ВСТУП в OpenGL

OpenGL(англ. *«**Open G**raphics **L**ibrary»*) у більшості випадків розглядається як API, що надає великий набір функцій, які ми можемо використовувати для керування графікою та зображеннями. Якщо конкретніше, то OpenGL є специфікацією, розробленою та підтримуваною **Khronos Group**.

Незалежно від того, чи намагаєтесь ви вивчити OpenGL в освітніх/кар'єрних цілях або просто шукаєте нове хобі,**цей курс з графічного програмування навчить вас основам та всім необхідним деталям роботи з OpenGL**.

Мета цього курсу — показати вам все, що є в сучасному OpenGL, простим для розуміння способом наочних прикладів, а також надати бекграунд для ваших подальших досліджень/поглиблень на цю тему.

Цей матеріал націлений як на тих людей, які не мають досвіду у графічному програмуванні, так і на досвідченіших програмістів, яким було б цікаво почитати та систематизувати свої знання. У цьому матеріалі OpenGL також обговорюються практичні концепції, які при додатковій креативності зможуть перетворити ваші ідеї на справжні 3D-додатки.

Вміння які буде отримано

Вивчення (і використання) OpenGL вимагає глибоких знань графічного програмування та розуміння того, як OpenGL працює «під капотом», щоб реально отримати максимальну користь із вашого майбутнього досвіду. Тому почнемо з обговорення основних графічних аспектів, з'ясування фактичної роботи OpenGL для промальовування пікселів на екрані, а також того, як можна використовувати ці знання для створення якихось незвичайних ефектів.

Крім основного базису, також буде розглянуто безліч корисних технік, які можна використовувати для своїх додатків, таких як переміщення сцени, створення гарного освітлення, завантаження об'єктів з програм моделювання, застосування постобробки та багатьох інших. Також буде створено невелику гру, застосувавши отримані знання з OpenGL.

Оскільки OpenGL є графічним API, а не власною платформою, то для роботи з ним знадобиться мова програмування. У нашому випадку це **мова C++** тому знадобляться знання з C++. Не обов'язково бути експертом C++, але потрібно вміти написати щось більше, ніж програму «Hello, world!».

Крім того, буде використовуватися математика (лінійна алгебра і геометрія разом з тригонометрією), яка також буде пояснена в міру необхідності.

Структура

Навчальний матеріал розбитий на послідовні розділи. Кожен із розділів містить уроки, на яких докладно розглядаються різні концепції. Оскільки уроки є послідовними, то найкраще починати від початку, оскільки кожному наступному уроці використовуються концепції, які розглядалися попередньому.

Заняття №1. Що таке OpenGL?

OpenGL (англ. *«**Open** **G**raphics **L**ibrary»*) розглядається як
API (англ. *«**A**pplication **P**rogramming **I**nterface» = «Интерфейс прикладного програмування»*), що надає великий набір функцій, які можна використовувати для керування графікою та зображеннями. Якщо конкретніше, то **OpenGL є специфікацією, розробленою та підтримуваною Khronos Group**.

Специфікація OpenGL визначає, яким має бути результат/виведення кожної функції, і як вона має виконуватися. А ось реалізація цієї специфікації залежить від конкретних розробників. Оскільки специфікація OpenGL не надає подробиць реалізації, то фактично розроблені версії OpenGL можуть мати різні реалізації доти, поки їх результати відповідають специфікації (і, отже, є однаковими для користувача).

Люди, які розробляють бібліотеки OpenGL, зазвичай є виробниками відеокарт. Кожна відеокарта, що була придбана, підтримує певні версії OpenGL, розроблені спеціально під цю лінійку відеокарт. При використанні програмного забезпечення від Apple бібліотека OpenGL підтримується власне розробниками Apple, а в Linux існує цілий набір версій графічних постачальників та адаптації від **опенсорс**-спільноти цих бібліотек Це також означає, що кожного разу, коли OpenGL демонструє дивну поведінку, якої не повинно бути, то це, швидше за все, вина виробників відеокарт (або тих, хто розробляв/підтримує цю бібліотеку).

Оскільки більшість реалізацій OpenGL створені виробниками відеокарт, то щоразу, коли знаходиться **помилка** в реалізації це зазвичай вирішується оновленням драйверів вашої відеокарти. Ці драйвери включають останні версії OpenGL, які підтримує ваша відеокарта. Це одна з основних причин, з яких завжди рекомендується оновлювати графічні драйвери.

Khronos привселюдно розміщує всі специфікації документів для всіх версій OpenGL. Зацікавлений читач може переглянути специфікацію OpenGL версії 3.3 (яка буде використана в цьому курсі) **тут**, де він зможе заглибитись у деталі OpenGL (зверніть увагу, що в даній специфікації в основному просто описуються результати, а не реалізації). Ця специфікація також надає чудову довідкову інформацію для розуміння того, який результат виконання функцій має бути.

Core-profile проти безпосереднього режиму

У старі часи використання OpenGL означало розробку в **безпосередньому режимі** (так званому*«Конвеєрі фіксованих функцій»*), який був простим у використанні методом для малювання графіки. Більшість функціоналу OpenGL була прихована всередині бібліотеки, і розробники не мали контролю над тим, як OpenGL виконує свої обчислення. Оскільки розробники прагнули більшої гнучкості, то згодом специфікації стали гнучкішими; розробники отримали більше контролю за своєю графікою. Безпосередній режим справді простий у використанні та розумінні, але він також вкрай неефективний. З цієї причини, починаючи зі специфікації версії 3.2, функціонал безпосереднього режиму почали вважати застарілим, мотивуючи цим розробників перейти на розробку в режимі **core-profile**, який є розділом специфікації OpenGL з повністю віддаленим застарілим функціоналом.

Використовуючи **режим core-profile**, OpenGL змушує застосовувати сучасні техніки. Щоразу, коли ми намагаємося використовувати одну із застарілих функцій OpenGL у режимі core-profile, OpenGL викидає помилку та зупиняє малювання. Перевагою вивчення сучасного підходу є його гнучкість та ефективність. Проте вчити його вже дещо

складніше. Безпосередній режим досить сильно абстрагувався від реальних операцій OpenGL, і, хоча це було легко освоїти, важко зрозуміти, як насправді працює OpenGL. Сучасний підхід вимагає від розробника розуміння роботи OpenGL та графічного програмування, і, хоча це трохи складно, це забезпечує набагато більшу гнучкість та ефективність.

Це також одна з причин, чому цей курс більш орієнтований на core-profile в OpenGL версії 3.3.

На сьогоднішній день доступні новіші версії OpenGL (на момент написання - версія 4.6), тому виникає логічне питання: «Чому вивчається OpenGL 3.3, коли вже є OpenGL 4.6?». Справа в тому, що всі наступні версії OpenGL, починаючи з версії 3.3, додають додаткові корисні можливості OpenGL без зміни фундаментального ядра/базису, що використовується в OpenGL; нові версії просто надають кілька ефективніших чи корисних способів вирішення тих самих завдань. У результаті всі концепції та техніки залишаються незмінними при виході нових версій OpenGL, тому вивчення OpenGL 3.3 є цілком справедливим.

***Примітка*:** При використанні функціоналу останніх версій OpenGL тільки найсучасніші відеокарти зможуть запустити ваш додаток. Саме тому більшість розробників зазвичай орієнтуються на більш ранні версії OpenGL і тільки при необхідності підключають функціонал новіших версій.

Розширення в OpenGL

Відмінною особливістю OpenGL є підтримка розширень. Щоразу, коли графічна компанія викочує нову методику або нову велику оптимізацію для рендерингу, часто зустрічається в розширенні, реалізованому в драйверах. Якщо обладнання, на якому працює програма, підтримує таке розширення, то розробник може використовувати функціонал, що надається цим розширенням, для більш просунутої або ефективної графіки. Таким чином, графічний розробник вже може використовувати нові методи рендерингу, просто перевіряючи, чи підтримується дане розширення відеокартою, при цьому не чекаючи поки OpenGL додасть цей функціонал у свою нову версію. Часто, коли розширення є популярним чи дуже корисним, воно зрештою стає частиною нової версії OpenGL.

Розробник повинен знати, чи доступні будь-які з цих розширень, перш ніж їх використовувати (або використовувати бібліотеку розширень OpenGL). Це дозволяє розробнику робити речі краще чи ефективніше в залежності від того, чи доступне розширення:

```
1 if(GL_ARB_extension_name)
2 {
3 // Робимо круті та сучасні речі, що підтримуються залізом
4 }
5 інше
6 {
7 // Розширення не підтримуються - робимо все "по-старому"
8 }
```

В OpenGL версії 3.3 рідко будуть потрібні розширення для більшості концепцій, які будуть розглянуті, але там, де це необхідно, будуть надані відповідні інструкції.

Стани в OpenGL

OpenGL сам по собі є великою *системою станів*, яка містить цілий набір змінних, що визначають те, як OpenGL повинен в даний момент виконувати операції. Стан OpenGL зазвичай називають **контекстом OpenGL**. При використанні OpenGL часто змінюють його стан, встановлюючи деякі параметри, маніпулюючи значеннями з буфера, а потім виконуючи рендеринг з використанням поточного контексту.

Наприклад, якщо зараз OpenGL малює трикутники, а потрібно лінії, то змінюємо стан OpenGL, переглядаючи деяку змінну контексту, яка встановлює спосіб малювання OpenGL. Як тільки буде змінено контекст, повідомивши OpenGL, що він повинен малювати лінії замість трикутників, всі наступні команди малювання будуть вже автоматично малювати лінії (а не трикутники).

Під час роботи з OpenGL часто зустрічаються з певними **функціями зміни стану**, які змінюють контекст, та певними **функціями використання стану**, які виконують зазначені операції, залежно від стану OpenGL.

Об'єкти в OpenGL

Бібліотеки OpenGL написані мовою програмування Сі, але при цьому допускають безліч відгалужень з використанням інших мов програмування. Оскільки багато мовних конструкцій мови Сі не дуже добре перекладаються в інші високорівневі мови, то OpenGL був розроблений з урахуванням кількох абстракцій. Однією із таких абстракцій є об'єкти.

Об'єкт у OpenGL - це набір параметрів, що представляє підмножина стану OpenGL. Наприклад, може бути об'єкт, який представляє налаштування вікна малювання; тоді можливо б змінити його розмір, кількість кольорів, що підтримуються, і так далі. Об'єкт у OpenGL можна собі візуалізувати як звичайну **структуру**:

```
1  struct object_name {
2  float option1;
3  int option2;
4  char[] ім'я;
5  };
```

Щоразу, коли використовуються об'єкти, вони зазвичай виглядають наступним чином (з візуалізацією контексту OpenGL як великої структури):

```
1 // Стан OpenGL
2 struct OpenGL_Context {
3          ...
4          object_name* object_Window_Target;
5          ...
6 };
```

Потім:

```
1  // Створюємо об'єкт
2  unsigned int objectId = 0;
3  glGenObject(1, &objectId);
4
```

```
// Зв'язуємо (привласнюємо) об'єкт із контекстом
glBindObject(GL_WINDOW_TARGET, objectId);

// Встановлюємо параметри об'єкта, який зараз пов'язаний із GL_WINDOW_TARGET
glSetObjectOption(GL_WINDOW_TARGET, GL_OPTION_WINDOW_WIDTH,  800);
glSetObjectOption(GL_WINDOW_TARGET, GL_OPTION_WINDOW_HEIGHT, 600);

// Повертаємо цільовий контекст (перехід до стану за умовчанням)
glBindObject(GL_WINDOW_TARGET, 0);
```

Цей невеликий фрагмент коду є робочим процесом, який часто спостерігається при роботі з OpenGL. Спочатку створюємо об'єкт і зберігаємо посилання на нього як ідентифікатор (дані реального об'єкта зберігаються «за лаштунками»). Потім пов'язуємо об'єкт (використовуючи його ідентифікатор) з цільовою локацією контексту (місце розташування цільового об'єкта вікна-прикладу визначається як `GL_WINDOW_TARGET`). Потім встановлюємо параметри вікна і, нарешті, від'єднуємо об'єкт, встановлюючи поточний ідентифікатор об'єкта цільового вікна, що дорівнює `0`. Встановлені параметри зберігаються в об'єкті, на який посилається `objectId`, і відновлюються, як тільки ми назад зв'язуємо об'єкт з `GL_WINDOW_TARGET`.

Перевагою використання об'єктів є те, що можна визначити більше одного об'єкта в нашому додатку, встановити їх параметри, і щоразу, коли запускаємо операцію, яка використовує стан OpenGL, то пов'язуємо об'єкт з нашими кращими налаштуваннями. Наприклад, є об'єкти-контейнери для зберігання даних 3D-моделі (`Дім` або `Персонаж`), і кожного разу, коли хочемо намалювати цю 3D-модель, то пов'язуємо з необхідним контекстом об'єкт, що містить дані моделі, яку хочемо намалювати (перед цим створивши та встановивши параметри для цих об'єктів). Наявність кількох об'єктів дозволяє нам вказувати безліч моделей, і щоразу, коли хочемо намалювати конкретну модель, перед малюванням просто зв'язуємо відповідний об'єкт (не встановлюючи йому всі параметри з нуля) з необхідним контекстом.

Заняття №2. Підготовка до першого проекту OpenGL: налаштування GLFW, та GLAD

Перше, що потрібно зробити, перш ніж почнемо програмувати графіку — це створити **Контекст OpenGL** та вікно програми для малювання. Однак, дані операції є специфічними для кожної операційної системи, тому OpenGL цілеспрямовано намагається абстрагуватися від них. Це означає, що ми самостійно повинні створити вікно, визначити контекст і обробляти введення користувача.

На щастя, існує досить багато бібліотек, які забезпечують необхідний нам функціонал, деякі з них спеціально націлені на роботу з OpenGL. Ці бібліотеки позбавлять нас всієї специфічної роботи, пов'язаної з особливостями конкретної операційної системи. Найбільш популярними бібліотеками є: **GLUT, SDL,SFML**та **GLFW**. В курсі будемо використовувати бібліотеку GLFW. Можете використовувати будь-яку іншу бібліотеку з

цього списку,. встановлення та налаштування для більшості з них аналогічне установці та налаштуванню GLFW.

Бібліотека GLFW

GLFW (англ. *«**G**raphics **L**ibrary **F**rame**W**ork»*) - це бібліотека, написана мовою Сі, спеціально призначена для роботи з OpenGL. Бібліотека GLFW надає всі необхідні інструменти, які будуть потрібні для рендерингу на екран різних об'єктів. Завдяки цьому можна створювати контексти OpenGL, визначати параметри вікна і обробляти введення користувача, що цілком корелює з нашими цілями.

Основна увага на цьому та наступному занятті приділяється вивченню бібліотеки GLFW, створенню коректного контексту OpenGL, а також простого вікна, в якому можна малювати наші об'єкти. Покроково розглянемо встановлення бібліотеки GLFW, а також процес складання та компіляції програми у зв'язці з GLFW.

Примітка: На момент написання матеріалу як середовище розробки будемо використовувати **Microsoft Visual Studio 2019** Якщо ж використовується старіша чи новіша версія Visual Studio (або взагалі інше середовище розробки), процес встановлення та налаштування GLFW аналогічний у більшості IDE.

Складання GLFW

Бібліотеку GLFW можна завантажити з **офіційного сайту**. Варто зазначити, що GLFW вже має попередньо скомпільовані бінарні та заголовні файли для Visual Studio 2010-2019.

Компіляція проекту

Після того, як ми створили чи скачати бібліотеку, потрібно переконатися, що IDE знає, де знайти даний файл, а також інші файли нашої програми OpenGL, що підключаються. Вирішити це питання можна двома способами:

Спосіб №1: Можна скопіювати вміст папки `include` GLFW у відповідну папку `include` вашої IDE або компілятора, а також скопіювати отриманий файл `glfw3.lib` у відповідну папку `/lib` вашого IDE або компілятора. Даний спосіб цілком робочий, але не рекомендується, тому що нова установка IDE або компілятора призведе до того, що вам доведеться підключати необхідні файли.

Спосіб №2: Можна створити нову папку, яка міститиме всі заголовні файли та файли зі сторонніх бібліотек. На цю папку можна посилатися зі своєї IDE або компілятора.

Наприклад, можна створити папку, в якій будуть перебувати папки `Lib` і `include`. У них будемо зберігати всі наші бібліотечні та файли, що підключаються, які будемовикористовувати для наших OpenGL-проектів. Виходить, що всі сторонні

бібліотеки будуть організовані в одному місці (і їх можна буде спільно використовувати на кількох комп'ютерах). Однак, кожного разу, коли створюєте новий проект, повинні вказувати для IDE відповідні шляхи до цих папок.

Як тільки необхідні файли будуть збережені у вибраному місці, можна розпочати створення першого OpenGL-GLFW-проекту.

Перший проект

Для початку відкриємо Visual Studio і створимо новий проект. Для цього потрібно вибрати тип проекту`"C++"`, а далі -`"Порожній проект"`(Не забудьте дати проекту відповідне ім'я).

Тепер є робочий простір для створення нашого першого OpenGL-додатку.

Лінкінг проекту

Щоб проект міг використовувати GLFW, потрібно зв'язати з ним отриману бібліотеку. Це можна зробити, вказавши в налаштуваннях лінкера, що хочемо використати бібліотеку `glfw3.lib`, але проект поки що не знає де її шукати, т.я. всі подібні файли ми перемістили до іншої папки. Таким чином, спочатку ми повинні додати цю папку до нашого проекту.

Для цього натисніть правою кнопкою мишки на ім'я проекту`"Обозреватель Решений" > "Властивості"`. У вікні виберіть`"Каталоги VC++" > "Каталоги бібліотек"`:

Тут можна додати власні каталоги, щоб проект знав, де шукати необхідні файли. Це можна зробити, вставивши вручну шлях до каталогу або клацнувши по відповідному рядку та вибравши пункт`<Змінити...>`.

Можна додати стільки додаткових каталогів, скільки необхідно, і з цього моменту IDE при пошуку файлів бібліотек також переглядатиме ці директорії. Тому, як тільки підключите папку `Lib` з проекту GLFW, ви зможете використовувати всі файли бібліотек із цієї папки. Аналогічно і з додаванням папки `include` для заголовних файлів.

Оскільки для VS були вказані всі необхідні файли, то нарешті можна зв'язати GLFW з нашим проектом, перейшовши до розділу`"Компонувальник" > "Введення"`:

Щоб зв'язати бібліотеку, потрібно вказати її ім'я для компонувальника. Оскільки бібліотека називається `glfw3.lib`, то додаємо назву цього файлу в розділ`"Додаткові залежності"`(вручну або через пункт`<Змінити...>`), і з цього моменту під час запуску процесу компіляції GLFW буде пов'язаний з нашим проектом. На додаток до GLFW ми також повинні додати посилання на бібліотеку OpenGL, але ці дії будуть відрізнятися, залежно від (вашої) операційної системи:

Бібліотека OpenGL у Windows. Якщо використовуєте операційну систему Windows, то необхідний файл бібліотеки `OpenGL32.Lib`, що входить до пакета Microsoft SDK, вже є у складі Visual Studio і не потребує окремої інсталяції. Оскільки ми використовуємо компілятор VS і працюємо в операційній системі Windows, все, що вам потрібно зробити - це додати назву файлу `OpenGL32.Lib` до загального списку параметрів компонувальника.

Примітка: У примітці підрозділу *«Складання GLFW»* сказано, що ми збираємо усі бібліотеки у форматі 64-бітних бінарних файлів. Водночас число*«32»*у назві файлу *OpenGL**32**.Lib* може викликати деяку плутанину, ніби натякаючи те що, у разі використовується саме 32-бітної версія бібліотеки. Насправді, це не так. Якщо у вас встановлений пакет Microsoft SDK, досить просто зайти в папку *C:\Program Files (x86)\Windows Kits\10\Lib\[Номер_версії_SDK]\um**x64***\ та переконатися, що в ній серед інших файлів знаходиться і наша бібліотека:

Більше того, якщо зайти до папки*C:\Program Files (x86)\Windows Kits\10\Lib\[Номер_версії_SDK]\um**x86***\, то й там можна зустріти файл*OpenGL**32**.Lib*:

Іншими словами, під однією назвою файлу *OpenGL32.Lib* існує дві різні версії (32-бітна та 64-бітна) бібліотеки. Тому просто не звертайте увагу на те, що в її імені є частина *«32»*. Все чудово працює і у 64-бітному оточенні.

Бібліотека OpenGL у Linux. Якщо ви працюєте в операційній системі **Linux**, то вам потрібно підключити бібліотеку `libGL.so` за допомогою ключів `-lGL`, що додаються до параметрів компонувальника. Якщо ви не можете знайти цю бібліотеку, вам, ймовірно, необхідно встановити будь-який з пакетів: Mesa, NVidia або AMD dev.

Потім, після додавання бібліотек GLFW і OpenGL до налаштувань компонувальника, ви зможете підключити заголовні файли GLFW наступним рядком коду:

```
1 #include <GLFW\glfw3.h>
```

***Примітка*:** Для користувачів Linux, які використовують компілятор GCC, компілювати проект допоможуть наступні параметри командного рядка:

```
-lglfw3 -lGL -lX11 -lpthread -lXrandr -lXi -ldl
```

Неправильне лінкування відповідних бібліотек призведе до помилок. Будьте уважні під час копіювання даних параметрів, оскільки знак `-` перед зазначеними параметрами може бути скопійовано некоректно, і ви отримаєте помилку. Краще вручну пропишіть цей код.

На цьому встановлення та налаштування GLFW завершено.

GLAD

Оскільки OpenGL насправді є лише стандартом/специфікацією, виробник драйверів повинен реалізувати специфікацію для драйвера, що підтримується конкретною відеокартою. Через те, що існує безліч різних версій драйверів OpenGL, розташування більшості OpenGL-функцій під час компіляції невідоме і має бути запитане під час

виконання програми. Завдання розробника полягає в тому, щоб отримати розташування потрібних йому функцій та зберегти їх у вигляді **покажчиків на функції** для подальшого використання. Отримання місць розташування цих функцій залежить від **конкретної операційної системи**.

У Windows це виглядає приблизно так:

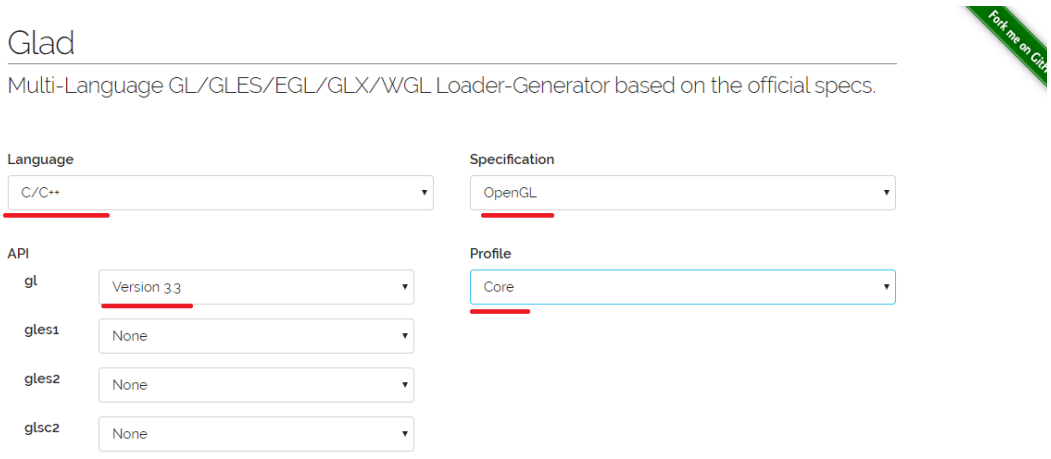
```
// Визначення прототипу функцій
typedef void (*GL_GENBUFFERS) (GLsizei, GLuint*);

// Пошук функції та її надання покажчику на функцію
GL_GENBUFFERS glGenBuffers  =
(GL_GENBUFFERS)wglGetProcAddress("glGenBuffers");

// Тепер функцією можна користуватися, як завжди
unsigned int буфер;
glGenBuffers(1, &buffer);
```

Як ви напевно могли помітити, цей код виглядає дещо складним, до того ж, розробникам доведеться щоразу проходити через цей громіздкий процес, здійснюючи дані дії для кожної функції, яка може знадобитися і яка ще не була оголошена. На щастя, є рішення бібліотека GLAD.

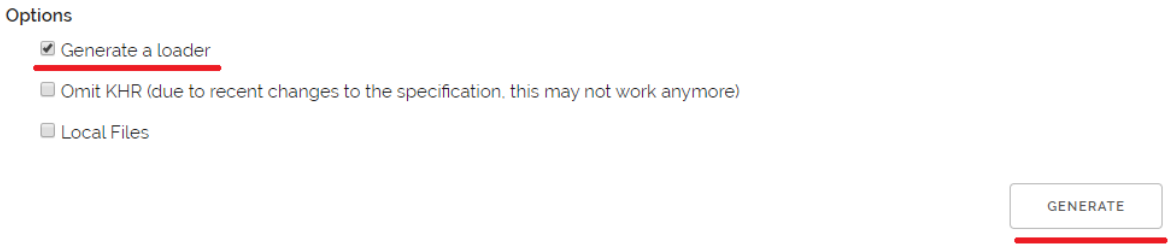
Встановлення GLAD

GLAD- це бібліотека з **відкритим вихідним кодом**, яка керує усією тією громіздкою роботою, про яку ми говорили вище. GLAD має дещо інше налаштування конфігурації, ніж більшість поширених бібліотек з відкритим вихідним кодом. Вона використовує веб-сервіс, де ми можемо повідомити GLAD, для якої версії OpenGL ми хотіли б визначити та завантажити усі відповідні функції OpenGL.

Перейдіть до **веб-сервіс** GLAD, переконайтеся, що в полі*"Мова"*обрана мова`"C/C++"`, та у розділі *«API»* виберіть версію OpenGL `3.3`(саме її ми і будемо використовувати; хоча нові версії також підійдуть). Крім того, переконайтеся, що в полі *«Профіль»* встановлено`"Core"`:



Також параметр `«Generate a loader»` має бути відзначений галочкою. Пункт *«Extensions»* ми поки що пропустимо, залишається натиснути кнопку `"Generate"` щоб створити потрібні нам файли бібліотеки:



До цього моменту GLAD надасть вам можливість завантажити zip-архів, що містить в собі дві папки, що підключаються, і файл `glad.c`. Потрібно скопіювати обидві ці папки (`glad` и `KHR`) у свою папку з файлами, що підключаються (або додайте додатковий елемент, що вказує на ці папки у властивостях проекту), а також додати файл `glad.c` у свій проект.

Після виконання цих кроків ви зможете використовувати наступну директиву include:

```
1 #include <glad/glad.h>
```

Тепер при спробі скомпілювати проект у вас не повинно виникати жодних помилок.

На наступному уроці ми розглянемо використання GLFW, налаштування контексту OpenGL та створення вікна. Обов'язково переконайтеся, що всі ваші `include-` та `library-` каталоги є коректними, і що імена бібліотек у налаштуваннях компонувальника відповідають зазначеним бібліотекам.

Додаткові ресурси

GLFW: віконний довідник— офіційне керівництво GLFW з налаштування та конфігурації вікна GLFW.

Створення додатків— надає відмінну інформацію про процес компіляції/лінкінгу вашої програми та великий список можливих помилок (+ рішення), які можуть виникнути.

GLFW з Code::Blocks- Розповідає про використання GLFW з Code::Blocks.

Використання CMake— короткий огляд того, як запустити CMake у Windows та Linux.

Polytonic/Glitter- Простий шаблонний проект, який попередньо налаштований з усіма відповідними бібліотеками; відмінно підійде, якщо вам потрібен приклад того, що можна зробити за допомогою OpenGL без необхідності компілювати всі бібліотеки самостійно.

Заняття №3. Перший проект у OpenGL — Створення вікна

Підключаємо GLFW

Давайте подивимося, чи зможемо ми запуститии **GLFW**. Для початку створимо основний файл нашої програми, який буде називатися main.cpp і підключимо в ньому два заголовні файли:

```
1 #include <glad/glad.h>
2 #include <GLFW/glfw3.h>
```

***Важлива примітка*:** Заголовковий файл GLAD обов'язково повинен підключатися перед іншими заголовними файлами, які вимагають OpenGL (наприклад, GLFW), оскільки всередині GLAD відбувається підключення файлів OpenGL (наприклад,`GL/gl.h`).

Тепер нам потрібно визначити функцію main(), в якій створюватиметься екземпляр вікна GLFW:

```
 1 int main()
 2 {
 3 glfwInit();
 4 glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
 5 glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
 6 glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
 7
 8 // Розкоментуйте рядок нижче, якщо ви використовуєте macOS
 9 // glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
10
11 return 0;
12}
```

На початку функції main() ми ініціалізуємо GLFW за допомогою функції glfwInit(). Після цього, використовуючи функцію glfwWindowHint(), ми задаємо конфігурацію GLFW:

перший аргумент функції glfwWindowHint() вказує на те, який з параметрів GLFW ми хочемо налаштувати, а всі разом вони утворюють безліч опцій, відмінною рисою яких є наявність префікса `GLFW_`;

Другий аргумент - це ціле число, яке встановлює значення нашого параметра. Зі списком усіх можливих опцій та відповідних їм значень можна ознайомитись **тут**.

Зараз ви можете спробувати скомпілювати та запустити цю програму. Якщо компілятор видасть купу помилок типу *невизначене посилання* Це означає, що ви неправильно підключили бібліотеку GLFW.

Оскільки основна увага на даному уроці приділяється OpenGL версії 3.3, за допомогою перших двох викликів функції glfwWindowHint() ми повідомляємо GLFW, що збираємося використовувати саме цю версію OpenGL. Таким чином, GLFW зможе правильно

організувати свою роботу під час створення контексту OpenGL. У випадку, якщо на комп'ютері користувача не встановлено потрібну версію OpenGL, GLFW не запуститься. Ми також повідомляємо GLFW, що хочемо явно використовувати **ядро-профіль**. Це робиться для того, щоб не підвантажувати зайві для нас OpenGL функції, призначені для підтримки зворотної сумісності додатків.

Зверніть увагу, якщо ви користувач macOS, вам необхідно розкоментувати наступний рядок у вашій програмі:

```
1 // glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
```

***Примітка*:** Переконайтеся, що ОС та апаратна частина комп'ютера підтримують версію OpenGL 3.3 і вище. В іншому випадку, додаток може демонструвати невизначену поведінку або взагалі перестати працювати. Визначити, яку версію OpenGL підтримує ваш комп'ютер, можна за допомогою програми **Переглядач розширень OpenGL** (для Windows) або команди `glxinfo` (Для Linux). Якщо в результаті цього з'ясується, що ваша версія OpenGL нижче рекомендованої, спробуйте перевірити, чи підтримує ваша відеокарта OpenGL 3.3+ (якщо ні, то вона дійсно стара) і/або оновіть драйвери.

Далі ми маємо створити об'єкт `window` представляє вікно програми. Присутність цього об'єкта потрібна більшість інших функцій GLFW, саме через нього відбувається керування вікном нашої програми:

```
1 GLFWwindow* window = glfwCreateWindow(800, 600, " OpenGL for test", NULL,
2 NULL);
3 if (window == NULL)
4 {
5     std::cout << "Failed to create GLFW window" << std::endl;
6     glfwTerminate();
7     return -1;
8 }
glfwMakeContextCurrent(window);
```

Розглянемо детально функцію glfwCreateWindow():

перші два аргументи, які вона приймає, є шириною та висотою вікна;

за допомогою третього аргументу ми вказуємо ім'я вікна.`«OpenGL for test»` (Ви можете встановити будь-яке інше ім'я);

останні 2 параметри поки що можна проігнорувати.

Ця функція повертає об'єкт `GLFWwindow`, Що пізніше знадобиться нам для інших операцій GLFW.

Після цього за допомогою функції `glfwMakeContextCurrent(window)`, ми повідомляємо GLFW зробити контекст нашого вікна основним контекстом у поточному потоці.

GLAD

На попередньому занятті ми згадували, що GLAD оперує **вказівниками** на OpenGL-функції, тому ми повинні спочатку ініціалізувати GLAD, і тільки після цього можна користуватися OpenGL-функціями:

```
1 if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
2 {
3     std::cout << "Failed to initialize GLAD" << std::endl;
4     return -1;
5 }
```

Як параметр ми передаємо GLAD-функцію, яка завантажує адреси покажчиків на OpenGL-функції (які можуть відрізнятися, залежно від ОС).

У той же час бібліотека GLFW містить зручний інструмент у вигляді функції glfwGetProcAddress(), яка самостійно може визначити потрібні нам для роботи функції, залежно від операційної системи, що використовується.

Вікно перегляду

Перш ніж зможемо почати рендеринг, нам потрібно повідомити OpenGL розмір видимої області вікна, через яку користувач зможе спостерігати за процесом рендерингу та відображення картинки. Ця область називається **вікном перегляду** (англ. ***«viewport»***). Його розміри задаються щодо основного вікна за допомогою функції glViewport():

```
1 glViewport(0, 0, 800, 600);
```

Першими двома параметрами цієї функції є координати лівого нижнього кута вікна перегляду. Третій та четвертий параметри встановлюють ширину та висоту вікна для рендерингу в пікселях, які ми встановлюємо рівними розміру GLFW-вікна: ширина вікна становить 800 пікселів, а висота – 600 пікселів.

Варто зауважити, що ми могли б встановити менші значення розмірів вікна перегляду щодо розмірів вікна GLFW; у такому випадку весь OpenGL-рендеринг відображався б у меншому вікні, і ми могли б, наприклад, відображати інші елементи поза вікном перегляду OpenGL.

***Примітка*:** Насправді OpenGL використовує дані з функції glViewport(), щоб перетворити оброблені ним 2D-координати на координати на вашому екрані. Наприклад, оброблена точка з координатами `(-0,5; 0,5)` буде (після фінального перетворення) відображена у крапку з координатами `(200; 450)` на екрані. Зверніть увагу, що оброблені координати в OpenGL знаходяться в діапазоні від `-1` до `1` тому ми відображаємо діапазон оброблених координат `(-1, 1)` на відповідному діапазоні координат `(0, 800)` і `(0, 600)` на екрані.

Однак у той момент, коли користувач змінює розміри вікна, повинен бути скоригований розмір області вікна перегляду.

Для цього необхідно визначити **callback-функцію** (або ***"функцію зворотного виклику"***), яка викликається при кожній зміні розміру вікна. Її прототип показаний нижче:

```
1 void framebuffer_size_callback(GLFWwindow* window, int width, int height);
```

Як перший аргумент дана функція приймає покажчик на об'єкт `GLFWwindow`, а двома наступними аргументами є нові розміри області вікна перегляду. Таким чином, щоразу, коли змінюється розмір вікна програми, GLFW викликає цю функцію, передаючи їй всі необхідні для обробки аргументи:

```
1 void framebuffer_size_callback(GLFWwindow* window, int width, int height)
2 {
3     glViewport(0, 0, width, height);
4 }
```

Для того, щоб повідомити GLFW, що ми хочемо викликати функцію framebuffer_size_callback() щоразу, коли відбувається зміна розмірів вікна, нам потрібно прописати наступне:

```
1 glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
```

Під час першого відображення вікна викликається функція framebuffer_size_callback(), що має аналогічні розміри отриманого вікна. Варто зауважити, що для Retina-дисплеїв ширина і висота, зрештою, будуть значно більшими за вихідні вхідні значення.

Існує безліч callback-функцій, які ми можемо налаштувати для реєстрації наших власних функцій. Наприклад, ми можемо визначити функцію callback для обробки змін вхідних даних джойстика, обробки повідомлень про помилки і т.д. Реєстрація callback-функцій відбувається після створення вікна та до початку циклу рендерингу.

Цикл візуалізації (рендерингу)

Ми не хочемо, щоб наш додаток намалював одну картинку, а потім одразу ж зачинив вікно. Ми хочемо, щоб наш додаток продовжував малювати картинки і обробляти введення користувача до тих пір, поки ми йому явно не повідомимо зупинитися. Для цього нам потрібно створити **цикл while**, іменований у подальшому **циклом рендерингу**, який буде працювати до тих пір, поки ми самі не повідомимо GLFW зупинитися.

У наступному фрагменті коду показаний приклад досить простого циклу рендерингу:

```
1 while(!glfwWindowShouldClose(window))
2 {
3     glfwSwapBuffers(window);
4     glfwPollEvents();
5 }
```

На початку кожної ітерації циклу функція glfwWindowShouldClose() перевіряє, чи повідомляли ми GLFW закрити програму. Якщо так, то функція повертає`true`, та

ігровий цикл зупиняється, після чого ми можемо завершити виконання нашої програми. Функція glfwPollEvents() стежить за тим, чи ініціюються якісь події (наприклад, введення з клавіатури або переміщення мишки), оновлює стан вікна та викликає відповідні функції (які ми можемо зареєструвати за допомогою callback-методів). Функція glfwSwapBuffers() змінює місцями колірний буфер (великий 2D-буфер, що містить значення кольору для кожного пікселя у вікні GLFW), який використовується для рендерингу під час цієї ітерації рендерингу, і виводить його на екран.

***Примітка*:** Коли програма виконує малювання сцени з використанням одного єдиного буфера, то може виникнути проблема у вигляді мерехтливого зображення. Це відбувається тому, що підсумкове зображення не створюється миттєво, а малюється піксель за пікселем зліва направо і зверху вниз. Оскільки це зображення не з'являється в одну мить для користувача, результат може містити **артефакти стиснення** (Глюки, спотворення і т.д.). Щоб позбавитися даних проблем, віконні програми використовують **технологію подвійного буфера**: front-буфер містить підсумкове вихідне зображення, яке користувач бачить на своєму екрані, у той час як усі команди малювання/рендерінг виконуються в back-буфері. Як тільки всі команди рендерингу закінчать свою роботу, ми змінюємо вміст front-буфера з вмістом back-буфера, щоб зображення можна було відобразити без виконання його рендерингу, оминаючи вищезазначену проблему появи артефактів.

Останні штрихи

Як тільки ми вийдемо з циклу рендерингу, нам потрібно буде очистити/видалити всі виділені для GLFW ресурси. Це можна зробити за допомогою функції glfwTerminate(), яку необхідно викликати наприкінці функції main():

```
glfwTerminate();
return 0;
```

Завдяки цьому ми очистимо всі задіяні раніше ресурси та коректно завершимо виконання програми.

На даний момент повний вихідний код нашої програми виглядає так:

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>

#include <iostream>

void framebuffer_size_callback(GLFWwindow* window, int width, int height);
void processInput(GLFWwindow *window);

//Константи
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;

int main()
{
// glfw: ініціалізація та конфігурування
glfwInit();
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

```
20
21 // Розкоментуйте цю частину коду, якщо ви використовуєте macOS
22 /*
23 #ifdef __APPLE__
24 glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
25 #endif
26 */
27
28 // glfw: створення вікна
29 GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "OpenGL
30 для тесту", NULL, NULL);
31 if (window == NULL)
32 {
33 std::cout << "Не вдалося створити вікно GLFW" << std::endl;
34 glfwTerminate();
35 return -1;
36 }
37 glfwMakeContextCurrent(window);
38     glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
39
40 // glad: завантаження всіх покажчиків на OpenGL-функції
41 if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
42 {
43 std::cout << "Не вдалося ініціалізувати GLAD" << std::endl;
44 return -1;
45 }
46
47 // Цикл візуалізації
48 while (!glfwWindowShouldClose(window))
49 {
50 // Обробка введення
51 processInput(window);
52
53                     // Виконання рендерингу
54 glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
55 glClear(GL_COLOR_BUFFER_BIT);
56
57 // glfw: обмін вмістом front-і back-буферів. Відстеження подій введення/виводу (чи
58 була натиснута/відпущена кнопка, переміщений курсор миші тощо)
59 glfwSwapBuffers(window);
60 glfwPollEvents();
61 }
62
63 // glfw: завершення, звільнення всіх раніше задіяних GLFW-ресурсів
64 glfwTerminate();
65 return 0;
66 }
67
68 // Обробка всіх подій введення: запит GLFW про натискання/відпускання клавіш на
69 клавіатурі в даному кадрі та відповідна обробка даних подій
70 void processInput(GLFWwindow *window)
71 {
```

```
72 if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
73  glfwSetWindowShouldClose(window, true);
74 }
75
76 // glfw: щоразу, коли змінюються розміри вікна (користувачем або операційною
77 системою), викликається дана callback-функція
78 void framebuffer_size_callback(GLFWwindow* window, int width, int height)
79 {
    // Переконуємось, що вікно перегляду відповідає новим розмірам вікна.
            // Зверніть увагу, висота вікна на Retina-дисплеях буде значно більшою, ніж
    зазначено в програмі
    glViewport(0, 0, width, height);
    }
```

Тепер спробуйте скомпілювати програму. Результат має бути просте вікно чорного кольору.

Якщо у вас вийшло дуже нудне і похмуре чорне зображення, ви все зробили правильно!

Якщо ж виникли проблеми з компіляцією програми, то спочатку переконайтеся, що встановлені всі потрібні параметри компонувальника і що ви правильно підключили потрібні каталоги до своєї IDE (як описано на **попередньому уроці**). Крім того, переконайтеся, що ваш код не містить помилок; ви можете легко перевірити його, порівнявши з повним вихідним кодом, розташованим вище.

Введення користувача

Непогано було б реалізувати можливість обробляти введення користувача (наприклад, натискання клавіш клавіатури, руху курсора миші і т.п.). На щастя, у GLFW вже є кілька функцій, призначених для вирішення подібних завдань. Одна з них - це GLFW-функція glfwGetKey(), що приймає як перший аргумент об'єкт введення користувача (у нашому випадку цим об'єктом буде `window`, тобто. саме вікно), а як другий аргумент — яку клавішу клавіатури потрібно відстежувати. В результаті функція повертає відповідь, чи ця клавіша була натиснута в даний момент. Щоб не допускати безладу в організації вихідного коду нашої програми, ми визначимо нову функцію `processInput(GLFWwindow *window)`, всередину якої і помістимо виклик функції glfwGetKey():

```
1 void processInput(GLFWwindow *window)
2 {
3 if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
4 glfwSetWindowShouldClose(window, true);
5 }
```

Як ви бачите, в ній ми перевіряємо, чи натиснув користувач клавішу *Escape*. У разі позитивної відповіді функція glfwGetKey() повертає значення `GLFW_PRESS` якщо ж натискання не було, функція повертає `GLFW_RELEASE`. Якщо користувач дійсно натиснув клавішу *Escape*, то ми закриваємо GLFW, встановивши за допомогою функції glfwSetwindowShouldClose() значення *true* для `WindowShouldClose`. Завдяки

цьому цикл рендерингу в блоці коду функції main() перерветься, і наша програма закриється.

Функція processInput() буде викликати кожну ітерацію нашого циклу рендерингу:

```
while (!glfwWindowShouldClose(window))
{
    processInput(window);

    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

Завдяки цьому ми можемо легко відстежувати натискання певних клавіш і, отже, реагувати на це належним чином у кожному **фреймі** *(*так найчастіше називається ітерація в циклі рендерингу*)*.

Рендеринг проекту

Всі команди, так чи інакше пов'язані з рендерингом, ми помістимо в цикл рендерингу, щоб вони викликали кожну ітерацію циклу (або, як ми вже казали, кожен фрейм циклу):

```
// Цикл візуалізації
while(!glfwWindowShouldClose(window))
{
// Користувальницьке введення
processInput(window);

// Тут знаходяться команди рендерингу
...

// Перевірка та обробка подій, обмін вмісту буферів
glfwPollEvents();
glfwSwapBuffers(window);
}
```

Щоб перевірити, чи все працює так, як потрібно, ми спробуємо зафарбувати наше вікно в який-небудь довільний колір за допомогою функції очищення екрану glClear(), яку помістимо на початок нашого фрейму. Якщо цю функцію розташувати в іншому місці, можна зіткнутися з негативним ефектом, коли на екрані буде видно результат відображення попереднього кадру.

Викликаючи функцію glClear(), ми хочемо, щоб вона очистила колірний буфер екрану. Використовуючи як аргументи так звані «буферні біти», ми можемо вказати нашій функції glClear() який саме буфер потрібно очистити:

`GL_COLOR_BUFFER_BIT`- Очищення буфера кольору;

`GL_DEPTH_BUFFER_BIT`- Очищення буфера глибини;

`GL_STENCIL_BUFFER_BIT`- Очищення буфера трафарету.

В даний момент нас цікавлять значення кольору, тому ми очищатимемо колірний буфер:

```
1 glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
2 glClear(GL_COLOR_BUFFER_BIT);
```

Зверніть увагу, що ми також задаємо колір для очищення екрану за допомогою функції glClearColor(). Щоразу, коли ми викликаємо glClear() і, тим самим, очищаємо колірний буфер, весь колірний буфер буде заповнений кольором, заданим за допомогою функції glClearColor(). В результаті ми отримаємо темно-зелено-блакитний колір.

***Примітка*:** OpenGL функція glClearColor() відноситься до класу функцій зміни контексту. А функція glClear() – до класу функцій використання контексту. Простими словами — функція glClear() використовує поточні параметри кольору, встановлені за допомогою функції glClearColor().

Результат має бути наступним просте вікно темно зеленого кольору.

Отже, зараз у нас вже все готове для заповнення циклу рендерингу безліччю різних команд, пов'язаних із відображенням зображення на екрані користувача. Цим ми й займемося на наступному занятті.