Лекція 2

Лабораторна робота 2

ТРИКУТНИК

У минулому уроці ми таки подужали відкриття вікна і примітивне введення користувача. У цьому уроці ми розберемо всі ази виведення вершин на екран і скористаємося всіма можливостями OpenGL, як VAO, VBO, EBO для того, щоб вивести пару трикутників.

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 на 2D простір для відображення на екрані. Процес перетворення 3D координат на 2D координати управляється графічним конвеєром OpenGL. **Графічний конвеєр можна розділити на 2 великі частини:** перша частина перетворює 3D координати на 2D координати, а друга частина перетворює 2D координати на кольорові пікселі.

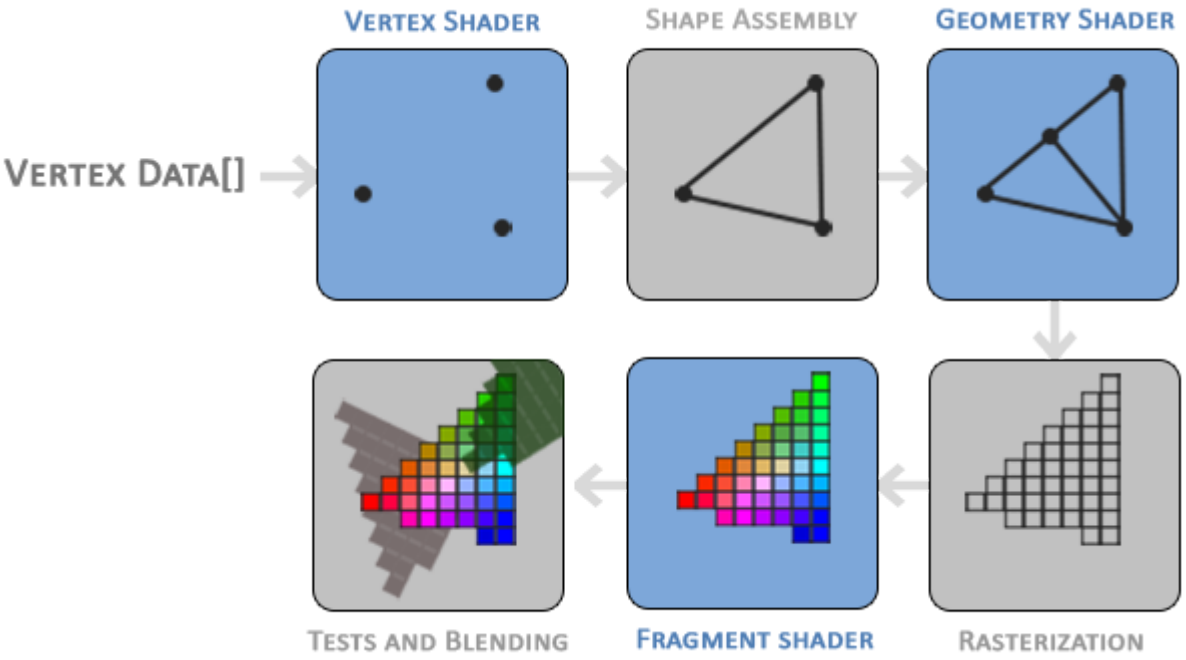
У цьому уроці ми детально обговоримо графічний конвеєр і те, як ми можемо його використовувати в плюс для створення красивих пікселів.

Є різниця між 2D координатами та пікселем. 2D координата – це дуже точне уявлення точки в 2D просторі, тоді як 2D піксель – це зразкове розташування в межах вашого екрана/вікна.

Графічний конвеєр приймає набір 3D координат і перетворює їх на кольорові 2D пікселі на екрані. Цей графічний контейнер можна розділити на кілька етапів, де кожен етап вимагає на вхід результату роботи минулого. Всі ці етапи вкрай спеціалізовані і можуть легко виконуватися паралельно. Через їхню паралельну природу більшість сучасних GPU мають тисячі маленьких процесорів для швидкої обробки даних графічного конвеєра за допомогою запуску великої кількості маленьких програм на кожному етапі конвеєра. **Ці маленькі програми називають шейдерами.**

Деякі з цих шейдерів можуть налаштовуватись розробником, що дозволяє нам писати **власні шейдери** для заміни стандартних. Це дає нам набагато більше можливостей тонкого настроювання специфічних місць конвеєра, і саме через те, що вони працюють на GPU, дозволяє нам зберегти процесорний час. Шейдери пишуться на OpenGL **Shading Language (GLSL)** і ми більше заглибимося до нього у наступному уроці.

На зображенні нижче можна побачити всі етапи графічного конвеєра. Сині частини описують етапи, для яких ми можемо специфікувати власні шейдери.



Як ви бачите, графічний конвеєр містить велику кількість секцій, кожна з яких обробляє певну частину перетворення ваших вершинних даних у повністю відрендерений піксель. Ми коротко пояснимо кожну частину спрощеним способом, щоб дати гарне уявлення про те, як працює конвеєр.

Як вхідні дані для графічного конвеєра ми передаємо список із трьох 3D-координат, які мають утворювати трикутник у масиві, який називається Вершинні дані; ці дані вершин є набором вершин. А вершина це набір даних для 3D координат. Дані цієї вершини представлені за допомогою атрибутів вершини які можуть містити будь-які дані, які ми хочемо, але для простоти давайте припустимо, що кожна вершина складається лише з 3D-позиції та деякого значення кольору.

Для того, щоб OpenGL знав, що робити з вашою колекцією координат і значень кольорів, OpenGL вимагає від вас підказки, які типи візуалізації ви хочете створити з даними. Чи хочемо ми, щоб дані відтворювалися як набір точок, набір трикутників чи, можливо, просто одна довга лінія? Ті підказки називаються примітивами і надаються OpenGL під час виклику будь-якої з команд малювання. Деякі з цих підказок

є`GL_POINTS`, `GL_TRIANGLES` і `GL_LINE_STRIP`.

Перша частина конвеєра - це вершинний шейдер що приймає на вхід одну вершину. Основною метою вершинного шейдера є перетворення 3D-координат у різні 3D-координати (докладніше про це пізніше), вершинний шейдер дозволяє виконувати базову обробку атрибутів вершин.

Вихід етапу вершинного шейдера передається в шейдер геометрії. Геометричний шейдер приймає як вхідні дані колекцію вершин, які утворюють примітив, і має можливість генерувати інші фігури, випускаючи нові вершини для формування нового (або іншого) примітиву(ів). У цьому прикладі він генерує другий трикутник із заданої форми.

Складання примітивів це етап який приймає як вхідні дані всі вершини (або вершину, якщо обрано `GL_POINTS`) з вершинного (або геометричного) шейдера, який формує один або більше примітивів і збирає всі точки у наданій примітивній формі; в даному випадку трикутник.

Вихід геометричного шейдера потім передається на етап растеризації де він відображає отримані примітиви на відповідні пікселі на фінальному екрані, у результаті чого утворюються фрагменти для використання фрагментним шейдером. Перед запуском фрагментних шейдерів виконується обрізка. Відсікання відкидає всі фрагменти, які знаходяться поза вашим полем зору, підвищуючи продуктивність.

> **Фрагмент в OpenGL — це всі дані, необхідні OpenGL для візуалізації одного пікселя.**

Основне призначення фрагментного шейдера це обчислення остаточного кольору пікселя, і зазвичай це стадія, на якій відбуваються всі додаткові ефекти OpenGL. Зазвичай фрагментний шейдер містить дані про 3D-сцену, які він може використовувати для розрахунку остаточного кольору пікселя (наприклад, світла, тіні, колір світла тощо).

Після визначення всіх відповідних значень кольорів остаточний об’єкт пройде ще один етап, який ми називаємо альфа-тест і етап змішування. На цьому етапі перевіряється відповідне значення глибини (і трафарету) фрагмента (ми перейдемо до них пізніше) і використовує їх, щоб перевірити, чи отриманий фрагмент знаходиться попереду чи позаду інших об’єктів і чи його слід відповідно відкинути. Сцена також перевіряє наявність альфа значення (альфа-значення визначають непрозорість об’єкта) і змішує кольори відповідно. Таким чином, навіть якщо вихідний колір пікселя обчислюється у фрагментному шейдері, остаточний колір пікселя все одно може бути чимось зовсім іншим під час візуалізації кількох трикутників.

Як бачите, графічний конвеєр є досить складним і містить багато настроюваних частин. Однак майже у всіх випадках нам доведеться працювати лише з вершинним і фрагментним шейдером. Геометричний шейдер

необов’язковий і зазвичай залишається стандартним шейдером. Існує також етап мозаїки та цикл зворотного зв’язку перетворення, які ми тут не зобразили, але про це пізніше.

У сучасному OpenGL необхідно визначити принаймні власний вершинний і фрагментний шейдер (у графічному процесорі немає вершинних/фрагментних шейдерів за замовчуванням). З цієї причини часто досить важко почати вивчати сучасний OpenGL, оскільки для того, щоб відобразити свій перший трикутник, потрібні багато знань.

Передача вершин

Щоб почати щось малювати, ми повинні спочатку надати OpenGL деякі вхідні дані вершин. OpenGL — це бібліотека тривимірної графіки, тому всі координати, які ми вказуємо в OpenGL, є тривимірними (x, y and z координата). OpenGL не просто перетворює ваші 3D-координати до 2D-пікселів на екрані; OpenGL обробляє 3D-координати лише тоді, коли вони знаходяться в певному діапазоні між ними `-1.0` і `1.0` по всіх 3 осях (x, y and z). Усі координати в межах цього проміжку називаються нормалізовані координати пристрою ,їх буде видно на вашому екрані (а всі координати за межами цього регіону не буде).

Оскільки ми хочемо відобразити один трикутник, ми хочемо вказати загалом три вершини, кожна з яких має тривимірну позицію. Ми визначаємо їх у нормалізованих координатах пристрою (видима область OpenGL) у `GLfloat` масиві:

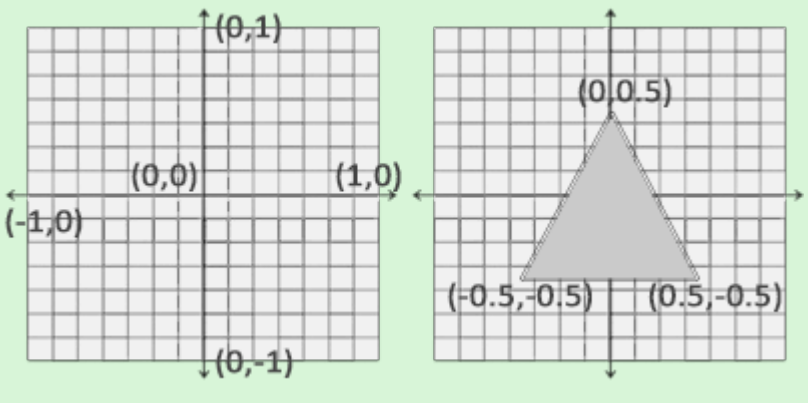
```
GLfloat vertices[] = {
    -0.5f, -0.5f, 0.0f,
     0.5f, -0.5f, 0.0f,
     0.0f,  0.5f, 0.0f
};
```

Оскільки OpenGL працює в 3D-просторі, а ми візуалізуємо 2D-трикутник, кожна вершина якого має Z координате `0.0`. Таким чином *глибина* трикутника залишається такою самою, що робить його схожим на 2D.

Нормалізовані координати пристрою (NDC)

Після того, як координати вершини оброблені у вершинному шейдері, вони мають бути в нормалізовані координати пристрою (NDC) це

невеликий простір, де x, y та z лежать у проміжку від `-1.0` до `1.0`. Будь-які координати, які виходять за межі цього діапазону, будуть відхилені/обрізані та не відображатимуться на екрані. Нижче ви можете побачити трикутник, який ми вказали в нормалізованих координатах пристрою:



На відміну від звичайних екранних координат, додатна вісь ординат вказує вгору та `(0,0)` координати знаходяться в центрі графіка, а не вгорі ліворуч. Зрештою ви необхідно, аби усі (перетворені) координати опинилися в цьому просторі координат, інакше вони не будуть видимі.

Далі, координати NDC буде перетворено на **координати екранного простору** через **перетворення вікна перегляду** Viewport використовуючи надані вами дані `glViewport`. Отримані координати екранного простору потім перетворюються на фрагменти і є вхідними даними для вашого шейдера фрагментів.

Після визначення вершинних даних потрібно передати в перший етап графічного конвеєра: у вершинний шейдер. Це робиться таким чином: виділяємо пам'яті на GPU, куди ми збережемо наші вершинні дані, вкажемо OpenGL як він повинен інтерпретувати передані йому дані та передамо GPU кількість переданих нами даних. Потім вершинний шейдер обробить в пам’яті таку кількість вершин, яку ми йому повідомили.

Ми керуємо цією пам'яттю через, так звані, об'єкти вершинного буфера (VBO), які можуть зберігати велику кількість вершин в пам'яті GPU. Перевага використання таких об'єктів буфера, що ми можемо надсилати у 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 ми намагатимемося відправляти якомога більше даних за один раз. Але як тільки дані будуть вже в GPU, вершинний шейдер отримає їх майже миттєво.

VBO це наша перша зустріч з об'єктами, описаних у першому уроці. Як і будь-який об'єкт в OpenGL, цей буфер має унікальний ідентифікатор. Ми можемо створити VBO за допомогою функції glGenBuffers:

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

OpenGL має велику кількість різних типів об'єктів буферів. Тип VBO - GL_ARRAY_BUFFER. OpenGL дозволяє прив'язувати безліч буферів, якщо вони мають різні типи. Ми можемо прив'язати GL_ARRAY_BUFFER до нашого буфера за допомогою glBindBuffer:

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

З цього моменту будь-який виклик, що використовує буфер, працюватиме з VBO. Тепер ми можемо викликати glBufferData копіювання вершинних даних у цей буфер.

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);
```

glBufferData це функція, мета якої - копіювання даних користувача у зазначений буфер. Перший її аргумент - це тип буфера, в який ми хочемо скопіювати дані (наш VBO зараз прив'язаний до GL_ARRAY_BUFFER). Другий аргумент визначає кількість даних (у байтах), які хочемо передати буферу. Третій аргумент – це самі дані.

Четвертий аргумент визначає як ми хочемо, щоб відеокарта працювала з переданими їй даними. Існує 3 режими:

- **GL_STATIC_DRAW: дані або ніколи не будуть змінюватися, або будуть змінюватися дуже рідко;**
- **GL_DYNAMIC_DRAW: дані будуть змінюватися досить часто;**
- **GL_STREAM_DRAW: дані будуть змінюватися під час кожного малювання.**

Дані про позицію трикутника не змінюватимуться і тому ми вибираємо GL_STATIC_DRAW. Якщо, наприклад, у нас був би буфер, значення якого

змінювалося б дуже часто — ми використовували б GL_DYNAMIC_DRAW або GL_STREAM_DRAW, надавши таким чином відеокарті інформацію, що дані цього буфера потрібно зберігати в області пам'яті, найбільш швидкої на запис.

Зараз ми зберегли вершинні дані на GPU об'єкт буфера, названого VBO.

Далі ми повинні створити вершинний і фрагментний шейдер для фактичної обробки даних.

Вершиний шейдер

Вершинний шейдер - один із програмованих шейдерів. Сучасний OpenGL вимагає, щоб був заданий вершинний і фрагментний шейдер якщо ми хочемо щось малювати, тому ми надамо два дуже простих шейдери для малювання нашого трикутника. У наступному уроці ми обговоримо шейдери докладніше.

На початку ми повинні написати сам шейдер спеціальною мовою GLSL (OpenGL Shading Language), а потім зібрати його, щоб програма могла з ним працювати. Ось код найпростішого шейдера:

```
#version 330 core

layout (location = 0) in vec3 position;

void main()

{

    gl_Position = vec4(position.x, position.y, position.z, 1.0);

}
```

Як ви можете помітити, GLSL дуже схожий на C. Кожен шейдер починається з встановлення його версії. З OpenGL версії 3.3 і від версії GLSL збігаються з версіями OpenGL (наприклад версія GLSL 420 збігається з OpenGL версії 4.2). Також ми явно зазначили, що використовуємо core profile.

Далі ми вказали усі вхідні вершинні атрибути у вершинному шейдері за допомогою ключового слова in. Зараз нам треба працювати лише з даними про позицію, тому ми вказуємо лише один вершинний атрибут. GLSL має векторний тип даних, що містить від 1 до 4 чисел з плаваючою точкою. Оскільки вершини мають тривимірні координати, то і ми створюємо vec3 з назвою position. Також

ми явно вказали позицію нашої змінної через layout (location = 0), пізніше ви побачите, навіщо ми це зробили.

Vector

У графічному програмуванні ми досить часто використовуємо математичну концепцію вектора, оскільки вона відмінно представляє позиції/напрямки в будь-якому просторі, а також має корисні математичні властивості. Максимальний розмір вектора в GLSL – 4 елементи, а доступ до кожного з елементів можна отримати через vec.x, vec.y, vec.z та vec.w відповідно. Зауважте, що компонент vec.w не використовується як позиція в просторі (ми ж працює в 3D, а не в 4D), але вона може бути корисна при роботі з поділом перспективи (perspective division). Ми обговоримо вектор глибше в наступному уроці.

Для позначення результату роботи вершинного шейдера ми маємо присвоїти значення зумовленої змінної gl_Position, яка має тип vec4. Після закінчення роботи main функції, щоб ми не передали в gl_Position, воно буде використано як результат роботи вершинного шейдера. Оскільки наш вхідний вектор тривимірний, ми повинні перетворити його на чотиривимірний. Ми можемо зробити це просто передавши компоненти vec3 в vec4, а компонент w задати значення 1.0f (Ми пояснимо чому так пізніше).

Цей вершинний шейдер, ймовірно, найпростіший шейдер, який тільки можна придумати, оскільки він не обробляє жодних даних, а просто передає ці дані на вихід. У реальних додатках вхідні дані не нормалізовані, тому на початку потрібно нормалізувати.

Компіляція шейдера

Ми написали вихідний код шейдера (зберігається в рядку C), але щоб цим шейдерів міг користуватися OpenGL, його треба зібрати(компілювати).

На початку ми маємо створити об'єкт шейдера. А оскільки доступ до створених об'єктів здійснюється через ідентифікатор — ми зберігатимемо його в змінній з типом GLuint, а створюватимемо його через glCreateShader:

```
GLuint vertexShader;

vertexShader = glCreateShader(GL_VERTEX_SHADER);
```

Під час створення шейдера ми повинні вказати тип шейдера, що створюється. Оскільки нам потрібний вершинний шейдер, ми вказуємо GL_VERTEX_SHADER.

Далі ми прив'язуємо вихідний код шейдера до об'єкта шейдера та компілюємо його.

```
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
```

Функція glShaderSource як перший аргумент приймає шейдер, який потрібно зібрати. Другий аргумент визначає кількість рядків. У нашому випадку рядок лише один. Третій параметр — це вихідний код шейдера, а четвертий параметр ми залишимо в NULL.

Швидше за все ви захочете перевірити успішність компіляції шейдера. І якщо шейдер не був зкомпільований — отримаємо помилки, які виникли під час компіляції. Перевірка на наявність помилок відбувається так:

```
GLint success;

GLchar infoLog[512];

glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
```

Спочатку ми оголошуємо число для визначення успішності складання та контейнер для зберігання помилок (якщо вони з'явилися). Потім ми перевіряємо успішність за допомогою glGetShaderiv. Якщо збірка провалиться, ми зможемо отримати повідомлення про помилку за допомогою glGetShaderInfoLog і вивести цю помилку:

```
if(!success)

{

   glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);

   std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" <<
infoLog << std::endl;

}
```

Якщо під час компіляції вершинного шейдера не виявлено жодних помилок, його буде скомпільовано.

Фрагментний шейдер

Фрагментний шейдер – це другий та останній шейдер, який нам знадобиться, щоб відмалювати трикутник. Фрагментний шейдер займається обчисленням кольорів пікселів. В ім'я простоти наш фрагментний шейдер виводитиме лише помаранчевий колір.

Колір у комп'ютерній графіці представляється як масив із 4 значень: червоний, зелений, синій та прозорість; така компонентна база називається RGBA. Коли ми задаємо колір в OpenGL або в GLSL, ми задаємо величину кожного компонента між 0.0 і 1.0. Якщо ми встановимо величину червоного і зеленого компонентів в 1.0f, ми отримаємо суміш цих кольорів — жовтий. Комбінація з трьох компонентів дає близько 16 мільйонів різних кольорів.

```
#version 330 core

out vec4 color;

void main()

{

     color = vec4(1.0f, 0.5f, 0.2f, 1.0f);

}
```

Фрагментний шейдер на вихід вимагає лише значення кольору, що є 4 компонентним вектором. Ми можемо вказати вихідну змінну за допомогою ключового слова out, а назвемо цю змінну color. Потім ми просто

встановлюємо значення цієї змінної vec4 із непрозорим помаранчевим кольором.

Процес складання фрагментного шейдера аналогічний складання вершинного, тільки потрібно вказати інший тип шейдера: GL_FRAGMENT_SHADER:

```
GLuint fragmentShader;
fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);
```

Обидва шейдери були зібрані і тепер тільки залишилося зв'язати їх у програму, щоб ми могли використовувати їх для малювання.

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 малюванні об'єктів, і ця програма буде використовуватися при виклику команд малювання.

При з'єднанні шейдерів у програму вихідні значення одного шейдера зіставляються з вхідними значеннями іншого шейдера. Ви також можете отримати помилки під час з'єднання шейдерів, якщо вхідні та вихідні значення не співпадають.

Створити програму дуже просто:

```
GLuint shaderProgram;
shaderProgram = glCreateProgram();
```

Функція glCreateProgram створює програму та повертає ідентифікатор цієї програми. Тепер нам треба приєднати наші зібрані шейдери до програми, а потім зв'язати їх за допомогою glLinkProgram:

```
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
```

Код має бути досить зрозумілим, ми приєднуємо шейдери до програми та зв’язуємо їх через `glLinkProgram`.

> **Також як і зі складанням шейдера ми можемо отримати успішність зв'язування та повідомлення про помилку. Єдина відмінність — що замість glGetShaderiv та glGetShaderInfoLog ми використовуємо:**

```
glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
If (!success) {
      glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
      …
}
```

Результатом є програмний об’єкт, який ми можемо активувати шляхом виклику `glUseProgram` з новоствореним програмним об’єктом як аргументом:

```
glUseProgram(shaderProgram);
```

Кожен виклик шейдера та функцій відтворення буде використовувати наш об'єкт програми (і відповідно наші шейдери).

Не забудьте видалити створені шейдери після зв'язування. Вони нам більше не знадобляться.

```
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

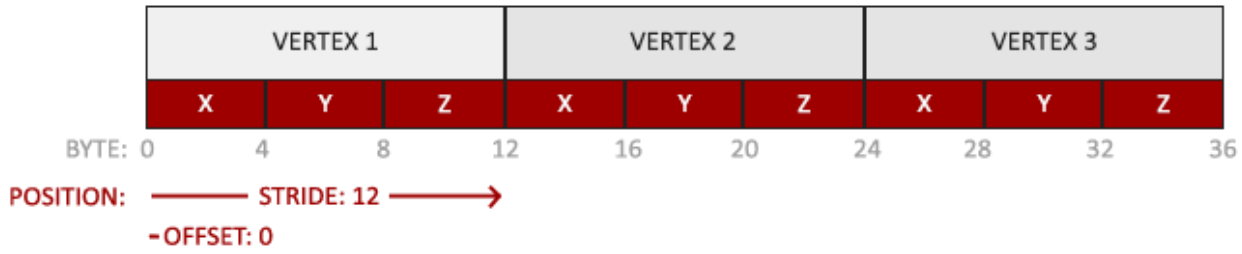
Прямо зараз ми надіслали вхідні дані вершин до GPU та вказали GPU, як він має обробляти дані вершин у вершинному та фрагментному шейдері. Ми майже

там, але ще не зовсім. OpenGL ще не знає, як він має інтерпретувати дані вершин у пам'яті та як він має з'єднувати дані вершин з атрибутами вершинного шейдера. Ми розповімо OpenGL, як це зробити.

Зв'язування атрибутів вершин

Вершиний шейдер дозволяє нам вказувати будь-які вхідні дані, які ми хочемо, у формі вершинних атрибутів, і хоча це забезпечує велику гнучкість, це означає, що ми повинні вручну вказувати, яка частина наших вхідних даних йде до того чи іншого атрибута вершини у вершинному шейдері. Це означає, що ми повинні вказати, як OpenGL має інтерпретувати дані вершини перед відтворенням.

Наші дані буфера вершин відформатовані таким чином:



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

Маючи ці знання, ми можемо сказати OpenGL, як він має інтерпретувати дані вершини (атрибут вершини), використовуючи `glVertexAttribPointer`:

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
```

```
glEnableVertexAttribArray(0);
```

Функція `glVertexAttribPointer` має досить багато параметрів, тому давайте уважно їх розглянемо:

- Перший аргумент описує, який аргумент шейдера ми хочемо налаштувати. Ми хочемо специфікувати значення аргументу position, позиція якого була вказана так: layout (location = 0).
- Наступний аргумент визначає розмір аргументу у шейдері. Оскільки ми використовували vec3 ми вказуємо 3.
- Третій аргумент визначає тип даних, що використовується. Ми вказуємо GL_FLOAT, оскільки vec у шейдері використовує числа з плаваючою точкою.

- **Четвертий аргумент вказує на необхідність нормалізувати вхідні дані. Якщо ми вкажемо GL_TRUE, всі дані будуть розташовані між 0 (-1 для знакових значень) і 1. Нам нормалізація не потрібно, тому ми залишаємо GL_FALSE;**
- **П'ятий аргумент називається кроком та описує відстань між наборами даних. Ми також можемо вказати крок 0 і тоді OpenGL вирахує крок (працює тільки з щільно упакованими наборами даних). Як отримати істотну користь від цього аргументу ми розглянемо пізніше.**
- **Останній параметр має тип GLvoid* і тому вимагає такого дивного наведення типів. Це усунення початку даних у буфері. У нас буфер не має усунення і тому ми вказуємо 0.**

Кожен атрибут вершини отримує значення з пам'яті, що керується VBO, яка в даний момент є прив'язаною до GL_ARRAY_BUFFER. Відповідно, якби ми викликали glVertexAttribPointer з іншим VBO — то вершинні дані були б взяті з іншого VBO.

Після того як ми повідомили OpenGL як він повинен інтерпретувати вершинні дані, ми повинні включити атрибут за допомогою glEnableVertexAttribArray. Таким чином, ми передамо вершинному атрибуту позицію аргументу. Після того, як ми всі налаштували, ми ініціалізували вершинні дані в буфері за допомогою VBO, встановили вершинний і фрагментний шейдер і повідомили OpenGL як зв'язати вершинний шейдер і вершинні дані. Відображення об'єкта в OpenGL буде виглядати якось так:

```
// 0. Копіюємо масив з вершинами в буфер OpenGL
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 1. Потім встановимо покажчики на вершинні атрибути
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
// 2. Використовуємо нашу шейдерну програму
glUseProgram(shaderProgram);
// 3. Тепер уже малюємо об'єкт
someOpenGlFunctionThatDrawsOutTriangle();
```

Нам потрібно повторювати цей процес кожного разу, коли ми хочемо намалювати об’єкт. Це може виглядати не так багато, але уявіть, що ми маємо більше 5 атрибутів вершин і, можливо, 100 різних об’єктів (що не є рідкістю). Зв’язування відповідних буферних об’єктів і налаштування всіх атрибутів вершин для кожного з цих об’єктів швидко стає громіздким процесом. Що, якби ми якимось чином могли зберегти всі ці конфігурації стану в об’єкт і просто зв’язати цей об’єкт, щоб відновити його стан?

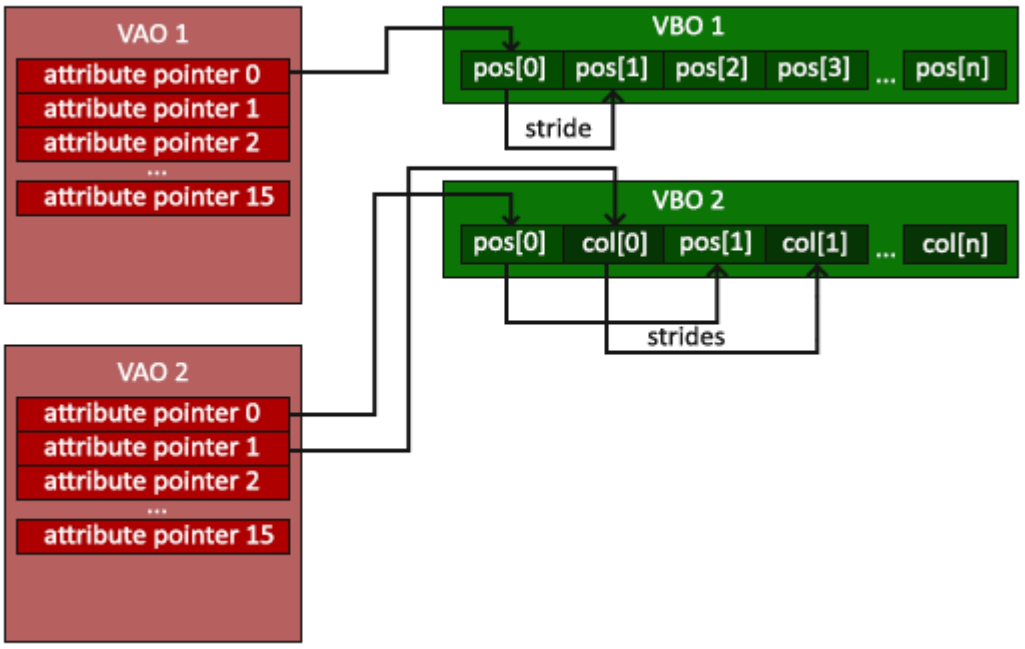
Vertex Array Object (**Об’єкт масиву вершин)**

Об'єкт вершинного масиву (VAO) може бути прив'язаний як і VBO і після цього всі наступні виклики вершинних атрибутів будуть зберігатися в VAO. Перевага цього методу в тому, що нам потрібно налаштувати атрибути лише один раз, а всі наступні рази буде використано конфігурацію VAO. Також такий метод спрощує зміну вершинних даних та конфігурацій атрибутів простим прив'язуванням різних VAO.

> **Core OpenGL вимагає, щоб ми використовували VAO для того, щоб OpenGL знав як працювати з нашими вхідними вершинами. Якщо ми не вкажемо VAO, OpenGL може відмовитися малювати будь-що.**

VAO зберігає наступне:

- **Виклики glEnableVertexAttribArray або glDisableVertexAttribArray.**
- **Конфігурація атрибутів виконана через glVertexAttribPointer.**
- **VBO асоційовані з вершинними атрибутами за допомогою glVertexAttribPointer**



Процес створення VAO виглядає подібно до процесу VBO:

```
GLuint VAO;

glGenVertexArrays(1, &VAO);
```

Для того, щоб використовувати VAO все, що вам потрібно зробити, це прив'язати VAO за допомогою glBindVertexArray. Тепер ми повинні налаштувати/прив'язати необхідні VBO та покажчики на атрибути, а наприкінці відв'язати VAO для подальшого використання. І тепер, кожного разу, коли ми хочемо малювати об'єкт, ми просто прив'язуємо VAO з необхідними нам налаштуваннями перед відображенням об'єкта. Виглядати це все має приблизно так:

```
// ..:: Код ініціалізації (виконується якось (якщо, звичайно, об'єкт не буде часто змінюватися)) :: ..
// 1. Прив'язуємо VAO
glBindVertexArray(VAO);
// 2. Копіюємо наш масив вершин у буфер для OpenGL
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. Встановлюємо покажчики на вершинні атрибути
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(0);
//4. Відв'язуємо VAO
glBindVertexArray(0);

[...]

// ..:: Код відтворення (в ігровому циклі) :: ..
// 5. Малюємо об'єкт
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
someOpenGLFunctionThatDrawsOurTriangle();
glBindVertexArray(0);
```

У OpenGL відв'язування об'єктів це звичайна справа. Як мінімум, просто для того, щоб випадково не зіпсувати конфігурацію.

От і все! Все, що ми робили протягом мільйонів сторінок, підводило нас до цього моменту. VAO, що зберігає вершинні атрибути та необхідний VBO. Найчастіше, коли ми маємо множинні об'єкти для малювання, ми на початку генеруємо і конфігуруємо VAO і зберігаємо їх для подальшого використання. І коли треба буде відмалювати один із наших об'єктів ми просто використовуємо збережений VAO.

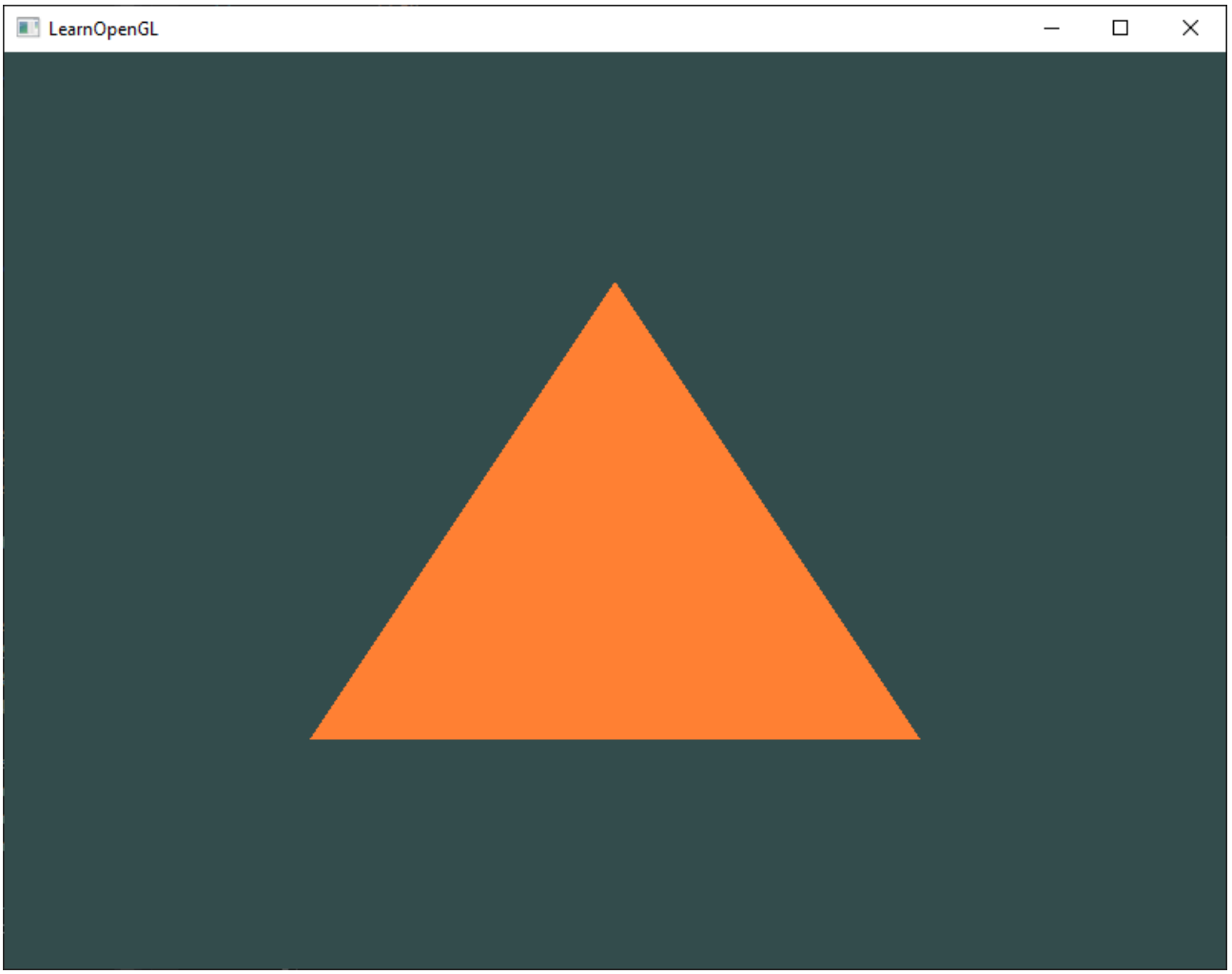
Трикутник, якого ми всі чекали

Для відображення наших об'єктів OpenGL надає нам функцію glDrawArrays. Вона використовує активний шейдер та встановлений VAO для малювання зазначених примітивів.

```
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 3);
glBindVertexArray(0);
```

Функція glDrawArrays приймає як перший аргумент OpenGL примітив, який потрібно відмалювати. Оскільки ми хочемо малювати трикутник і тому, що ми не хочемо брехати вам, ми вказуємо GL_TRIANGLES. Другий аргумент вказує початковий індекс масиву з вершинами, який ми хочемо відмалювати, ми просто залишимо 0. Останній аргумент вказує кількість вершин для малювання, що нам потрібно відмалювати 3 (довжина одного трикутника - 3 вершини).

Тепер можна зібрати та запустити написаний код. Ви побачите наступний результат:



Вихідний код повної програми можна знайти тут .

Якщо ваш результат виглядає не так, ви, ймовірно, зробили щось не так, тому перевірте повний вихідний код і подивіться, чи ви нічого не пропустили.

Element Buffer Object (**Об'єкти буфера елементів)**

Останнє про що ми сьогодні поговоримо на тему відображення вершин - це element buffer objects (EBO). Для того, щоб пояснити, що це і як працює, краще навести приклад: припустимо, що нам треба відмалювати не трикутник, а чотирикутник. Ми можемо намалювати чотирикутник за допомогою 2 трикутників (OpenGL в основному працює з трикутниками).

Відповідно треба буде оголосити наступний набір вершин:

```
GLfloat vertices[] = {
    // Перший трикутник
     0.5f, 0.5f, 0.0f, // Верхній правий кут
     0.5f, -0.5f, 0.0f, // Нижній правий кут
    -0.5f, 0.5f, 0.0f // Верхній лівий кут
    // Другий трикутник
```

```
    0.5f, -0.5f, 0.0f, // Нижній правий кут
    -0.5f, -0.5f, 0.0f, // Нижній лівий кут
    -0.5f, 0.5f, 0.0f // Верхній лівий кут
};
```

Як ви можете помітити: ми двічі вказали нижню праву та верхню ліву вершину. Це не дуже раціональне використання ресурсів, оскільки ми можемо описати прямокутник 4 вершинами замість 6. Проблема стає ще більш вагомою, коли ми маємо справу з більшими моделями, у яких може бути більше 1000 трикутників. Найправильніше вирішення цієї проблеми - це зберігати тільки унікальні вершини, а потім окремо вказувати порядок, в якому ми хочемо, щоб здійснювалося відмальовування. У нашому випадку нам потрібно було б зберігати тільки 4 вершини, а потім вказати порядок у якому їх треба відмалювати. Було б чудово, якби OpenGL надавав таку можливість.

На щастя, EBO це саме те, що нам потрібно. EBO - це буфер, на зразок VBO, але він зберігає індекси, які OpenGL використовує, щоб вирішити якусь вершину відмалювати. Це називається малювання за індексами (indexed drawing) і є рішенням вищезгаданої проблеми. На початку нам треба буде вказати унікальні вершини та індекси для малювання їх як трикутників:

```
GLfloat vertices[] = {
    0.5f, 0.5f, 0.0f, // Верхній правий кут
    0.5f, -0.5f, 0.0f, // Нижній правий кут
    -0.5f, -0.5f, 0.0f, // Нижній лівий кут
    -0.5f, 0.5f, 0.0f // Верхній лівий кут
};
GLuint indices[] = { // Пам'ятайте, що ми починаємо з 0!
    0, 1, 3 // Перший трикутник
    1, 2, 3 // Другий трикутник
};
```

Ви бачите, що при використанні індексів нам потрібно лише 4 вершини замість 6. Далі нам потрібно створити EBO:

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

Так само як і з VBO ми прив'язуємо EBO та копіюємо індекси у цей буфер через glBufferData. Також, як і з VBO, ми поміщаємо виклики між командами зв'язування та розв'язування (glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0), тільки цього разу типом буфера є GL_ELEMENT_ARRAY_BUFFER.

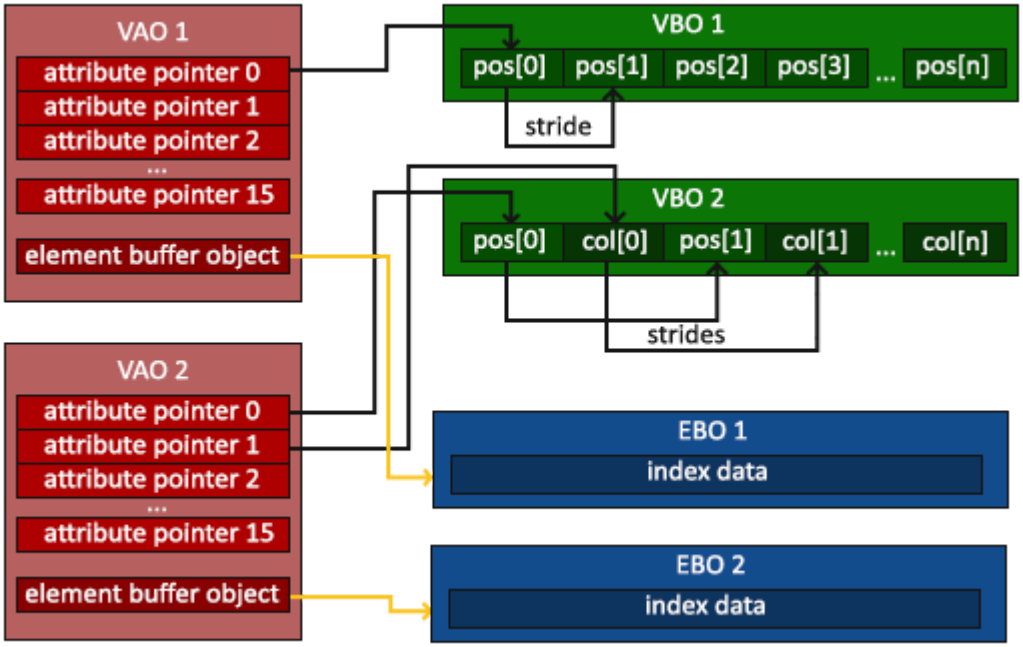
```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
GL_STATIC_DRAW);
```

Зауважте, що зараз ми передаємо GL_ELEMENT_ARRAY_BUFFER як цільовий буфер. Останнє, що нам залишилося зробити - це замінити виклик glDrawArrays на виклик glDrawElements для того, щоб вказати, що ми хочемо відмалювати трикутники з буфера з індексами. Коли використовується glDrawElements, робиться малюнок з прив'язаного в даний момент EBO:

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Перший аргумент описує примітив, який ми хочемо відмалювати, як і в glDrawArrays. Другий аргумент – це кількість елементів, які ми хочемо відмалювати. Ми вказали 6 індексів, тому передаємо функції 6 вершин. Третій аргумент – це тип даних індексів, у нашому випадку – це GL_UNSIGNED_INT. Останній аргумент дозволяє нам задати зміщення в EBO (або передати сам масив з індексами, але при використанні EBO так не роблять), тому ми вказуємо просто 0.

Функція glDrawElements бере індекси з поточного прив'язаного до GL_ELEMENT_ARRAY_BUFFER EBO. Це означає, що якщо ми повинні щоразу прив'язувати різні EBO. Але VAO може зберігати і EBO.



VAO зберігає виклики glBindBuffer, якщо метою є GL_ELEMENT_ARRAY_BUFFER. Це також означає, що він зберігає і виклики відв'язування, тому переконайтеся, що ви не відв'язали ваш EBO перед тим як відв'язати VAO, інакше у вас взагалі не буде прив'язаного EBO.

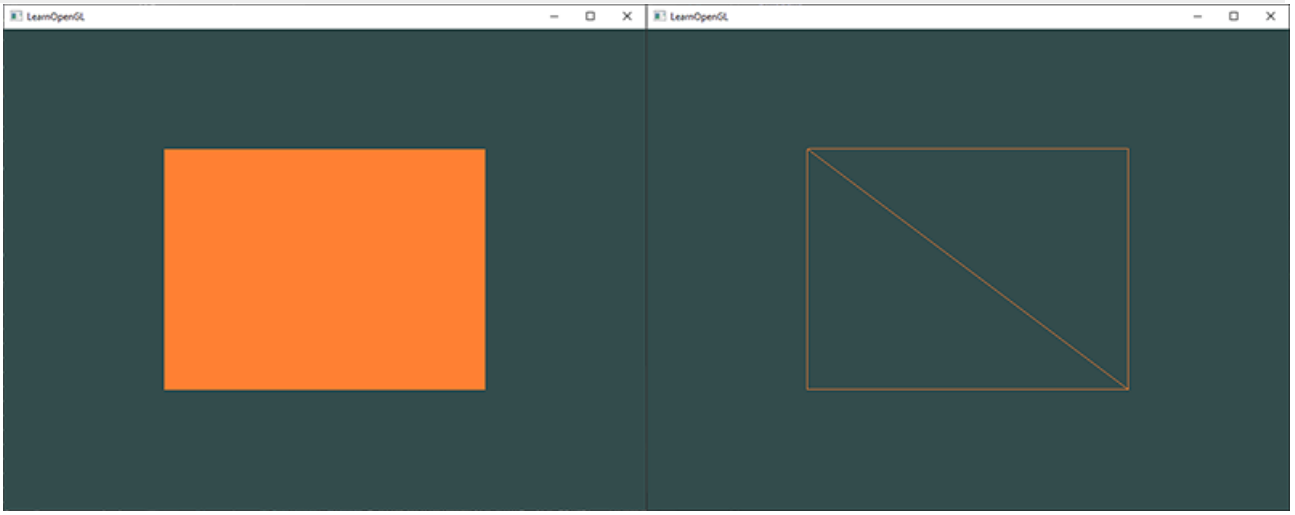
Отриманий код ініціалізації та малювання тепер виглядає приблизно так:

```
//..:: Код ініціалізації ::..
// 1. Прив'язуємо VAO
glBindVertexArray(VAO);
    // 2. Копіюємо наші вершини в буфер для OpenGL
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
    // 3. Копіюємо наші індекси в буфер для OpenGL
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
    // 3. Встановлюємо покажчики на вершинні атрибути
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
    glEnableVertexAttribArray(0);
// 4. Відв'язуємо VAO (НЕ EBO)
glBindVertexArray(0);

[...]
```

```
// ..:: Код відтворення (в ігровому циклі) :: ..
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0)
glBindVertexArray(0);
```

Запуск програми має дати зображення, як показано нижче. Ліве зображення має виглядати знайомим, а праве – це намальований прямокутник каркасний режим. Каркасний прямокутник показує, що прямокутник справді складається з двох трикутників.



Режим Wireframe

Щоб відобразити ваші трикутники в цьому режимі, вкажіть OpenGL, як малювати примітиви за допомогою glPolygonMode(GL_FRONT_AND_BACK, GL_LINE). Перший аргумент вказуємо, що хочемо малювати передню і задню частини всіх трикутників, а другий аргумент, що хочемо малювати лише лінії. Щоб повернутися до початкової конфігурації, викличте glPolygonMode(GL_FRONT_AND_BACK, GL_FILL).

Якщо у вас є якісь помилки, поверніться назад і подивіться, чи ви щось не пропустили. Ви можете знайти повний вихідний код тут.

Якщо вам вдалося намалювати трикутник або прямокутник, як це зробили ми тоді, вітаємо, вам вдалося пройти одну з найскладніших частин сучасного OpenGL: намалювати свій перший трикутник. Це складна частина, тому що перед тим, як намалювати свій перший трикутник, потрібно мати велику

частину знань. На щастя, тепер ми подолали цей бар'єр, і, сподіваюся, наступні розділи будуть набагато легшими для розуміння.

Додаткові ресурси

- antongerdelan.net/hellotriangle: погляд Антона Герделана на візуалізацію першого трикутника.
- open.gl/малюнок: погляд Олександра Овервоорда на рендеринг першого трикутника.
- antongerdelan.net/vertexbuffers: деякі додаткові відомості про об'єкти буфера вершин.
- learnopengl.com/In-Practice/Debugging: у цьому розділі багато кроків; якщо ви застрягли, можливо, варто почитати трохи про налагодження в OpenGL (до розділу виводу налагодження).

Вправи

Щоб справді добре зрозуміти обговорювані концепції, було створено кілька вправ. Рекомендується опрацювати їх, перш ніж переходити до наступного завдання, щоб переконатися, що ви добре розумієте, що відбувається.

1. Спробуйте намалювати 2 трикутники один поруч з одним за допомогою `glDrawArrays` додавши більше вершин до ваших даних:рішення.
2. Тепер створіть ті самі 2 трикутники, використовуючи два різних VAO та VBO для їхніх даних:рішення.
3. Створіть дві шейдерні програми, де друга програма використовує інший фрагментний шейдер, який виводить жовтий колір; знову намалюйте обидва трикутники, де один виведе жовтий колір:рішення.