Лекція 6

Лабораторна робота 6

Системи координат

У минулому розділі ми вивчили, як можна використовувати матриці на користь, трансформуючи всі вершини за допомогою матриць трансформації. OpenGL очікує, що всі вершини, які ми хочемо зробити видимими, будуть в нормалізованих пристройних координатах після кожного запуску вершинного шейдера. Тобто координати x, y і z кожної вершини повинні бути в діапазоні від -1.0 до 1.0; координати поза цим діапазоном не будуть видимими. Зазвичай ми вказуємо координати в діапазоні (або просторі), який ми визначаємо самі, і у вершинному шейдері трансформуємо ці координати в нормалізовані пристройні координати (NDC). Ці NDC потім передаються рабитизатору для перетворення їх у 2D-координати/пікселі на вашому екрані.

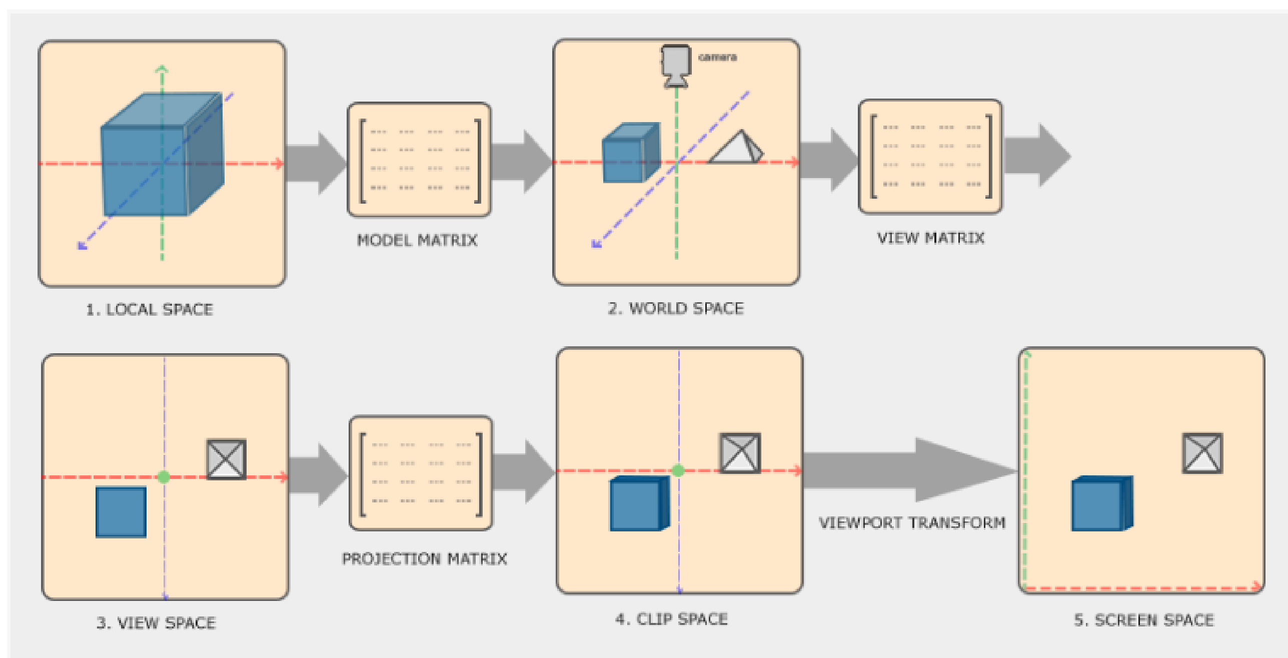
Перетворення координат в NDC зазвичай виконується крок за кроком, де ми трансформуємо вершини об'єкта в різні системи координат, перш ніж остаточно перетворити їх в NDC. Перевага трансформації їх в кілька проміжних систем координат полягає в тому, що деякі операції/розрахунки є простішими в певних системах координат, як це невдовзі стане зрозумілим. Всього існує 5 різних систем координат, які мають для нас значення:

1. Локальна система координат (або об'єктна система)
2. Світова система координат
3. Система координат огляду (або система координат ока)
4. Система обрізання
5. Екранна система координат

Ці системи координат визначають різні стани, в яких наші вершини будуть трансформовані, перш ніж остаточно перетворитися в фрагменти.

Велика картина (Global picture)

Для перетворення координат з одного простору в інший ми використовуватимемо кілька матриць трансформації, з яких найважливішими є матриця моделі, огляду та проекції. Координати наших вершин спочатку починаються в локальній системі як локальні координати, а потім подальше обробляються в координати світу, координати огляду, координати обрізання та, в кінцевому рахунку, перетворюються в екранні координати. Наступне зображення відображає процес і показує, що робить кожне перетворення:



1. Локальні координати - це координати вашого об'єкта відносно його локального початку; це координати, з якими ваш об'єкт починається.

2. Наступним кроком є перетворення локальних координат в координати світового простору, які є координатами відносно більшого світу. Ці координати відносяться до глобального початку світу, разом із багатьма іншими об'єктами, також розташованими відносно цього початку світу.

3. Далі ми перетворюємо світові координати в координати простору огляду так, що кожна координата виглядає з точки зору камери або глядача.
4. Після перетворення координат у просторі огляду ми хочемо проектувати їх в координати обрізання. Координати обрізання обробляються в діапазоні від -1.0 до 1.0 і визначають, які вершини потраплять на екран. Проектування в координати обрізання може додати перспективу при використанні перспективної проекції.

5. І, нарешті, ми перетворюємо координати обрізання в екранні координати в процесі, який ми називаємо "перетворенням відображення", що перетворює координати з -1.0 і 1.0 в діапазон координат, визначений glViewport. Отримані координати потім відсилаються на рабітізатор для перетворення їх в фрагменти.

Мабуть, ви вже отримали загальне уявлення, для чого використовуються кожен окремий простір. Причина, чому ми перетворюємо наші вершини в усі ці різні простори, полягає в тому, що деякі операції роблять більше сенсу або є легше використовувати в певних системах координат. Наприклад, при модифікації вашого об'єкта найлогічніше це робити в локальному просторі, тоді як обчислення деяких операцій щодо положення інших об'єктів має найбільший сенс в глобальних координатах і так далі. Якщо бажаємо, ми могли б визначити одну матрицю перетворення, яка йде від локального простору до простору обрізання одним рухом, але це призводить до меншої гнучкості.

Ми розглянемо кожну систему координат більш детально нижче.

Локальний простір

Простір об'єкта (Local space)

Локальний простір - це простір координат, який є локальним для вашого об'єкта, тобто там, де починається ваш об'єкт. Уявіть собі, що ви створили свій куб в програмі моделювання (наприклад, у Blender). Вихідна точка вашого куба, ймовірно, розташована в (0,0,0), навіть якщо ваш куб може потрапити в інше місце у вашому кінцевому застосуванні. Ймовірно, всі моделі, які ви створили, мають (0,0,0) як їхнє початкове положення. Таким чином, всі вершини вашої моделі є локальними координатами: вони всі місцеві для вашого об'єкта.

Вершини контейнера, якими ми користувалися, були вказані як координати від -0,5 до 0,5 з нулем як його початком. Це локальні координати.

Простір світу (World space)

Якщо ми імпортуємо всі наші об'єкти безпосередньо в програму, вони, ймовірно, всі розташовані десь всередині один в одному в світовому початку (0,0,0), що не є тим, що нам потрібно. Ми хочемо визначити положення для кожного об'єкта, щоб розташувати їх всередині більшого світу. Координати в просторі світу - це саме те, на що вони схожі: координати всіх ваших вершин відносно (ігрового) світу. Це простір координат, в якому ви хочете, щоб ваші об'єкти перетворювались так, щоб вони всі розкидані по місцю (ідеально - реалістично). Координати вашого об'єкта

перетворюються з локальних в світовий простір; це досягається за допомогою матриці моделі.

Матриця моделі - це матриця перетворення, яка переміщає, масштабує і / або обертає ваш об'єкт, щоб розмістити його в світі в певному місці / орієнтації. Подумайте про це, як про те, як трансформувати будинок, зменшивши його (він був трошки занадто великим у локальному просторі), переміщуючи його в пригородне місто і трошки повертаючи його ліворуч навколо вісі y, щоб він красиво вписувався в сусідні будинки. Ви можете розглядати матрицю у попередньому розділі для розташування контейнера по всій сцені як своєрідну матрицю моделі; ми трансформували локальні координати контейнера в яке-небудь інше місце в сцені / світі.

Простір огляду (View space)

Простір перегляду - це те, що люди зазвичай називають камерою OpenGL (іноді його також відомо як простір камери чи простір ока). Простір перегляду - це результат перетворення ваших координат з простору світу в координати, які знаходяться перед точкою огляду користувача. Простір перегляду - це отже простір, як бачить його камера. Це, як правило, досягається за допомогою комбінації переміщень і обертань для того, щоб перенести / обернути сцену так, що певні предмети перетворюються перед камерою. Ці комбіновані перетворення, як правило, зберігаються всередині матриці виду, яка перетворює світові координати в простір огляду. У наступному розділі ми розглянемо, як створити таку матрицю виду для моделювання камери.

Простір обрізки (Clip space)

В кінці кожного запуску вершинного сенсору OpenGL очікує, що координати будуть в певному діапазоні, і будь-які координати, які виходять за цей діапазон, вирізаються. Координати, які вирізаються, відкидаються, тому залишаються лише координати, які потрапляють на ваш екран. Також відсюди походить назва "простір обрізки".

Тому що вказати всі видимі координати в діапазоні від -1.0 до 1.0 не є дуже інтуїтивним, ми визначаємо свій власний набір координат, в якому працюємо, і конвертуємо їх назад у NDC так, як очікує OpenGL.

Щоб перетворити координати вершин з виду в простір обрізки, ми визначаємо так звану матрицю проекції, яка вказує діапазон координат, наприклад, від -1000 до 1000 в кожному напрямку. Матриця проекції потім перетворює координати

всередині цього вказаного діапазону в нормалізовані координати пристрою (-1.0, 1.0) (не безпосередньо, тут також є етап, який називається перспективним діленням). Всі координати поза цим діапазоном не будуть відображатися між -1,0 і 1,0 і, таким чином, вони вирізаються. З цим діапазоном, який ми вказали в матриці проекції, координата (1250, 500, 750) не була б видимою, оскільки координата x знаходиться за межами діапазону і, отже, перетворюється в координату, що перевищує 1,0 в NDC і, отже, вирізається.

Зверніть увагу, що якщо тільки частина примітиву, наприклад, трикутника, знаходиться за межами об'єму обрізки, OpenGL відновить трикутник або кілька трикутників так, щоб вони вписувались у межі обрізки.

Ця прямокутна коробка перегляду, яку створює матриця проекції, називається frustum і кожна координата, яка опиняється всередині цієї frustum коробки, потрапить на екран користувача. Загальний процес перетворення координат в межах визначеного діапазону в NDC, які легко можна відобразити на 2D координати простору перегляду, називається проекцією, оскільки матриця проекції проектує 3D-координати на легкі для відображення 2D нормалізовані пристрої координат.

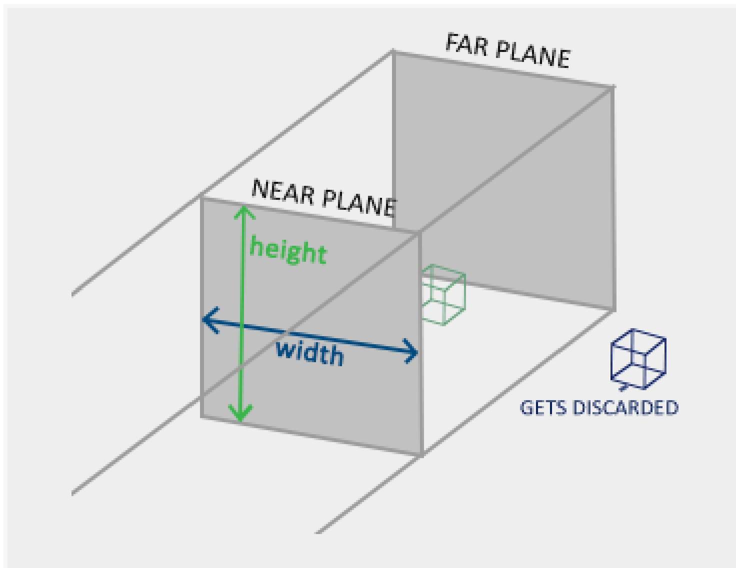
Щойно всі вершини перетворено в простір обрізки, виконується остання операція, яку називають перспективним поділом, де ми ділимо x, y та z компоненти векторів положення на гомогенну компоненту w вектора; перспективний поділ - це те, що перетворює координати 4D простору обрізки в 3D нормалізовані пристрої координати. Цей крок виконується автоматично в кінці кроку вершинного шейдера.

Це після цього етапу, координати перетворені в координати екрану (з використанням налаштувань glViewport) і перетворені в фрагменти.

Матриця проекції для перетворення координат перегляду в координати обрізки, як правило, має дві різні форми, при цьому кожна форма визначає власну унікальну frustum коробку. Ми можемо створити матрицю ортографічної проекції або матрицю перспективної проекції.

Ортографічна проекція

Матриця ортографічної проекції визначає коробку форми куба, яка визначає область обрізки, де кожна вершина поза цією коробкою обрізається. При створенні матриці ортографічної проекції ми вказуємо ширину, висоту та довжину видимої frustum коробки. Всі координати всередині цієї frustum коробки потраплять в діапазон NDC після трансформації її матрицею і, отже, не будуть вирізані. frustum коробка трохи схожа на контейнер:



Ця діаграма ілюструє, як матриця ортографічної проекції створює frustum коробку, яка обмежує видимий об'єкт. У frustum коробці всі видимі об'єкти знаходяться в однакових границях, незалежно від їхнього положення у світі.

фрустум визначає видимі координати і вказується шириною, висотою та близьким і далеким планом. Будь-яка координата перед близьким планом відсікається, і те саме стосується координат за далеким планом. Ортографічний фрустум безпосередньо відображає всі координати всередині фрустуму на нормалізовані пристрої координат без будь-яких спеціальних побічних ефектів, оскільки він не торкатиметься компонента w трансформованого вектора; якщо компонент w залишається рівним 1.0, поділ перспектив не змінить координати.

Для створення ортографічної матриці проекції ми використовуємо вбудовану функцію glm::ortho в GLM:

```
glm::mat4 projection = glm::ortho(left, right, bottom, top, near, far);
```

де `left`, `right`, `bottom`, `top` - це границі фрустуму, а `near` та `far` - це близький та далекий плани відсічення відповідно.

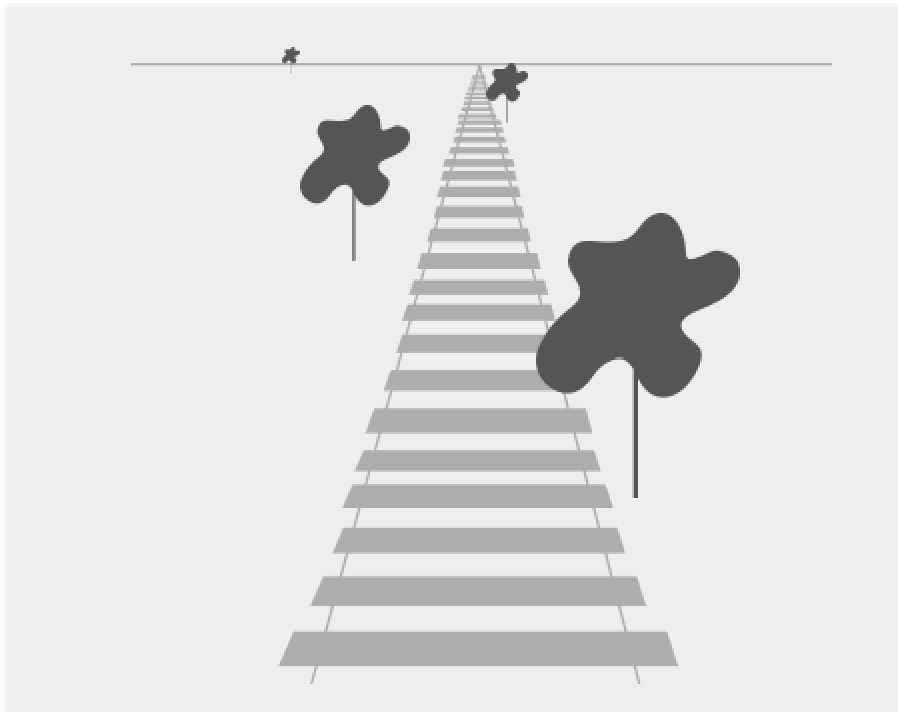
Перші два параметри визначають ліву та праву координату фрустуму, а третій та четвертий параметри визначають нижню та верхню частину фрустуму. З цими чотирма точками ми визначили розмір близького та далекого планів, а п'ятий та шостий параметри визначають відстані між близьким та далеким планом. Ця

конкретна матриця проекції перетворює всі координати між цими значеннями x, y та z в нормалізовані пристрої координат.

Ортографічна матриця проекції безпосередньо відображає координати на площину 2D, яка є вашим екраном, але насправді пряма проекція виробляє нереалістичні результати, оскільки проекція не враховує перспективу. Це виправляє матриця перспективної проекції.

Перспективна проекція

Якщо ви коли-небудь насолоджувалися графікою реального життя, ви помітите, що об'єкти, які знаходяться далеко, здаються набагато меншими. Цей дивний ефект - це те, що ми називаємо перспективою. Перспектива особливо помітна, коли дивишся на кінець нескінченної автомагістралі чи залізниці, як показано на наступному зображенні:



Як видно, через перспективу лінії здаються співпадати на достатню відстань. Це саме той ефект, який намагається імітувати матриця перспективної проекції. Матриця проекції відображає певний діапазон видимого простору на простір обрізки, але також маніпулює значенням w для кожної координати вершини так, що чим далі координата вершини від глядача, тим вищим стає цей компонент w. Після того як координати трансформовані в простір обрізки, вони перебувають у діапазоні від -w до w (все, що знаходиться за межами цього діапазону, обрізається). OpenGL вимагає, щоб видимі координати потрапляли у діапазон від -1,0 до 1,0 як вихід з вершинного шейдера, тому, як тільки координати знаходяться в просторі обрізки, до них застосовується перспективне ділення координат простору обрізки:

$$out = \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix}$$

Кожна компонента координати вершини ділиться на свій компонент w, що призводить до зменшення координат вершин, чим далі вершина знаходиться від глядача. Це ще одна причина, чому компонент w важливий, оскільки він допомагає нам з перспективною проекцією. Отримані координати знаходяться в просторі нормалізованих пристроїв. Якщо вас цікавить, як власне обчислюються матриці ортографічної та перспективної проекції (і ви не занадто боїтеся математики), я можу порекомендувати цю відмінну статтю від Songho.

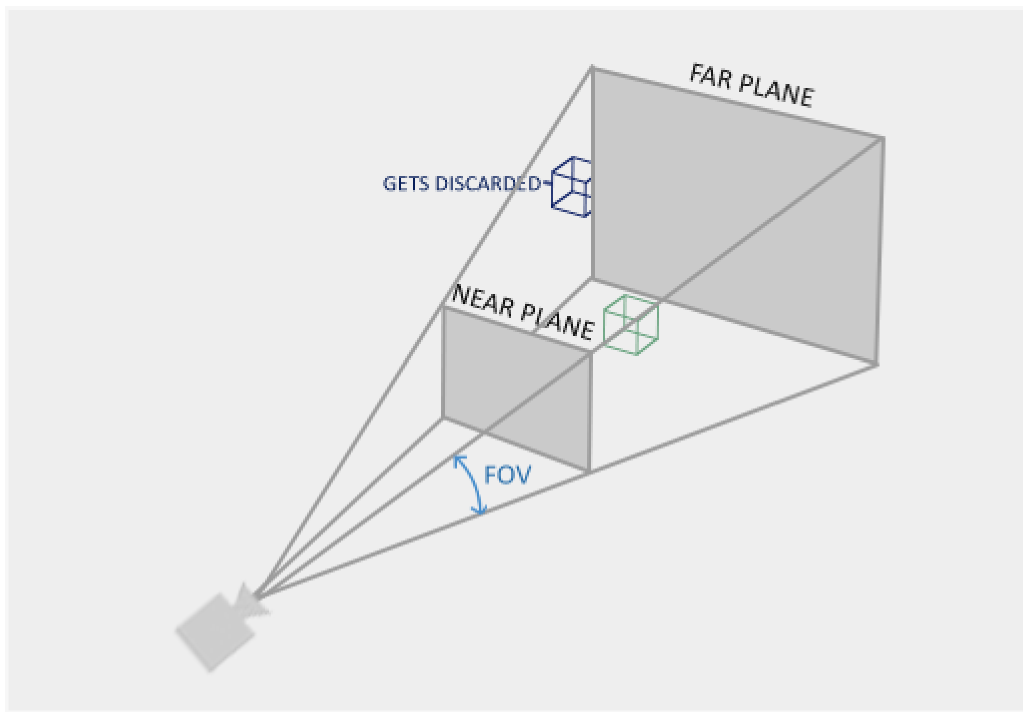
Матрицю перспективної проекції можна створити в GLM так:

```
glm::mat4 proj = glm::perspective(glm::radians(45.0f), (float)SCR_WIDTH /
(float)SCR_HEIGHT, 0.1f, 100.0f);
```

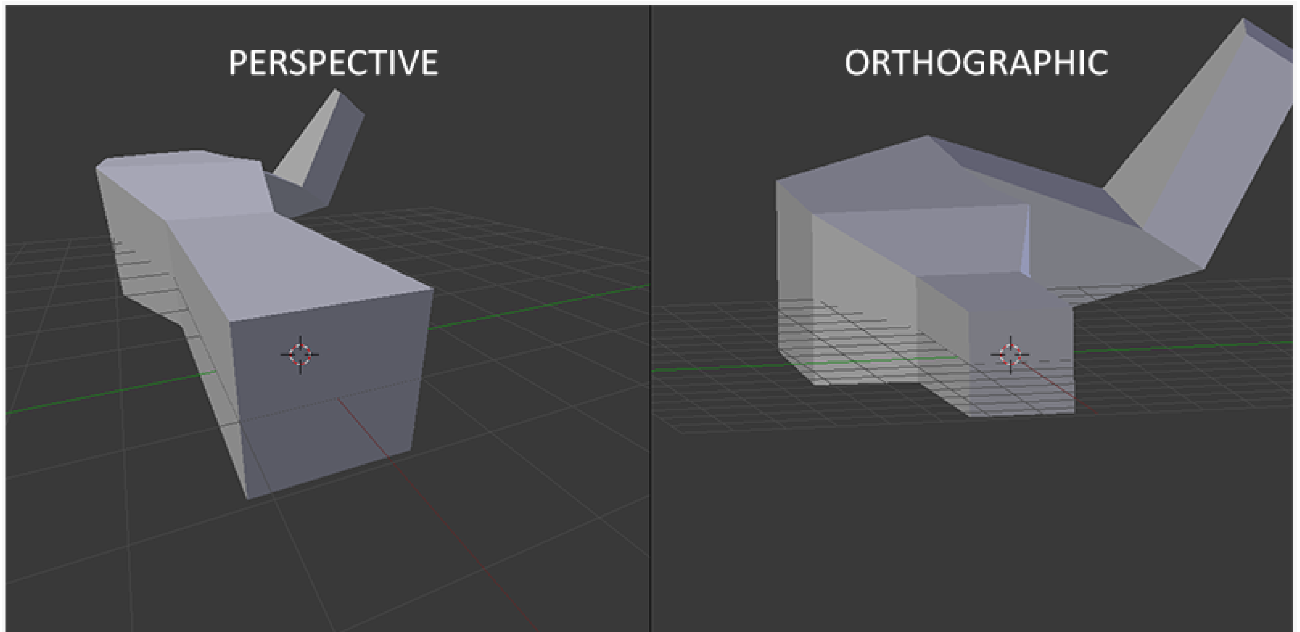
У цьому прикладі `glm::perspective` - це вбудована функція GLM для створення матриці перспективної проекції. Параметри цієї функції включають:

- Поле зору (FOV): `glm::radians(45.0f)` - у цьому випадку 45 градусів, але може бути змінене відповідно до вашого проекту.
- Відношення ширини та висоти вікна: `(float)SCR_WIDTH / (float)SCR_HEIGHT` - це важливий параметр для забезпечення коректного відображення в 3D просторі.
- Відстань до ближньої та дальньої площини обрізки: `0.1f` та `100.0f` - ці значення можна змінити в залежності від ваших потреб.

Ця матриця визначає перспективну проекцію і трансформує координати вершин у простір обрізки.



При використанні ортографічної проекції кожна з координат вершин безпосередньо відображається в простір обрізки без будь-якого складного ділення для перспективи (все їще відбувається ділення для перспективи, але компонент w не маніпулюється (він залишається 1) і, таким чином, не має ефекту). Оскільки ортографічна проекція не використовує проекцію перспективи, об'єкти, які знаходяться далеко, не здаються меншими, що призводить до дивовижного візуального ефекту. З цієї причини ортографічна проекція в основному використовується для 2D-відображень і деяких архітектурних або інженерних застосувань, де ми бажаємо, щоб вершини не спотворювалися перспективою. Додатки, такі як Blender, які використовуються для 3D-моделювання, іноді використовують ортографічну проекцію для моделювання, оскільки вона більш точно зображує розміри кожного об'єкта. Нижче ви побачите порівняння обох методів проекції в Blender:



На цьому зображенні ви бачите, як ортографічна проекція та перспективна проекція впливають на відображення об'єктів в Blender. Ортографічна проекція зберігає розміри об'єктів, незалежно від їх віддаленості, тоді як перспективна проекція враховує ефект перспективи, роблячи об'єкти, розташовані далеко, меншими.

Ми створюємо матрицю трансформації для кожного з вищезазначених етапів: матриці моделі, виду та проекції. Координати вершин потім трансформуються в обрізку, як показано нижче:

$$V_{clip} = M_{projection} \cdot M_{view} \cdot M_{model} \cdot V_{local}$$

```
// Define the model matrix (transforms from local space to world space)
glm::mat4 model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(0.0f, 0.0f, 0.0f)); // Example translation

// Define the view matrix (transforms from world space to view space)
glm::mat4 view = glm::mat4(1.0f);
view = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f),   // Camera position in world space
                   glm::vec3(0.0f, 0.0f, 0.0f),   // Point the camera is looking at
                   glm::vec3(0.0f, 1.0f, 0.0f));  // Up vector
```

```
// Define the projection matrix (transforms from view space to clip space)
glm::mat4 projection = glm::perspective(glm::radians(45.0f),   // Field of view
                                800.0f / 600.0f,      // Aspect ratio
                                0.1f, 100.0f);        // Near and far planes

// Combine the transformations into a single matrix (model * view * projection)
glm::mat4 mvp = projection * view * model;

// Transform a vertex coordinate to clip coordinates
glm::vec4 vertexLocal(1.0f, 0.0f, 0.0f, 1.0f);  // Example local space coordinate
glm::vec4 vertexClip = mvp * vertexLocal;
```

У цьому коді ми створюємо матриці моделі, виду та проекції. Потім ми об'єднуємо їх у єдину матрицю (MVP), яка представляє собою їхнє композицію. Координата вершини у локальному просторі трансформується в координати обрізки за допомогою цієї MVP-матриці.

Це складна тема для розуміння, тому якщо ви все ще не зовсім впевнені, для чого використовуються простір, вам не треба хвилюватися. Нижче ви побачите, як ми можемо фактично використовувати ці координатні простори, і достатньо прикладів буде показано в наступних розділах.

Переходимо до 3D

Тепер, коли ми знаємо, як трансформувати 3D-координати в 2D-координати, ми можемо почати рендерити справжні 3D-об'єкти замість нудної 2D-площини, яку ми показували дотепер.

Щоб почати малювати в 3D, ми спочатку створимо матрицю моделі. Матриця моделі складається з переміщень, масштабувань та/або обертань, які ми хочемо застосувати для трансформації всіх вершин об'єкта в глобальний простір світу. Давайте трошки трансформуємо нашу площину, обертаючи її навколо осі x так, щоб вона виглядала, як лежить на підлозі. Тоді матриця моделі виглядає наступним чином:

```
glm::mat4 model = glm::mat4(1.0f);
model = glm::rotate(model, glm::radians(-55.0f), glm::vec3(1.0f, 0.0f, 0.0f));
```

Помножаючи координати вершин на цю матрицю моделі, ми трансформуємо координати вершин у світові координати. Наша площина, яка трошки знаходиться на підлозі, таким чином представляє площину в глобальному світі.

Далі нам потрібно створити матрицю виду. Ми хочемо трошки відійти назад у сцені, щоб об'єкт став видимим (коли ми знаходимося в світовому просторі, ми розташовані в точці (0,0,0)). Щоб рухати камеру вздовж сцени, подумайте про наступне:

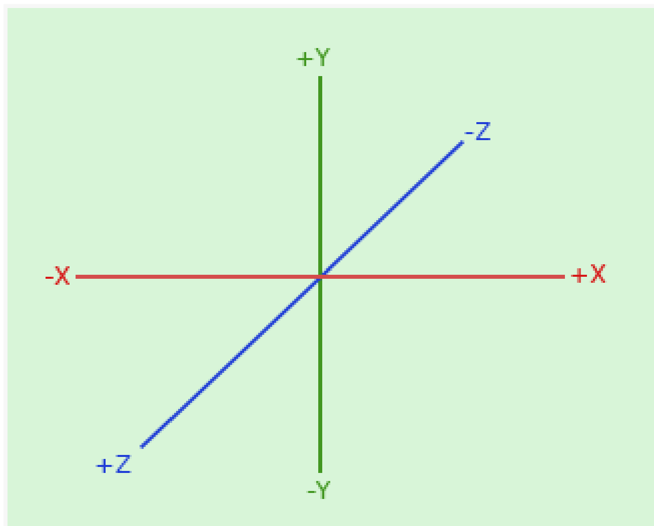
Рухати камеру назад - це те саме, що рухати всю сцену вперед.

Це саме те, що робить матриця виду: ми рухаємо всю сцену назад, протилежно до того, куди ми хочемо, щоб камера рухалася.

Оскільки ми хочемо рухатися назад, і так як OpenGL - це праворучна система, нам потрібно рухатися вздовж позитивної вісі z. Ми це робимо, транслюючи сцену в напрямку від'ємної вісі z. Це створює враження, ніби ми рухаємося назад.

Праворучна система

За конвенцією, OpenGL є праворучною системою. За суттю, це означає, що позитивна вісь x розташована справа від вас, позитивна вісь y вгорі, а позитивна вісь z віддаче. Подумайте про ваш екран як про центр трьох вісей, а позитивна вісь z проходить через ваш екран до вас. Вісі зображено наступним чином:



Щоб зрозуміти, чому його називають праворучним, виконайте наступне:

Простягніть вашу праву руку вздовж позитивної вісі y з рукою зверху.

Ваш великий палець повинен вказувати праворуч.

Ваш вказівний палець повинен вказувати вгору.

Тепер нахилите вниз ваш середній палець на 90 градусів.

Якщо ви все зробили правильно, ваш великий палець повинен вказувати в напрямку позитивної вісі x, вказівний палець - в напрямку позитивної вісі y, а ваш середній палець - в напрямку позитивної вісі z. Якщо ви це зробите лівою рукою, ви побачите, що вісь z відображена зворотно. Це відомо як ліворучна система і широко використовується у DirectX. Зауважте, що в нормалізованих пристройних координатах OpenGL фактично використовує ліворучну систему (матриця проекції перемикає засіб).

Ми обговоримо, як рухатися навколо сцени більш детально в наступному розділі. Наразі матриця перегляду виглядає наступним чином:

```
glm::mat4 view = glm::mat4(1.0f);
// зверніть увагу, що ми перекладаємо сцену в оберненому напрямку того, куди хочемо рухатися
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
```

Останнє, що нам потрібно визначити, це матриця проекції. Ми хочемо використовувати перспективну проекцію для нашої сцени, тому оголосимо матрицю проекції так:

```cpp
glm::mat4 projection;
projection = glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 100.0f);
```

Тепер, коли ми створили матриці перетворення, ми повинні передавати їх у наші шейдери. Спочатку давайте оголосимо матриці перетворення як юніформи в вершинному шейдері та помножимо їх на координати вершин:

```glsl
#version 330 core
layout (location = 0) in vec3 aPos;
...
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    // зверніть увагу, що ми читаємо множення зправа наліво
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    ...
}
```

Також ми повинні відправити матриці в шейдер (це зазвичай робиться кожен кадр, оскільки матриці перетворення часто змінюються):

```cpp
int modelLoc = glGetUniformLocation(ourShader.ID, "model");
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
... // те ж саме для матриці перегляду та матриці проекції
```

```

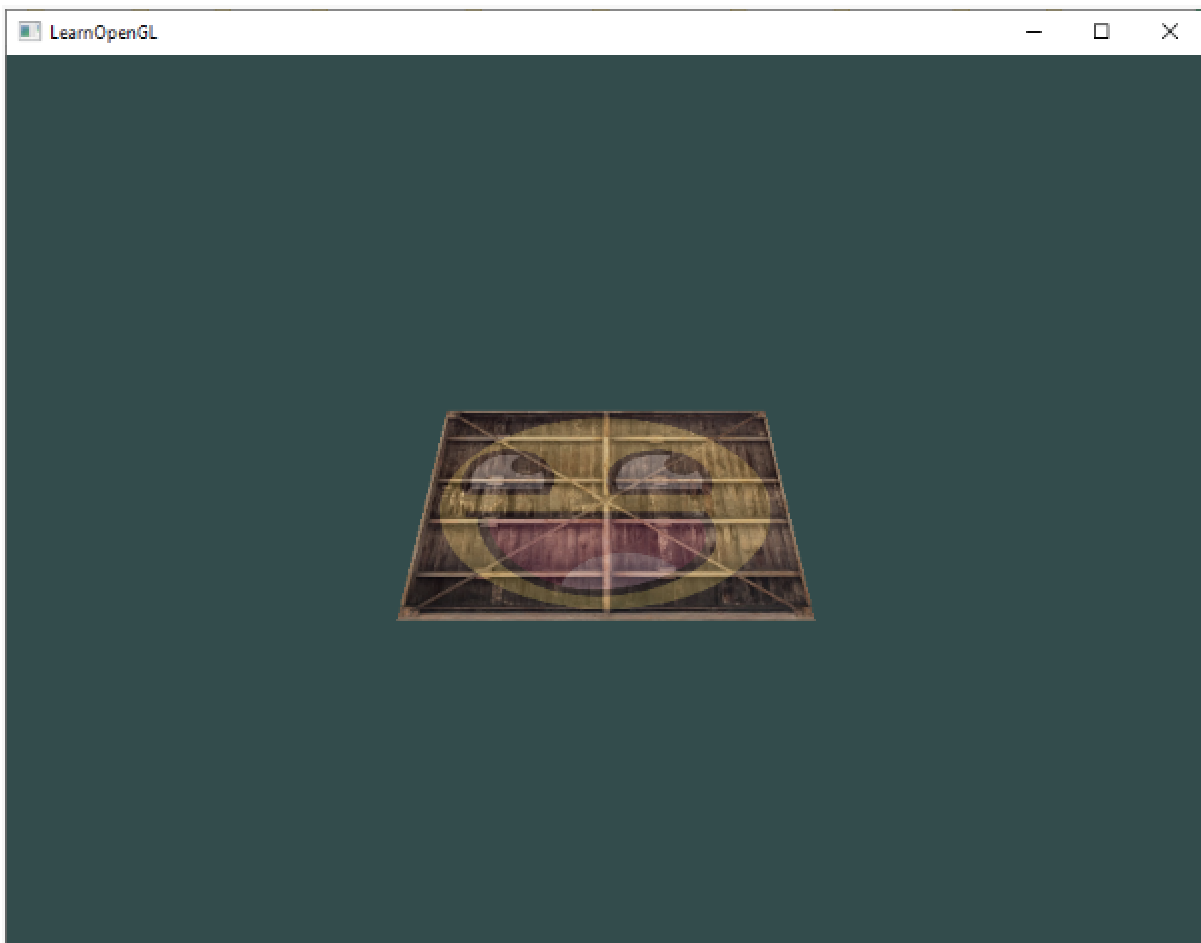
Тепер, коли наші координати вершин трансформуються за допомогою матриць model, view та projection, кінцевий об'єкт повинен бути:

- Наклонений назад на підлогу.

- Трошки далі від нас.

- Відображається з перспективою (він повинен зменшуватися, чим далі його вершини).

Давайте перевіримо, чи результат дійсно відповідає цим вимогам.




Справді виглядає так, ніби літак є тривимірним, який стоїть на якійсь уявній підлозі. Якщо ви не отримуєте той самий результат, порівняйте свій код із повним вихідним кодом .

## Більше 3D
```

Поки що ми працювали з 2D-площиною, навіть у 3D-просторі, тому давайте виберемо авантюрний шлях і розширимо нашу 2D-площину до 3D-куба. Щоб відобразити куб, нам потрібно всього 36 вершин (6 граней * 2 трикутники * 3 вершини кожна). 36 вершин – це багато, щоб підсумувати, тому ви можете отримати їх [звідси](#) .

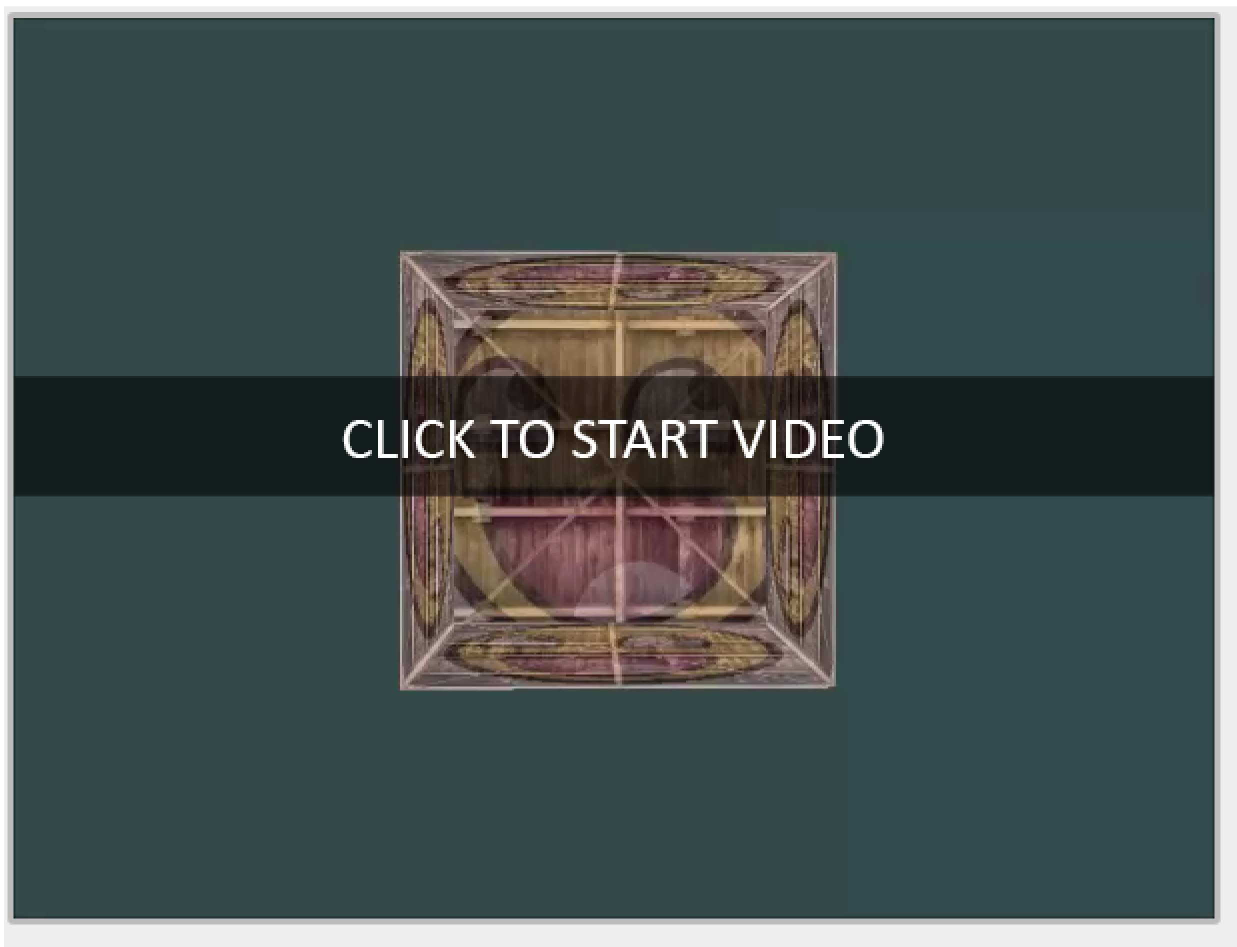
Для розваги ми дозволимо кубу обертатися з часом:

```
model = glm::rotate(model, (float)glfwGetTime() * glm::radians(50.0f), glm::vec3(0.5f, 1.0f, 0.0f));
```

А потім ми намалюємо куб за допомогою`glDrawArrays`(оскільки ми не вказували індекси), але цього разу з кількістю вершин 36.

```
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Ви повинні отримати щось подібне до наступного:



https://learnopengl.com/video/getting-started/coordinate_system_no_depth.mp4

Він трохи нагадує куб, але щось не так. Одні сторони куба накреслені поверх інших сторін куба. Це відбувається тому, що коли OpenGL малює ваш куб трикутник за трикутником, фрагмент за фрагментом, він перезаписує будь-які кольори пікселів, які могли бути вже намальовані там раніше. Оскільки OpenGL не дає жодних гарантій щодо порядку візуалізації трикутників (в межах одного виклику малювання), деякі трикутники малюються один над одним, навіть якщо один має бути чітко попереду іншого.

На щастя, OpenGL зберігає інформацію про глибину в буфері під назвоюz-буферщо дозволяє OpenGL вирішувати, коли малювати поверх пікселя, а коли ні. Використовуючи z-буфер, ми можемо налаштувати OpenGL для тестування глибини.

Z-буфер

OpenGL зберігає всю інформацію про глибину в z-буфері, також відомому як аглибинний буфер. GLFW автоматично створює для вас такий буфер (подібно до того, як він має кольоровий буфер, який зберігає кольори вихідного зображення). Глибина зберігається в кожному фрагменті (як значення фрагмента z), і щоразу, коли фрагмент хоче вивести свій колір, OpenGL порівнює його значення глибини з z-буфером. Якщо поточний фрагмент знаходиться позаду іншого фрагмента, він відкидається, інакше перезаписується. Цей процес називаєтьсяглибинне тестуванняі виконується автоматично OpenGL.

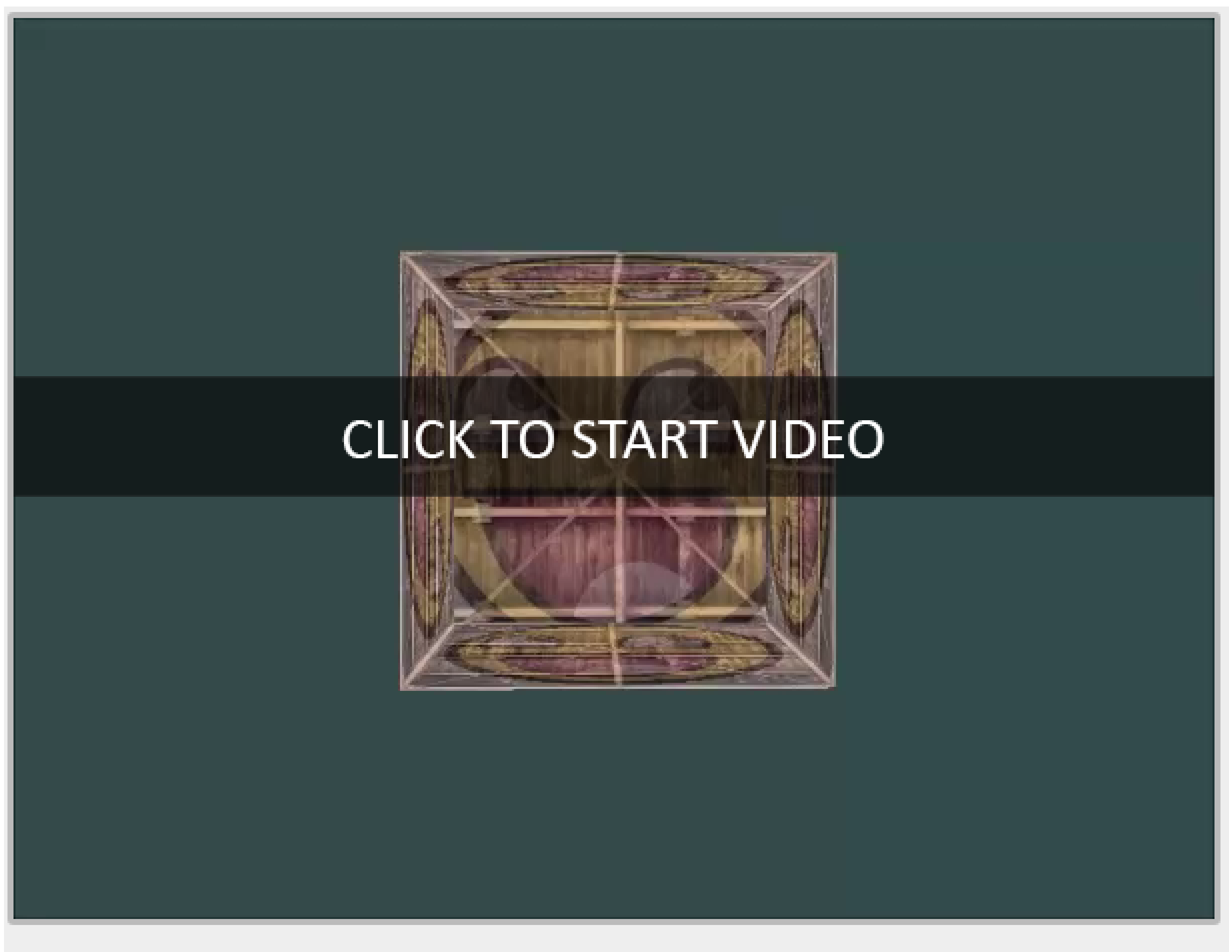
Однак, якщо ми хочемо переконатися, що OpenGL дійсно виконує глибинне тестування, нам спочатку потрібно повідомити OpenGL, що ми хочемо ввімкнути глибинне тестування; він вимкнений за замовчуванням. Ми можемо включити глибинне тестування за допомогою`glEnable`. The`glEnable`і`glВимкнути`функції дозволяють нам увімкнути/вимкнути певні функції в OpenGL. Потім ця функція вмикається/вимикається, доки не буде зроблено інший виклик, щоб вимкнути/увімкнути її. Прямо зараз ми хочемо ввімкнути тестування глибини, увімкнувши `GL_DEPTH_TEST` :

```
glEnable(GL_DEPTH_TEST);
```

Оскільки ми використовуємо буфер глибини, ми також хочемо очищати буфер глибини перед кожною ітерацією візуалізації (інакше інформація про глибину попереднього кадру залишається в буфері). Подібно до очищення буфера кольорів, ми можемо очистити буфер глибини, вказавши біт `DEPTH_BUFFER_BIT` у`glClear`функція:

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

Давайте повторно запустимо нашу програму та перевіримо, чи тепер OpenGL виконує глибинне тестування:



https://learnopengl.com/video/getting-started/coordinate_system_depth.mp4

Там ми йдемо! Повністю текстурований куб із належним тестуванням глибини, який обертається з часом. Перевірте вихідний код тут .

Більше кубиків!

Скажімо, ми хочемо показати на екрані 10 наших кубів. Кожен кубик виглядатиме однаково, але відрізнятиметься лише тим, де він розташований у світі, з різним обертанням. Графічний макет куба вже визначено, тому нам не потрібно змінювати наші буфери чи масиви атрибутів під час візуалізації додаткових об’єктів. Єдине, що нам потрібно змінити для кожного об’єкта, це його модельну матрицю, де ми перетворюємо куби на світ.

Спочатку давайте визначимо вектор трансляції для кожного куба, який визначає його положення у світовому просторі. Ми визначимо 10 позицій куба в `glm::vec3`масиві:

```
glm::vec3 cubePositions[] = {
    glm::vec3( 0.0f,  0.0f,  0.0f),
```

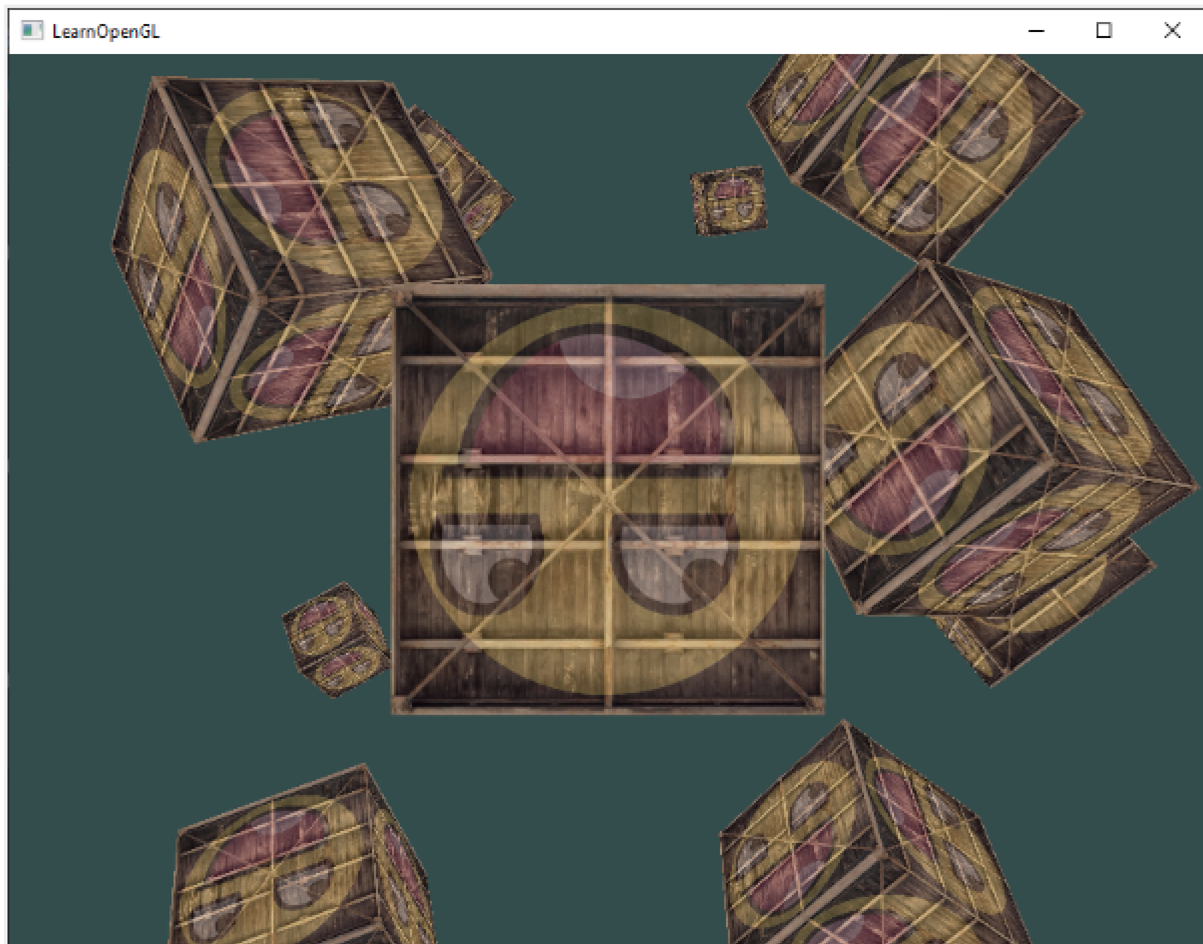
```
    glm::vec3( 2.0f,  5.0f, -15.0f),
    glm::vec3(-1.5f, -2.2f, -2.5f),
    glm::vec3(-3.8f, -2.0f, -12.3f),
    glm::vec3( 2.4f, -0.4f, -3.5f),
    glm::vec3(-1.7f,  3.0f, -7.5f),
    glm::vec3( 1.3f, -2.0f, -2.5f),
    glm::vec3( 1.5f,  2.0f, -2.5f),
    glm::vec3( 1.5f,  0.2f, -1.5f),
    glm::vec3(-1.3f,  1.0f, -1.5f)
};
```

Тепер у циклі візуалізації, який ми хочемо викликати `glDrawArrays` 10 разів, але цього разу кожного разу надсилайте іншу матрицю моделі до вершинного шейдера, перш ніж надсилати виклик малювання. Ми створимо невеликий цикл у циклі візуалізації, який рендерить наш об’єкт 10 разів з іншою матрицею моделі щоразу. Зауважте, що ми також додаємо невелике унікальне обертання до кожного контейнера.

```
glBindVertexArray(VAO);
for(unsigned int i = 0; i < 10; i++)
{
    glm::mat4 model = glm::mat4(1.0f);
    model = glm::translate(model, cubePositions[i]);
    float angle = 20.0f * i;
    model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));
    ourShader.setMat4("model", model);

    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

Цей фрагмент коду оновлюватиме матрицю моделі кожного разу, коли малюється новий куб, і робити це загалом 10 разів. Прямо зараз ми маємо подивитися на світ, наповнений 10 кубиками, що дивно повертаються:



Ідеально! Схоже, наш контейнер знайшов друзів-однодумців. Якщо ви застрягли, перевірте, чи можете ви порівняти свій код із вихідним кодом .

вправи

- Спробуйте поекспериментувати з параметрами FoVi функції `aspect-ratio`GLM `projection`. Подивіться, чи можете ви зрозуміти, як вони впливають на перспективу.
- Пограйте з матрицею перегляду, переміщаючись у кількох напрямках, і подивіться, як змінюється сцена. Думайте про матрицю перегляду як об'єкт камери.
- Спробуйте змусити кожен 3-й контейнер (включаючи 1-й) обертатися з часом, залишаючи інші контейнери статичними, використовуючи лише матрицю моделі: рішення .