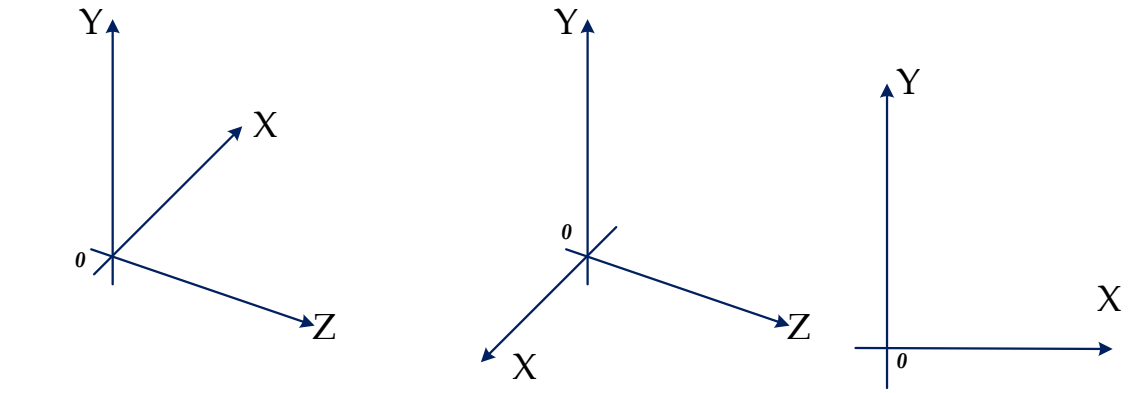
6.1. Основний теоретичний матеріал

6.1.1. Загальні відомості.

Системи координат. В OpenGl за замовчуванням використовується три типи систем координат (рис. 8.1.) [16]:

- **Тривимірна правостороння система *OXYZ*** (рис. 8.1.а), в якій із кінця осі ***Z*** поворот від осі ***X*** до осі ***Y*** бачиться проти годинної стрілки. Таким чином визначаються світова система координат (ССЛ) та локальна/об'єктна система координат (ЛСК / ОСК). Передбачається, що вісь ***Z*** спрямована на спостерігача. В світовій системі (ССК) представлені координати центрів всіх об'єктів сцени та їх орієнтація. С другого боку локальна система координат (ЛСК) жорстко зв'язана з «тілом» деякого об'єкту. В ЛСК визначаються всі геометричні параметри об'єкту.

- **Тривимірна лівостороння система** (рис. 8.1.б), в якій із кінця осі ***Z*** поворот від осі ***X*** до осі ***Y*** бачиться вслід годинної стрілки. Це система координат спостерігача - видова система координат (ВСК), яка повзано з положенням та орієнтацією спостерігача в «світовому» просторі. Передбачається, що вісь ***Z*** спрямована в глиб екрану.
- **Двовимірна система** (рис. 8.1.в) – система координат області виведення віконна система, в який розташовано вікно екрану приладу візуального виведення.



а) Правостороння система координат (об'єктна/локальна та світова системи)

б) Лівостороння система координат – видова система координат (перетворення проектування)

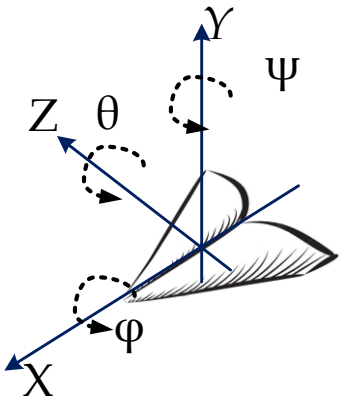
в) Система координат області виведення (вікно екрану)

Рис. 8.1. Системи координат, що використовуються OpenGl

Для представлення орієнтації «тіла» в просторовій системі координат використовується аналог кутів Ейлера, які визначаються як (рис. 8.2.)

- Кут нутації **α** (або **θ**) – задає обертання довкола осі X.
- Кут власного обертання ***β*** (або **ψ**) – задає обертання довкола осі Y.
- Кут прецесії ***γ*** (або **φ**) – задає обертання довкола осі Z.

В залежності від виду об'єкта і розв'язуваної задачі назви кутів орієнтації і порядок їх "використання" для розрахунку просторової орієнтацій об’єкту можуть бути різними. Так, наприклад, зазвичай для літальних апаратів (рис. 8.2) використовуються наступні кути:



- крен $\boldsymbol{\varphi}$ – задає кут обертання довкола осі **X,**
- тангаж $\boldsymbol{\theta}$ – задає кут обертання довкола осі **Z,**
- рискання $\boldsymbol{\psi}$ – задає кут обертання довкола осі **Y**.

Рисунок 8.2. Кути просторової орієнтації.

Більш детально визначення кутів Ейлера надано в [17].

Задача синтезу відображення 3D сцени в найпростішому випадку можна представити наступним чином (рис. 8.3.).

За звичай в сцені присутні декілька об’єктів для відображення. В найпростішому випадку, який розглядається, присутній один об’єкт $\mathbf{O}$, геометрія якого задана в локальній системі координат (ЛСК). Тобто, в ЛСК задано геометричний опис об’єкту списками вершин $\boldsymbol{v}_i, i = 0,1,\dots,n$, ребр та граней. Будемо вважати, що положення та орієнтація об’єкту $\mathbf{O}$ в світовий системі координат $\boldsymbol{OXYZ}$ задається вектором положення $[x_O \quad y_O \quad z_O]^T$ об’єкту та трьом кутами $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$ послідовного повороту об’єкту навколо осей $\boldsymbol{X_O, Y_O, Z_O}$ ЛСК відповідно.

Додатково можливе завдання одного або декількох точкових та/або просторових джерел освітлення $\boldsymbol{L}$**,** які задаються своїм положенням в світі та, при необхідності, кутами просторової орієнтації.

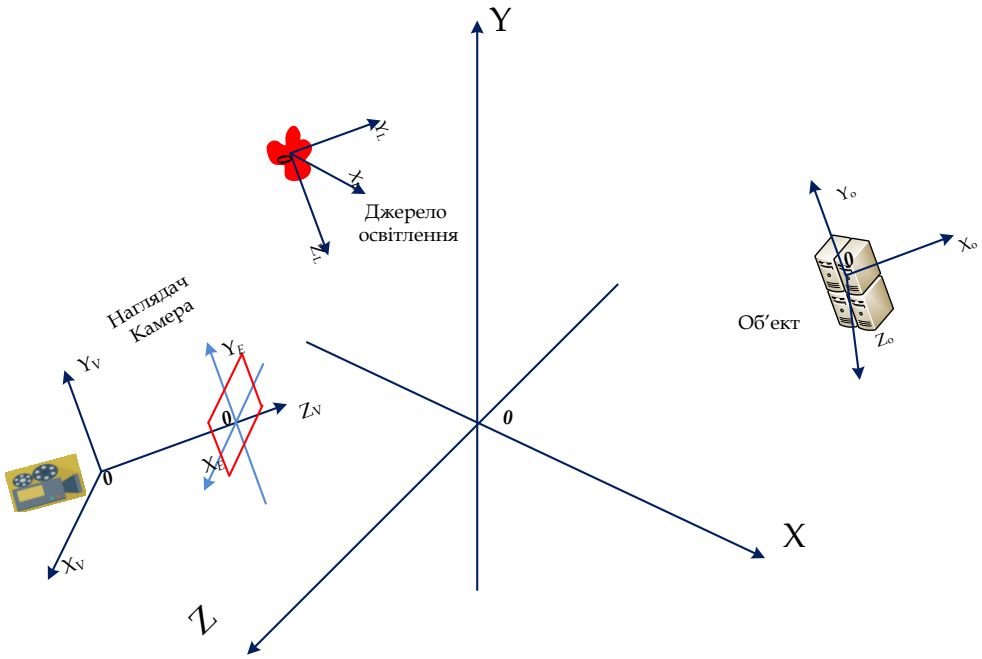


Рис. 8.3. Найпростіша 3D сцена.

В системі присутній спостерігач $\boldsymbol{V}$ (наглядач, камера), з яким пов'язана лівостороння видова система координат (ВСК) $\mathbf{0}\,\boldsymbol{X_v Y_v Z_v}$. Спостерігач розташован в початку ВСК, а напрям його погляду збігається с віссю $\boldsymbol{Z_v}$. Позиція спостерігача у ССК задається вектором його положення $[x_V \quad y_V \quad z_V]^T$ та трьом кутами $\boldsymbol{\alpha_V}$, $\boldsymbol{\beta_V}$, $\boldsymbol{\gamma_V}$ послідовного повороту об'єкту навколо осей $\boldsymbol{X_V}$, $\boldsymbol{Y_V}$, $\boldsymbol{Z_V}$ ВСК відповідно.

Спостерігач наглядає за «подіями» світу через вікно, яке обмежує його поле зору. Вікно задано в системі координат виведення $\mathbf{0}\,\boldsymbol{X}_{\mathrm{E}}\boldsymbol{Y}_{\mathrm{E}}$, початок якої співпадає з точкою $[\mathbf{0}, \mathbf{0}, \boldsymbol{Z}_{\mathrm{E}},]$ ВСК, тобто центр екрану розташовано на відстані $\boldsymbol{Z}_{\mathrm{E}}$ від спостерігача. Вважаємо екран симетричним з розмірами $\pm\frac{WIDTH}{2}, \pm\frac{HEIGHT}{2}$.

Задача синтезу 3D зображення сцени полягає в побудові плоского (2D) зображення, яке бачить спостерігач з його позиції через вікно нагляду.

6.1.2. Матриці перетворення.

Для реалізації різних просторових перетворень в OpenGl використовуються операції в однорідних координатах над векторами та матрицями, тобто:

а) точці з декартовими координатами (x, y, z) відповідає вектор $[hx,\ hy, hz, h]^T, h \neq 0$, де h - множник, типово = 1;

б) матриці просторового перетворення мають розмір 4 X 4,

в) множення деякої матриці $\boldsymbol{M}$ на вектор $\boldsymbol{P}$ виконується в наступному вигляді $\boldsymbol{P}^* = \boldsymbol{M} * \boldsymbol{P}$.

Матриця перетворення масштабування (розтягу, стиску) $\boldsymbol{S}$ з коефіцієнтами $s_x,\ s_y, s_z$ має вигляд

$$\boldsymbol{S} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Операція повороту на кут $\boldsymbol{\gamma}$ навколо осі $\boldsymbol{Z}$ має матрицю перетворення

$$\boldsymbol{R}_z(\boldsymbol{\gamma}) = \begin{bmatrix} \cos(\boldsymbol{\gamma}) & \sin(\boldsymbol{\gamma}) & 0 & 0 \\ -\sin(\boldsymbol{\gamma}) & \cos(\boldsymbol{\gamma}) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Операція повороту на кут $\boldsymbol{\alpha}$ навколо осі $\boldsymbol{X}$ має матрицю перетворення

$$\boldsymbol{R}_x(\boldsymbol{\alpha}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\boldsymbol{\alpha}) & \sin(\boldsymbol{\alpha}) & 0 \\ 0 & -\sin(\boldsymbol{\alpha}) & \cos(\boldsymbol{\alpha}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Операція повороту на кут $\boldsymbol{\beta}$ навколо осі $\boldsymbol{Y}$ має матрицю перетворення

$$\boldsymbol{R}_y(\boldsymbol{\beta}) = \begin{bmatrix} \cos(\boldsymbol{\beta}) & 0 & -\sin(\boldsymbol{\beta}) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\boldsymbol{\beta}) & 0 & \cos(\boldsymbol{\beta}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Матриця перенесення (зсуву) $\boldsymbol{T}$ на відстань t_x, t_y, t_z має вигляд

$$\boldsymbol{T} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_x \\ 0 & 0 & 1 & t_x \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Матриця перспективного одноточкового проектування $\boldsymbol{P}$ на площину екрану, який знаходиться на відстані $\boldsymbol{Z}_{\mathrm{E}}$ від наглядача має вигляд

$$\boldsymbol{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & {}^{1}/_{\boldsymbol{Z}_{\mathrm{E}}} & 0 \end{bmatrix}.$$

6.1.3. Послідовність перетворень при синтезі геометрії сцени

Послідовність геометричних перетворень при синтезі сцени за допомогою OpenGL ілюструється рис. 8.4.

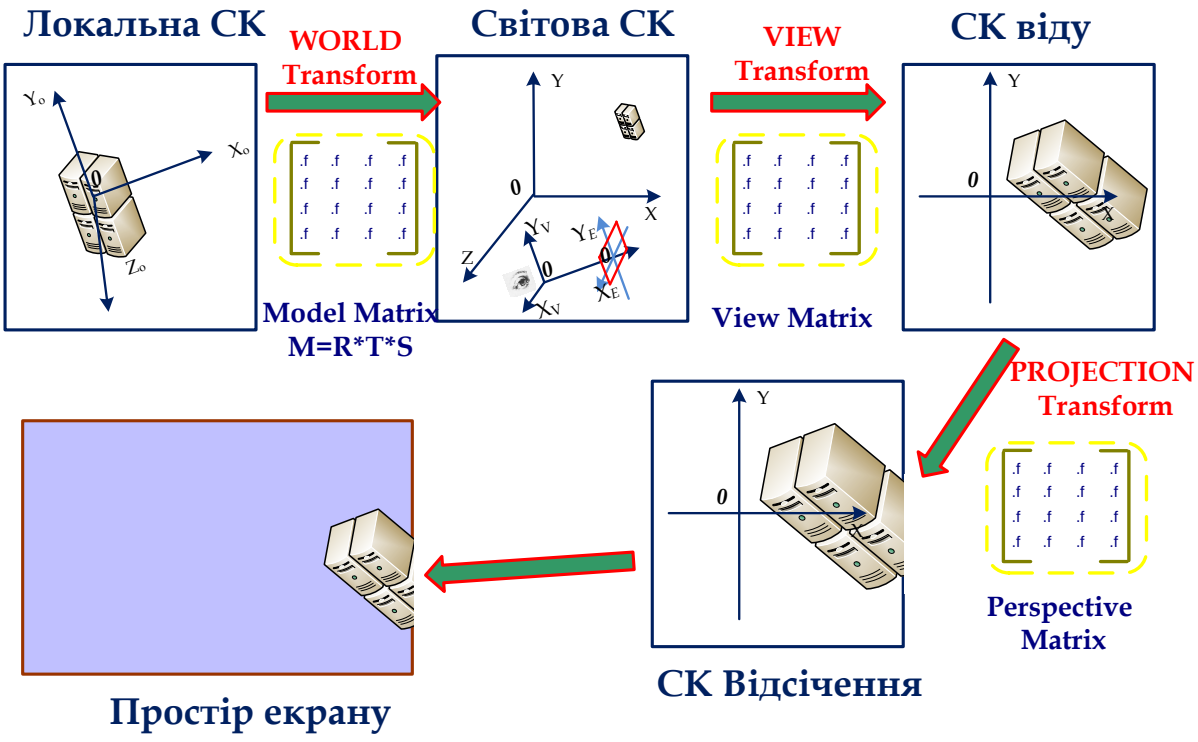


Рис. 8.4. Послідовність OpenGL перетворень

Крок 1. Модельне перетворення - перетворення координат об'єкту **O** із локальної системи координат до світової системи координат. Для цього формується модельна матриця (model matrix) $\boldsymbol{M}$ як результат послідовного множення матриць масштабування $\boldsymbol{S}$, повороту $\boldsymbol{R} = \boldsymbol{R}_z(\boldsymbol{\gamma}) * \boldsymbol{R}_x(\boldsymbol{\alpha}) * \boldsymbol{R}_y(\boldsymbol{\beta})$ навколо координатних осей та зсуву $\boldsymbol{T}$. Тобто

$$\boldsymbol{M} = \boldsymbol{T} * \boldsymbol{R} * \boldsymbol{S}\,.$$

В процесі відображення виконується операція $\boldsymbol{v_i^{CCK}} = \boldsymbol{M} * \boldsymbol{v}_i, \forall\, i = 1,2,\ldots,n$, в результат якої знаходяться координати вершин об'єкту в ССК.

Крок 2. Видове перетворення – перетворення координат об'єкту **O** із ССК координат до видової системи координат (ВСК), тобто до системи координат спостерігача. Для цього створюється матриця видового перетворення (View matrix) $\boldsymbol{M}_V$ з урахуванням положення спостерігача в ССК та його орієнтації в просторі і виконується $\boldsymbol{v_i^{BCK}} = \boldsymbol{M} * \boldsymbol{v_i^{CCK}}, \forall\, i = 1,2,\ldots,n.$

Крок 3. Проектування – перетворення координат об'єкту **O** із ВСК координат до системи координат виведення (екранної системи координат -ЕСК) за допомогою матриці проектування. З використанням перспективного одноточкового проектування це матриця $\boldsymbol{P}$ **.** В процесі відображення виконується операція $\boldsymbol{v_i^{ECK}} = \boldsymbol{P} * \boldsymbol{v_i^{BCK}}, \forall\, i = 1,2,\ldots,n\; i = 1,2,\ldots,n,$ що дає координати проекції об'єкту на площину екрану.

Крок 4. Зазвичай на цьому етапі виконуються афінні перетворення (зсув, масштабування) в ЕСК. З урахуванням спрощеної постановки задачі ці перетворення в роботі, не використовується.

Крок 4. Виконується усунення (відсічення) невидимих ділянок об'єкту відображення. Розгляд методів вирішення цієї достатньо важкої задачі виходить за рамки лабораторної роботи. В OpenGL для цього використовується метод **Z**–буферу (глибинної буферізації) [ ].

6.1.4. Огляд бібліотеки OpenGL Mathematics

Для виконання математичних перетворень при написанні програм в середовище OpenGL рекомендується використовувати бібліотеку OpenGL

Mathematics (GLM), оптимізовану для роботи з 2-, 3- , 4-розмірними векторами (**`glm::vec2, glm::vec3, glm::vec4`**) та матрицями (**`glm::mat2, glm::mat3, glm::mat4`**), відповідно. Бібліотека сумісна практично із усіма компіляторами C++ та CUDA. Офіційний опис бібліотеки наведено в [18].

Надаємо короткий огляд необхідних базових функцій бібліотеки.

♦ Формування матриці масштабування $\boldsymbol{M} := \boldsymbol{S} * \boldsymbol{M}$ – повертає поточну матрицю ***M*** перемножену на матрицю ***S*.**

```
glm::mat4 glm::scale( glm::mat4 const & m, glm::vec3 const & factors);
```

Параметри:

m – поточна матриця,

faktors = [s, sY, sZ] – вектор масштабних коефіцієнтів.

♦ Формування матриці повороту $\boldsymbol{M} := \boldsymbol{R} * \boldsymbol{M}$ **-** повертає поточну матрицю ***M*** перемножену на матрицю ***R*.**

```
glm::mat4 glm::rotate( glm::mat4 const & m, float angle, glm::vec3 const & axis);
```

Параметри:

m – поточна матриця,

angle – кут повороту (в кутових градусах),

axis = [aX, aY, aZ] – вектор, що задає вісь повороту.

♦ Формування матриці зсуву $\boldsymbol{M} := \boldsymbol{T} * \boldsymbol{M}$ **-** повертає поточну матрицю ***M*** перемножену на матрицю ***T*.**

```
glm::mat4 glm::translate( glm::mat4 const & m, glm::vec3 const & translation);
```

Параметри:

m – поточна матриця,

translation = [tX, tY, tZ] – вектор зсуву.

♦ Формування матриці виду ***V*** - повертає матрицю видового перетворення.

```
glm::mat4 glm::lookAt( glm::vec3 const & eye, glm::vec3 const & center, glm::vec3 const & up);
```

Параметри:

eye = [eyeX, eyeY, eyeZ] – вектор позиції камери в ССК,

center = [centerX, centerY, centerZ] - визначають напрям зору спостерігача,

up = [upX, upY, upZ] – вектор "вертикальної" орієнтації камери.

♦ Формування матриці перспективного перетворення ***P*** - повертає матрицю перспективного проектування.

```
glm::mat4 perspective( float fov, float aspect, float zNear, float zFar);
```

Параметри:

fov — визначає поле зору в напрямку осі ***Y*** (в кутових градусах),

aspect –визначає співвідношення сторін екрану,

zNear – визначає дистанцію від спостерігача до «ближчої» площини відсічення по осі ***Z.***

zFar – визначає дистанцію від спостерігача до «дальньої» площини відсічення по осі ***Z.***

Рисунки 8.5 та 8.6 пояснюють визначення параметрів функції **perspective( )**.

Параметри **zNear** та **zFar** обмежують поле зору спостерігача по осі ***Z***. Тобто спостерігач не бачить об’єктів, розташованих ближче ніж **zNear** і далі ніж **zFar**. Поле зору (**fov**) задає кут відносно горизонтальної площини, який обмежує поле зору спостерігача по вертикалі.

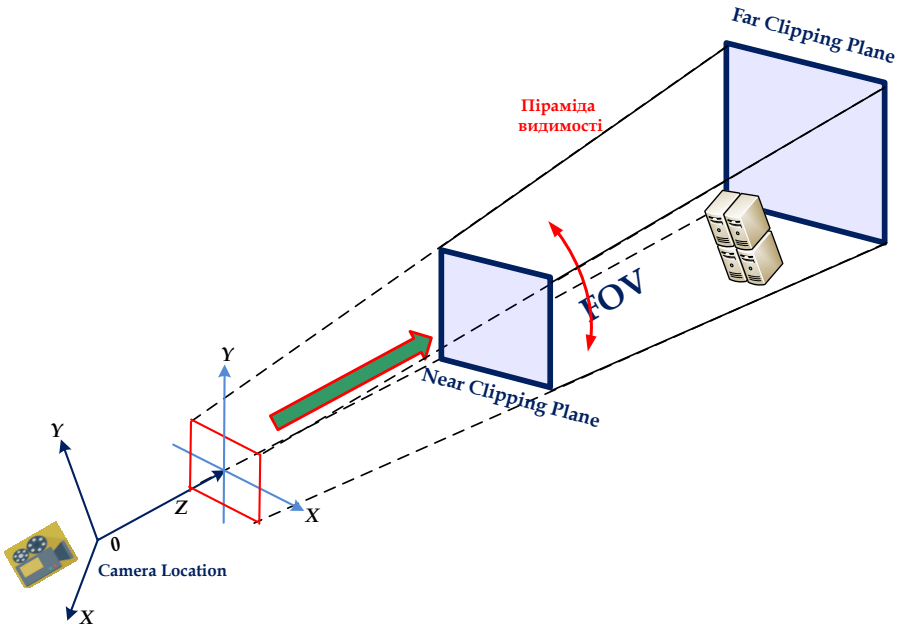


Рис. 8.5. До формування матриці перспективного перетворення

Параметр **aspect** задає кут відносно вертикальної площини, який обмежує поле зору спостерігача по горизонталі. Але цей кут задається як співвідношення розмірів екрану `WIDTH/HEIGHT`.

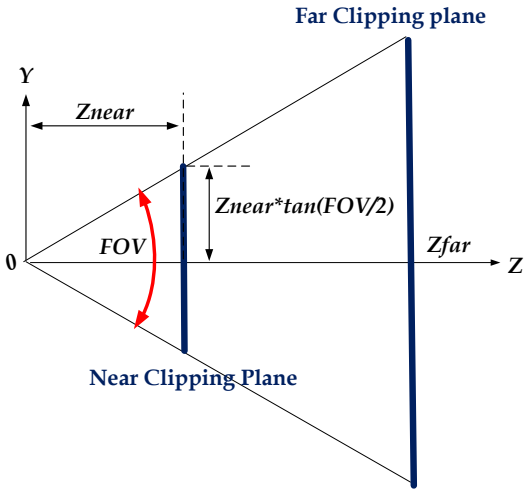


Рис. 8.6. Пояснення формування FOV

Таким чином ці параметри визначають так звану тривимірну **піраміду видимості** – область простору, який «бачить» спостерігач.

♦ Множення двох матриць $\boldsymbol{M1}, \boldsymbol{M2}$ - повертає матрицю $\boldsymbol{M} = \boldsymbol{M1} \times \boldsymbol{M2}$.

glm::mat4 glm::matrixCompMult(matType const & M1, matType const & M2)

Параметри:

M1, M2 — вказівники на матриці.

6.2. Завдання до лабораторної роботи

Базове завдання.

Відобразити сцену (рис. 8.7) відповідно до варіанту, що наведено в таблицях 3.1 (координати вершин об'єкту), 3.2 (кольори вершин об'єкту) та 3.3 (положення спостерігача та параметри піраміди видимості). Всі кути повороту об'єкту і спостерігача відносно ССК **нульові**.

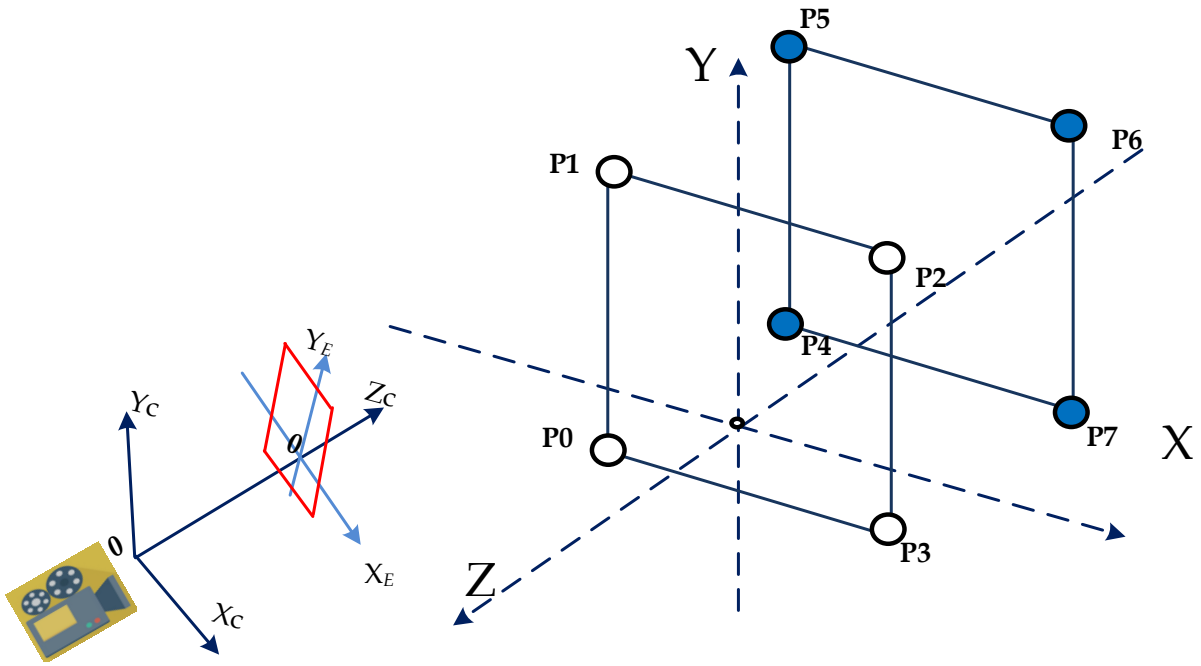


Рис. 8.7. Загальний вигляд 3 сцени для відтворення.

Додаткове завдання 1.

За допомогою операції зсуву змінити положення об'єкту в ССК відповідно до даних, наведених в таблиці 3.4.1

Додаткове завдання 2.

Змінити положення спостерігача, змістивши його положення в ССК відповідно до даних наведених в таблиці 3.4.2.

Додаткове завдання 3.

Змінити орієнтацію об’єкту за допомогою повороту навколо заданої осі на заданий кут відповідно до даних наведених в таблиці 3.4.3.

6.3. Підготовка до лабораторної роботи

- Ознайомитись з теоретичним матеріалом до лабораторної роботи (Розділ 6.1).
- Виконати базове задання лабораторної роботи **№ 8**.
- Модифікувати програму для реалізації додаткового завдання 1.
- Модифікувати програму для реалізації додаткового завдання 2.
- Модифікувати програму для реалізації додаткового завдання 3.

6.4. Контрольні питання

Для чого використовується світова система координат (ССК)?

Для чого використовується локальна система координат (ЛСК)?

Для чого використовується видова система координат (ВСК)?

Для чого використовується та як формується матриця масштабування?

Для чого використовується та як формується матриця повороту навкруг координатної осі?

Для чого використовується та як формується матриця виду?

Для чого використовується та як формується матриця перспективного перетворення?

Що таке ***fov*** ?

Вкажіть послідовність оброки геометрії сцени при візуалізації 3D сцен.

Таблиця 3.1 – Вершинні данні об’єкту

	P0			P1			P2			P3			P4			P5			P6			P7		
	X	**Y**	**Z**	**X**	**Y**	**Z**	**X**	**Y**	**Z**	**X**	**Y**	**Z**	**X**	**Y**	**Z**	**X**	**Y**	**Z**	**X**	**Y**	**Z**	**X**	**Y**	**Z**
1	-1	-1	-1	-1	+1	-1	+1	+1	-1	+1	-1	-1	-1	-1	-2	-1	+1	-2	+1	+1	-2	+1	-1	-2
2	-1	-1	-2	-1	+1	-2	+1	+1	-2	+1	-1	-2	-2	-2	-3	-2	+2	-3	+2	+2	-3	+2	-2	-3
3	-2	-2	-3	-2	+2	-3	+2	+2	-3	+2	-2	-3	-3	-3	-4	-3	+3	-4	+3	+3	-4	+3	-3	-4
4	-2	-2	-1	-2	+2	-1	+2	+2	-1	+2	-2	-1	-2	-2	-2	-2	+2	-2	+2	+2	-2	+2	-2	-2
5	-2	-2	-2	-2	+2	-2	+2	+2	-2	+2	-2	-2	-2	-2	-3	-2	+2	-3	+2	+2	-3	+2	-3	-3
6	-2	+2	-3	+2	+2	-3	+2	-3	-3	+2	-2	-3	-3	-3	-4	-3	+3	-4	+3	+3	-4	+3	-3	-4
7	-3	-3	-4	-3	+3	-4	+3	+3	-4	+3	-3	-4	-1	-1	-5	-1	+1	-5	+1	+1	-5	+1	-1	-5
8	-2	-2	-1	-2	+2	-1	+2	+2	-2	+1	-1	-2	-4	-4	-4	-4	+4	-4	+4	+4	-4	+4	-4	-4
9	-2	-2	-2	-2	+2	-2	+2	+2	-2	+2	-2	-3	-3	-3	-4	-3	+3	-4	+3	+3	-4	+3	-3	-4
10	-2	+2	-3	+2	+2	-3	+2	-3	-2			4	-5	-5	-4	-5	+5	-4	+5	+5	-4	+5	-5	-4

Таблиця 3.2 – Колір вершин об’єкту

	P0			P1			P2			P3			P4			P5			P6			P7		
	R	**G**	**B**	**R**	**G**	**B**	**R**	**G**	**B**	**R**	**G**	**B**	**R**	**G**	**B**	**R**	**G**	**B**	**R**	**G**	**B**	**R**	**G**	**B**
1	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.5	0.5	0.5	0.0	0.5	1.0	0.0	0.5	0.5	0.0	1.0
2	0.0	1.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	1.0	0.5	0.5	0.0	0.5	1.0	0.0	0.5	0.5	0.0	1.0	0.0	0.5
3	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.5	0.5	0.0	0.0	0.0	0.5	0.0	0.0	0.0	0.5	0.0	0.8
4	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	1.0	1.0	0.0	0.8	0.8	0.0	0.0	0.0	0.8	0.0	0.0	0.0	0.8	0.0	0.9
5	0.0	1.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	1.0	0.9	0.9	0.0	0.0	0.0	0.9	0.0	0.0	0.0	0.9	0.0	0.9
6	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.9	0.9	0.9	0.0	0.9	0.0	0.0	0.0	0.9	0.0	1.0
7	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	1.0	1.0	0.0	0.5	0.5	0.5	0.5	0.0	0.0	0.0	0.5	0.0	0.0	0.0	0.5
8	0.0	1.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	1.0	0.8	0.8	0.8	0.8	0.0	0.0	0.0	0.8	0.0	0.0	0.0	0.8
9	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.9	0.0	0.9	0.9	0.0	0.0	0.0	0.9	0.0	0.0	0.0	0.9
10	0.9	0.9	0.0	0.0	0.0	0.9	0.0	0.0	0.0	0.9	0.0	0.9	0.9	0.0	0.0	0.9	0.9	0.9	0.0	0.9	0.0	0.0	0.0	0.9

Таблиця 3.3 – Позиція спостерігача в ССК та параметри піраміди видимості

Орієнтація спостерігача для всіх варіантів кути повороту нульові.

	P0			UP			ZNear	zFar	FOV	WIDTH	HEIGHT
	X	Y	Z	X	Y	Z			градуси	Пиксел	пиксел
1	0.0	0.0	2.0	0.0	-1.0	0.0	1.0	100.0	15.0	800	500
2	0.0	0.0	3.0	0.0	-1.0	0.0	1.5	200.0	20.0	900	500
3	0.0	0.0	1.0	0.0	-1.0	0.0	2.0	100.0	25.0	900	600
4	0.0	0.0	2.0	0.0	-1.0	0.0	1.0	200.0	15.0	1000	750
5	0.0	0.0	3.0	0.0	-1.0	0.0	1.5	100.0	20.0	800	500
6	0.0	0.0	1.0	0.0	-1.0	0.0	2.0	200.0	25.0	900	500
7	0.0	0.0	2.0	0.0	-1.0	0.0	1.0	100.0	15.0	900	600
8	0.0	0.0	3.0	0.0	-1.0	0.0	1.5	200.0	20.0	1000	750
9	0.0	0.0	1.0	0.0	-1.0	0.0	2.0	100.0	25.0	1100	900
10	0.0	0.0	4.0	0.0	-1.0	0.0	1.0	200.0	30.0	1200	1000

Таблиця 3.4 – Додаткові завдання.

3.4.1.Додаткове завдання 1

	Зміщення об’єкту		
	X	**Y**	**Z**
1	1.0	0.0	0.0
2	0.0	1.0	0.0
3	0.0	0.0	1.0
4	-1.0	0.0	0.0
5	0.0	-1.0	0.0
6	0.0	0.0	-1.0
7	-1.0	0.0	1.0
8	1.0	-1.0	0.0
9	1.0	0.0	-1.0
10	1.0	-1.0	-1.0

3.4.2.Додаткове завдання 2

	Зміщення спостерігача		
	X	**Y**	**Z**
1	1.0	0.0	2.0
2	0.0	1.0	3.0
3	-1.0	0.0	4.0
4	0.0	-1.0	5.0
5	-1.0	1.0	2.0
6	-1.0	-1.0	3.0
7	1.0	1.0	4.0
8	1.0	-1.0	5.0
9	1.0	-1.0	3.0
10	1.0	1.0	4.0

3.4.3.Додаткове завдання 3

	Поворот	
	Ось	Кут (°)
1	X	25.0
2	Y	45.0
3	Z	10.0
4	X	-10.0
5	Y	-15.0
6	Z	-20.0
7	X	+60.0
8	Y	+50.0
9	Z	-50.0
10	X	+30.0

6.5. Методичні вказівки до виконання лабораторної роботи № 8

6.5.1. Встановлення пакету

Бібліотеку **GLM** можна скачати за посиланням https://glm.g-truc.net/0.9.9/index.html. Збірка та лінкування **GLM** виконується подібно бібліотеці **GLFW** , як викладено в п.2.2.3.

В програмі потрібно додати команди препроцесору для включення необхідних заголовних файлів **GLM** :

```
// GLM
// Include all GLM core -- GLSL features
#include <glm/glm.hpp> // vec2, vec3, mat4, radians
// Include all GLM extensions
#include <glm/ext.hpp> // perspective, translate, rotate
```

6.5.2. Використання буферу EBO

Буфер EBO зберігає масив індексів (номерів) вершин, яку OpenGL використовує, щоб встановити послідовність обробки вершин. Це називається відмалювання за індексами (**indexed drawing**) (рис. 8. ???))

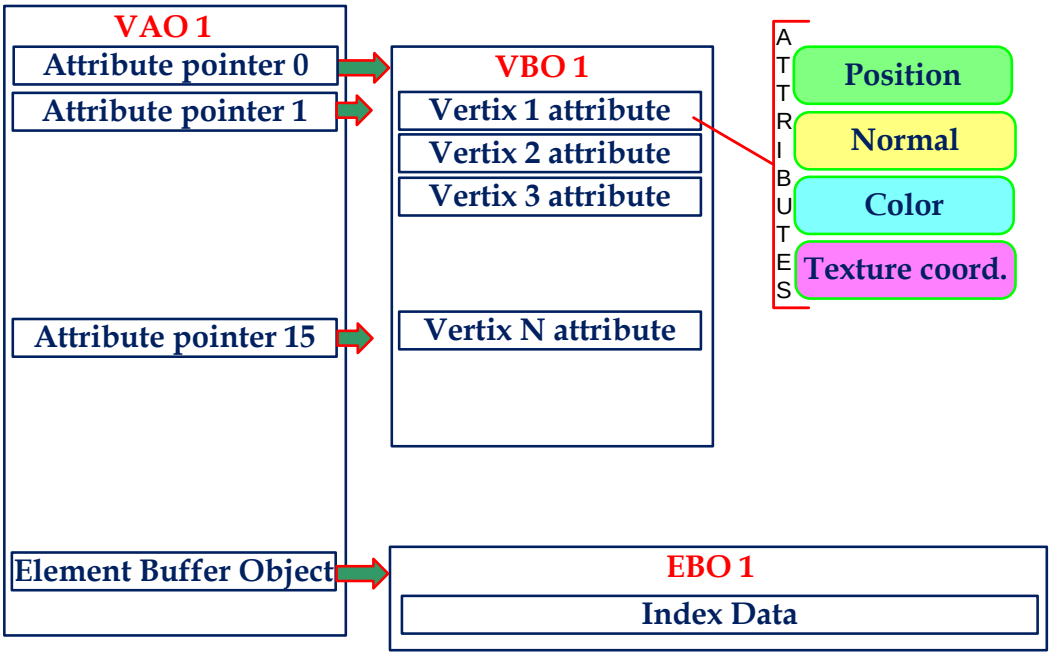


Рис. 7.2. Зміщення та шаг після додавання текстурних координат

Порядок створення EBO подібне створенню и використанню VAO. Наприклад, об’єкт, що відображується, має 8 вершин. Визначається масив індексів

```
GLuint elements[ ] = {// Опис індексів
    0, 1, 2, 3, 0, 4, 5, 6, 7};
```

Далі створюється буфер з ім’ям **ebo:**

```
GLuint ebo;
glGenBuffers(1, &ebo);
```

Створений буфер зв’язується з об’єктом **GL_ELEMENT_ARRAY_BUFFER:**

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
```

І, на останнє, до об’єкту **GL_ELEMENT_ARRAY_BUFFER** записуються масив індексів**:**

```
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(elements), elements,
GL_STATIC_DRAW);
```

При виконанні в ігровому циклі програми команди**:**

```
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT, 0);
```

вершини об’єкту OpenGL буде обробляти в порядку 0, 1, 2, 3, 0, 4, 5, 6.

6.5.3. Обчислення матриць перетворень

Спочатку визначаються матриці перетворень:

```
glm::mat4 transmatr;  // матриця
glm::mat4 model; // матриця
glm::mat4 view; // матриця
glm::mat4 camera; // матриця
glm::mat4 projection; // матриця
```

Формується матриця масштабування $\boldsymbol{S}$ (приклад наведено для $s_x = 1$, $s_y = 1$, $s_z = 1$):

```
model = glm::scale(glm::mat4(1.0f), glm::vec3(1.0f, 1.0f, 1.0f));
```

Далі обчислюється матриця $\boldsymbol{R}$ моделі (приклад наведено для повороту об’єкту навкруг осі X на кут 30 градусів):

```
model = glm::rotate(model, glm::radians(30.0), glm::vec3(1.0f, 0.0f,
0.0f));
```

Знаходиться матриця $\boldsymbol{T}$ моделі (приклад наведено зміщення об'єкту на -1.0 вдовж осі $\boldsymbol{Z}$):

```
view = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, -1.0f));
```

Обчислюється матриця видового перетворення $\boldsymbol{M_v}$ моделі (приклад наведено для позиції наглядача зміщеної вдовж осі $\boldsymbol{Z}$ ССК на 2.0 одиниці, напрям зору вдовж осі $\boldsymbol{Z}$ ВСК та вертикального $\boldsymbol{UP}$ вектору):

```
camera = glm::lookAt(glm::vec3(0.0f, 0.0f, 2.0f), glm::vec3(0.0f,
0.0f, -1.0f), glm::vec3(1.0f, 0.0f, 0.0f));
```

Знаходиться матриця проектування (приклад наведено для $\boldsymbol{fov}$ = 5.0, $\boldsymbol{Znear}$, = 1.0, $\boldsymbol{Zfar}$ = 100.0), помножена на матрицю $\boldsymbol{M_v}$:

```
projection = glm::perspective(5.0f,
glm::float32(WIDTH)/glm::float32(HEIGHT), 1.0f, 100.f)*camera;
```

І, на останнє, обчислюється зведена матриці перетворення як:

```
transmatr = projection * view* model;
```

6.5.4. Передача матриць до вершинного шейдеру

Обчислені матриці передаються до вершинного шейдеру за допомогою **Uniform** змінних. Спочатку ініціюются вказівники на необхідні змінні та матриці у шейдерх:

```
int Displ_Location = glGetUniformLocation(shaderProgram, "displ");
int M_Model_Location = glGetUniformLocation(shaderProgram, "model");
int M_View_Location = glGetUniformLocation(shaderProgram, "view");
int M_View_Projection = glGetUniformLocation(shaderProgram,
    "projection");
```

Далі вказується шейдерна програма, що буде виконуватися, та зв'язуються вказівники на **Uniform** змінні у шейдерній програмі та вказівники на відповідні змінні в основної програмі.

```
glUseProgram(shaderProgram);
glUniform3f(Displ_Location, 0.0f, displ, 0.0f);
glUniformMatrix4fv(M_Model_Location, 1, GL_FALSE,
    glm::value_ptr(model));
```

```
glUniformMatrix4fv(M_View_Location, 1, GL_FALSE,
    glm::value_ptr(view));
glUniformMatrix4fv(M_View_Projection, 1, GL_FALSE,
    glm::value_ptr(projection));
```

6.5.5. Вершинний шейдер

В процесі відображення вершинний шейдер формує остаточну матрицю перетворення як множену матриць та формує координат фрагменту кожен вектор вершини перемножує на цю матрицю та передає їх до графічного конвеєру через вбудовану змінну **gl_Position.** Колір передається до фрагментного шейдеру через вихідну змінну **ourColor**, значення якої дорівнює кольору вершини, що обробляється

Текст вершинного шейдеру:

```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;

uniform mat4 model; // матриця моделі
uniform mat4 view; // матриця виду
uniform mat4 projection; // матриця проекції

out vec3 ourColor; // Для передачі кольору до фрагментного шейдеру

void main()
{
     gl_Position = projection*view*model*vec4(position, 1.0f);
     ourColor = color;
}
```

6.5.6. Фрагментний шейдер

Фрагментний шейдер отримує параметри кольору для кожного фрагменту (пікселю) формуємого зображення від вершинного шейдеру через змінну **ourColor** та передає в графічний конвеєр для подальшої обробки колір пікселю як 4-х розмірний вектор (додається **alfa** компонента = 1.0) за допомогою змінної **FragColor**.

Текст фрагментного шейдеру

```
version 330 core
in vec3 ourColor;
```

```
out vec4 FragColor;

void main()
{
    FragColor = vec4(ourColor, 1.0f);
}
```

6.5.7. Включення тесту Z-буферу

Для урахування відсікання невидимих елементів об'єктів, що відображуються, OpenGL використовує алгоритм **Z-**буферу, який реалізовано на апаратному рівні в відеокарті. Буфер включається наступнимb командами:

```
glEnable(GL_DEPTH_TEST);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

6.6. Звіт з виконання лабораторної роботи

Звіт повинен містити наступні пункти:

- Вступну частину
- Лістинг програми
- Копію екрану з результатами виконання базового індивідуального завдання
- Копію екрану з результатами виконання додаткових завдань (якщо виконано).
- Висновки .
- Письмову відповідь на питання для самоконтролю.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Learn OpenGL [Електронний ресурс] – Режим доступу: https://learnopengl.com/
2. Creating a window [Електронний ресурс] – Режим доступу: https://learnopengl.com/Getting-started/Creating-a-window
3. Learnopengl. Урок 1.2 — Создание окна [Електронний ресурс] – Режим доступу: https://habr.com/post/311198/
4. Hello Window [Електроний ресурс] – Режим доступу: https://learnopengl.com/Getting-started/Hello-Window
5. Learnopengl. Урок 1.3 — Hello Window [Електронний ресурс] – Режим доступу: https://habr.com/post/311234/
6. Hello Triangle [Електроний ресурс] – Режим доступу: https://learnopengl.com/Getting-started/Hello-Triangle
7. Learnopengl. Урок 1.4 — Hello Triangle [Електронний ресурс] – Режим доступу: https://habr.com/post/311808/
8. OpenGL API Documentation [Електронний ресурс] – Режим доступу: http://docs.gl/
9. Shaders [Електроний ресурс] – Режим доступу: https://learnopengl.com/Getting-started/Shaders
10. Learnopengl. Урок 1.5 — Shaders [Електронний ресурс] – Режим доступу: https://habr.com/post/313380/
11. GLSL [Електроний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/GLSL
12. Маценко В.Г. Комп’ютерна графіка: Навчальний посібник. – Чернівці: Рута, 2009 – 343 с. ISBN 966-568-846-4

13. OpenGL super bible comprehensive tutorial and reference / Richard S. Wright Jr. ...[et al.]. — 5th ed., Pearson Education, Inc., 2011.- 969 p., ISBN 978-0-321-71261-5

14. OpenGL ES Software Development Kit [Електронний ресурс] – Режим доступу: https://www.khronos.org/registry/OpenGL-Refpages/es3.0/

15. Опис формате BMP [Електроний ресурс] – Режим доступу: https://docs.microsoft.com/en-us/windows/desktop/gdi/bitmap-storage

16. Modern OpenGL Guide test [Електронний ресурс] – Режим доступу: https://open.gl/transformations

17. Кути повороту [Електронний ресурс] – Режим доступу: **https://api-**2d3d-cad.com/euler_angles_quaternions/

18. OpenGL Mathematics [Електронний ресурс] – Режим доступу: https://glm.g-truc.net/0.9.9/index.html

19. Z-буферизація [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/Z-буферізація

Додаток. Довідка щодо функцій для створення вікна

Функція, змінна або ідентифікатор	Опис функції, змінної або ідентифікатора	Аргументи
`#define GLEW_STATIC` `#define GLEW_DINAMIC`	**Статична лінковка** означає, що бібліотека буде інтегрована з виконуваним файлом під час компіляції. Перевага такого підходу в тому, що не має потреби стежити за додатковими файлами, крім виконуваного файлу. Недоліки такого підходу полягають у тому, що виконуваний файл збільшується в розмірах і при оновленні бібліотеки/проекту змушено збирати знову виконуваний файл. **Динамічна лінковка** здійснюється за допомогою .dll і .so файлів, здійснюючи поділ коду бібліотеки і коду додатка, зменшуючи розмір файлу і спрощуючи оновлення бібліотеки. Недоліком такого підходу є той факт, що вам доведеться випускати DLL файли разом з фінальним додатком. Файли в процесі виконання лабораторних робіт не такі великі, що треба застосовувати зменшення розміру, тому для роботи використовуйте **Статичну** версію GLEW	

```
#include <GL/glew.h>;
#include <glfw3.h>;
```

Підключення заголовків. **Переконайтеся в тому, що підключення GLEW відбувається раніше GLFW.** Заголовки GLEW містять в собі підключення всіх необхідних заголовків файлів OpenGL, таких як GL / gl.h

```
glfwInit();
```

Ініціалізація GLFW

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
```

Оскільки буде використовуватися OpenGL версії 3.3, то необхідно повідомити GLFW, що використовується саме ця версія, що відбувається в результаті виклику методу `glfwWindowHint();`

Якщо версія OpenGL, встановлена на ПК, не відповідає вказаній в аргументі, або вона нижче, то GLFW не буде запущено

Першим аргументом необхідно передати ідентифікатор параметра, який піддається зміні, а другим параметром передається значення, яке встановлюється відповідно параметру. Ідентифікатори параметрів, а також деякі їх значення знаходяться в загальному перерахуванні з префіксом GLFW_

```
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

Функція призводить до помилки у разі використання застарілих версій OpenGL.

Примітка: для успішного виконання лабораторних робіт переконайтесь, що на вашому ПК встановлена версія OpenGL 3.3 та вище. Для цього можна використати програму **OpenGL Extensions Viewer** (https://download.cnet.com/OpenGL-Extensions-Viewer/.html). Якщо версія нижче, переконайтесь, що ваш

	ПК підтримує версію OpenGL та/або відновіть драйвери відеокарти	
`glfwCreateWindow();`	Створення об’єкту вікна	
`glfwMakeContextCurrent(window);`	Створення контексту вікна	
`glewExperimental = GL_TRUE;`	Ініціалізація GLEW. GLEW управляє покажчиками на функції OpenGL для виклику його функцій. Установка значення glewExperimental в GL_TRUE дозволяє GLEW використовувати новітні техніки для управління функціоналом OpenGL. Також, якщо залишити цю змінну зі значенням за замовчуванням, то можуть виникнути проблеми звикористанням Core-profile	
`glfwGetFramebufferSize();`	Функція, яка отримує розміри вікна, та привласнює їх змінним	
`glViewport();`	OpenGL використовує дані, вказані через `glViewport()`, щоб перетворити 2D координати, які він обробляє, до координат на вашому екрані. Наприклад, оброблена точка розташування (-0,5,0,5) (як її остаточне перетворення) буде відображатися в (200,450) у координатах екрана. Можна задавати менші значення для ViewPort. У такому випадку	Перші 2 аргументу функції – це позиція нижнього лівого кута вікна. Третій і четвертий – це ширина і висота візуалізації вікна у пікселях

вся оформлена інформація буде менших розмірів, що дає змогу на іншій частині екрана зробити інші побудови.

`glfwWindowShouldClose();` Функція перевіряє на початку кожної ітерації циклу, чи отримала GLFW інструкцію до закриття, якщо так - то функція поверне **true** і ігровий цикл перестане працювати, після чого можна закрити додаток.

`glfwPollEvents();` Функція перевіряє, чи були викликані якісь події (наприклад, введення з клавіатури або переміщення миші). Звичайно функція обробки подій викликається на початку ітерації циклу.

`glfwSwapBuffers();` Функція заміняє кольоровий буфер (великий буфер, що містить значення кольору для кожного пікселя в GLFW вікні), який використовувався для показу під час поточної ітерації і показує результат на екрані.

Примітка: Подвійна буферізація

Коли додаток малює в єдиний буфер, то результуюче зображення може мерехтіти. Причина такого поведінки в тому, що малювання відбувається не миттєво, а по - піксельна зверху зліва, вправо вниз. Оскільки зображення відображається не миттєво, а поступово, то воно може мати чимало артефактів. Для уникнення цих проблем,

	віконні додатки використовують подвійну буферізацію. Передній буфер містить результуюче зображення, що відображається користувачу, в цей же час на задньому буфері ведеться малювання. Як тільки малювання завершується, ці буфери міняються місцями і зображення одноразово відображається користувачу. *Примітка 2:* Між функціями `glfwPollEvents();` `glfwSwapBuffers();` виконуються команди малювання.	
`glClearColor();`	Функція дозволяє очищувати екран від минулих дій, зафарбовуючи його відповідним кольором. Треба кожен раз зафарбовувати екран, щоб не бачити результати минулого малювання.	Аргументами функції є палітра RGBA. Перший – червоний колір; Другий – зелений колір; Третій – синій колір; Четвертий – альфа канал, зазвичай не використовується, тому поставити значення 1.0f. Для аргументів застосовувати значення типу **float.**
`glClear();`	Функція очищує екран, передаючи спеціальні біти, щоб вказати які буфери нам потрібно очистити. *Примітка:*	Аргументами можуть бути наступні буфери: 1) GL_COLOR_BUFFER_BIT

`glClearColor()` - це функція встановлення стану, а `glClear()` - це функція, що використовує стан, який використовує стан для визначення кольору заповнення екрана.

Буфер кольору

2) GL_DEPTH_BUFFER_BIT

Буфер глибини

3) GL_STENCIL_BUFFER_BIT

Буфер шаблону/трафарету

Одночасно можна очистити кілька буферів

`(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)`

```
glfwTerminate();
```

Після виходу з ігрового циклу, потрібно очистити виділені нам ресурси. Робиться це за допомогою функції `glfwTerminate()` наприкінці функції **main.**