Лекція 5

Лабораторна робота 5

Трансформації

Тепер ми знаємо, як створювати об'єкти, розфарбовувати їх та/або надавати їм деталізованого вигляду за допомогою текстур, але вони все одно не такі цікаві, оскільки є статичними об'єктами. Ми можемо спробувати змусити їх рухатися, змінюючи їхні вершини та переналаштовуючи їхні буфери кожного кадру, але це громіздко і коштує досить багато обчислювальної потужності. Існує набагато кращий спосіб трансформувати об'єкт - це використання (декількох) матричних об'єктів. Це не означає, що ми будемо говорити про кунг-фу та великий цифровий штучний світ.

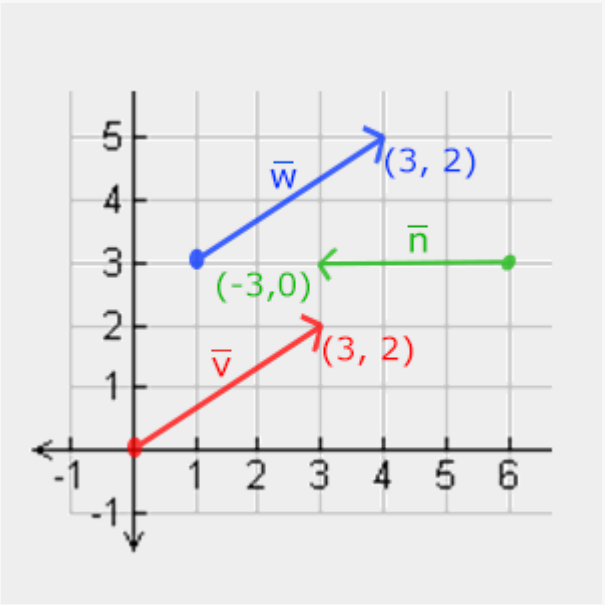
Матриці - це дуже потужні математичні конструкції, які спочатку здаються страшними, але як тільки ви звикнете до них, вони виявляться надзвичайно корисними. При обговоренні матриць нам доведеться зробити невелике занурення в математику, і для більш схильних до математики читачів я опублікую додаткові ресурси для подальшого читання.

Однак, щоб повністю зрозуміти перетворення, ми повинні спочатку заглибитися у вектори, перш ніж перейти до матриць. Мета цієї глави - дати вам базові математичні знання з тем, які знадобляться пізніше. Якщо ці теми є складними, спробуйте зрозуміти їх настільки, наскільки це можливо, і поверніться до цієї глави пізніше, щоб повторити концепції, коли вони вам знадобляться.

Вектори

У найпростішому визначенні, вектори - це напрямки і нічого більше. Вектор має напрямок і величину (також відому як сила або довжина). Ви можете думати про вектори, як про напрямки на карті скарбів: "Ідіть наліво 10 кроків, тепер на північ 3 кроки і направо 5 кроків"; тут "наліво" - це напрямок, а "10 кроків" - величина вектора. Таким чином, напрямок на карті скарбів містить 3 вектори. Вектори можуть мати будь-яку розмірність, але ми зазвичай працюємо з розмірністю від 2 до 4. Якщо вектор має 2 виміри, він представляє напрямок на площині (уявіть собі 2D-графіки), а коли він має 3 виміри, він може представляти будь-який напрямок у 3D-світі.

Нижче ви побачите 3 вектори, де кожен вектор представлений у `(x,y)` вигляді стрілок у двовимірному графіку. Оскільки відображення векторів у 2D (а не в 3D) більш інтуїтивно зрозуміле, ви можете розглядати 2D-вектори як 3D-вектори з zкоординатою `0`. Оскільки вектори представляють напрямки, початок вектора не змінює його значення. На графіку нижче ми бачимо, що вектори рівні, навіть якщо вони мають різне походження:



При описі векторів математики зазвичай вважають за краще описувати вектори як символи символів з невеликою смугою над головою.
. Крім того, під час відображення векторів у формулах вони зазвичай відображаються таким чином:

$$\bar{v} = \begin{pmatrix} x \\ \mathrm{p} \\ \mathrm{з} \end{pmatrix}$$

Оскільки вектори задаються як напрямки, іноді буває важко візуалізувати їх як позиції. Якщо ми хочемо зобразити вектори як позиції, ми можемо уявити початок вектора напрямку `(0,0,0)`, а потім вказати на певний напрямок, який вказує на точку, зробивши його вектором позиції (ми також можемо вказати інший початок, а потім сказати: "цей вектор вказує на цю точку в просторі з цього початку координат"). Тоді вектор положення `(3,5)` `вказуватиме на точку` `(3,5)` на графіку з початком координат `(0,0)`. Таким чином, за допомогою векторів ми можемо описувати напрямки **і** положення в 2D і 3D просторі.

Подібно до звичайних чисел, ми також можемо визначити декілька операцій над векторами (деякі з них ви вже бачили).

Операції зі скалярними векторами

Скаляр - це одноцифрове число. При додаванні/відніманні/множенні і діленні вектора зі скаляром ми просто додаємо/віднімаємо/множимо і ділимо кожен елемент вектора на скаляр. Для додавання це буде виглядати так:

$$\begin{pmatrix}1\\2\\3\end{pmatrix} + x \rightarrow \begin{pmatrix}1\\2\\3\end{pmatrix} + \begin{pmatrix}x\\x\\x\end{pmatrix} = \begin{pmatrix}1 + \mathrm{x}\\2 + \mathrm{x}\\3 + \mathrm{x}\end{pmatrix}$$

Де+може бути +,−,⋅або÷де ⋅ є оператором множення.

Векторна зміна знаку

Зміна знаку вектора призводить до вектора в оберненому напрямку. Вектор, який спрямований на північний захід, буде спрямований на південний схід після зміни знаку. Для зміни знаку вектора ми додаємо знак мінус до кожної компоненти (ви також можете представити це як множення вектора на скалярну величину -1):

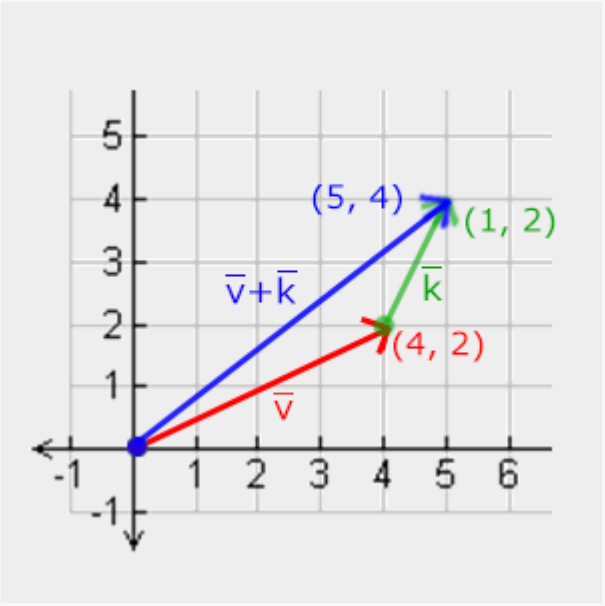
$$-\bar{v} = -\begin{pmatrix}v_x\\v_y\\v_z\end{pmatrix} = \begin{pmatrix}-v_x\\-v_y\\-v_z\end{pmatrix}$$

Додавання та віднімання

Додавання двох векторів визначається як покомпонентне додавання, тобто кожна компонента одного вектора додається до відповідної компоненти іншого вектора таким

$$\bar{v} = \begin{pmatrix}1\\2\\3\end{pmatrix}, \bar{k} = \begin{pmatrix}4\\5\\6\end{pmatrix} \rightarrow \bar{v} + \bar{k} = \begin{pmatrix}1+4\\2+5\\3+6\end{pmatrix} = \begin{pmatrix}5\\7\\9\end{pmatrix}$$

Візуально це виглядає так на векторах `v=(4,2)` і `k=(1,2)`, де другий вектор додається поверх кінця першого вектора, щоб знайти кінцеву точку результуючого вектора (метод "від голови до хвоста"):



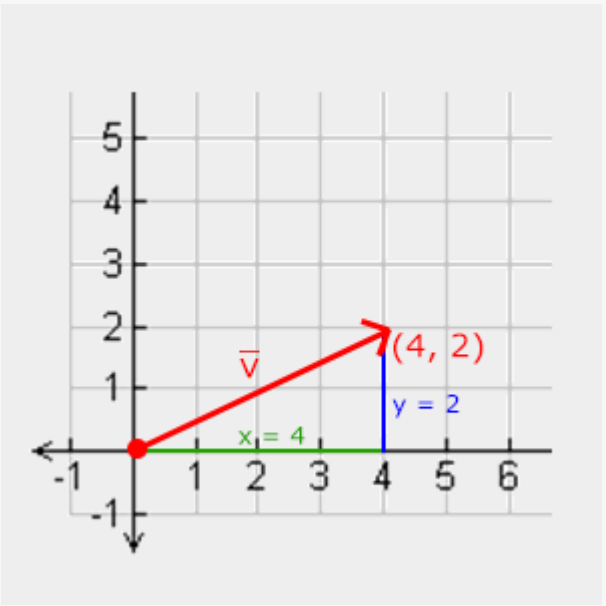
Подібно до звичайного додавання і віднімання, векторне віднімання - це те саме, що додавання з від'ємним другим вектором:

$$\bar{v} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \bar{k} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \rightarrow \bar{v} + -\bar{k} = \begin{pmatrix} 1+(-4) \\ 2+(-5) \\ 3+(-6) \end{pmatrix} = \begin{pmatrix} -3 \\ -3 \\ -3 \end{pmatrix}$$

Віднімання двох векторів один від одного дає вектор, який є різницею позицій, на які вказують обидва вектори. Це виявляється корисним у певних випадках, коли нам потрібно отримати вектор, який є різницею між двома точками.

Довжина

Щоб знайти довжину/величину вектора, ми використовуємо теорему Піфагора, яку ви, можливо, пам'ятаєте з уроків математики. Вектор утворює трикутник, якщо уявити його окремі компоненти x та y як дві сторони трикутника:



Оскільки довжини двох сторін (x, y) відомі, і ми хочемо дізнатися довжину нахиленої ми можемо обчислити його за допомогою теореми Піфагора як:

$$||\bar{v}|| = \sqrt{x^2 + y^2}$$

Where $||\bar{v}||$ is denoted as *the length of vector* $\bar{v}$. This is easily extended to 3D by adding z^2 to the equation.

In this case the length of vector (4, 2) equals:

$$||\bar{v}|| = \sqrt{4^2 + 2^2} = \sqrt{16 + 4} = \sqrt{20} = 4.47$$

Which is 4.47.

Існує також особливий тип вектора, який ми називаємо одиничним вектором. Одиничний вектор має одну додаткову властивість - його довжина дорівнює рівно 1. Ми можемо обчислити одиничний вектор

$$\hat{n} = \frac{\bar{v}}{||\bar{v}||}$$

Це називається нормалізацією вектора. Одиничні вектори позначаються зверху горизонтальною рискою (для позначення одиничної довжини) і, як правило, їх легше обробляти, особливо коли нам цікава лише їхні напрямки (напрямок не змінюється, якщо ми змінюємо довжину вектора).

Множення вектор-вектор

Множення двох векторів - це трохи дивний випадок. Звичайне множення не визначено для векторів, оскільки воно не має візуального значення, але у нас є два конкретних варіанти, з якими ми могли б працювати при множенні: одним є скалярний добуток, позначений як $\bar{v} \cdot \bar{k}$, а іншим - векторний добуток, позначений як $\bar{v} \times \bar{k}$.

Скалярний добуток

Скалярний добуток двох векторів дорівнює скалярному добутку їх довжин, помноженому на косинус кута між ними. Якщо це здається заплутаним, подивіться на його формулу:

Точковий добуток

Точковий добуток двох векторів дорівнює скалярному добутку їхніх довжин на косинус кута між ними. Якщо це звучить незрозуміло, подивіться на його формулу:

$$\bar{v} \cdot \bar{k} = ||\bar{v}|| \cdot ||\bar{k}|| \cdot \cos\theta$$

Де кут між ними позначається як тета (θ). Чому це цікаво? Припустімо, що $\bar{v}$ та $\bar{k}$ є одиничними векторами, тобто їх довжина дорівнює 1. Це фактично спростило б формулу до:

$$\hat{v} \cdot \hat{k} = 1 \cdot 1 \cdot \cos\theta = \cos\theta$$

Тепер скалярний добуток лише визначає кут між обома векторами. Можливо, ви пам'ятаєте, що функція косинус або cos дорівнює 0, коли кут дорівнює 90 градусів або 1, коли кут дорівнює 0 градусів. Це дозволяє нам легко перевірити, чи обидва вектори ортогональні один одному або паралельні один одному за допомогою скалярного добутку (ортогональні означає, що вектори перпендикулярні один одному). Якщо ви хочете дізнатися більше про функції синуса або косинуса, я рекомендую переглянути відео на тему основи тригонометрії на Khan Academy.

Ви також можете обчислити кут між двома нормованими векторами, але потім вам доведеться розділити результат на довжини обох векторів, щоб отримати cosθ.

Отже, як ми обчислюємо скалярний добуток? Скалярний добуток - це компонентне множення, в результаті чого додавати результати разом. Він виглядає так з двома одиничними векторами (ви можете перевірити, що їхня довжина дорівнює точно 1):

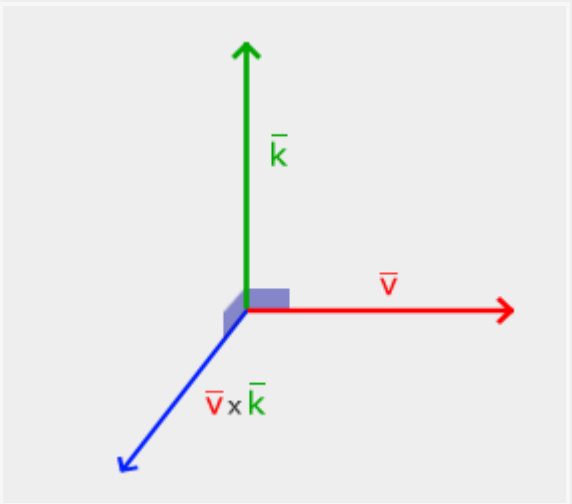
Тож як ми обчислюємо точковий добуток? Точковий добуток - це покомпонентне множення, де ми додаємо результати разом. Це виглядає так для двох одиничних

$$\begin{pmatrix} 0.6 \\ -0.8 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = (0.6 * 0) + (-0.8 * 1) + (0 * 0) = -0.8$$

Для обчислення кута між цими двома одиничними векторами ми використовуємо обернену функцію косинуса cos^(-1), і це дає нам 143,1 градуса. Таким чином, ми ефективно обчислили кут між цими двома векторами. Скалярний добуток дуже корисний при обчисленнях освітлення в подальших розділах.

Векторний добуток

Векторний добуток визначений лише в просторі 3D і приймає два не-паралельних вектори на вході, продукуючи третій вектор, який ортогональний обом вхідним векторам. Якщо обидва вхідні вектори також є ортогональними один одному, векторний добуток призведе до отримання 3 ортогональних векторів; це буде корисно в наступних розділах. Наступне зображення показує, як це виглядає в просторі 3D:



На відміну від інших операцій, векторний добуток не є дуже інтуїтивним без вивчення лінійної алгебри, тому найкраще просто запам'ятати формулу і ви будете в порядку

(або ні, ви, ймовірно, також будете в порядку). Нижче ви побачите векторний добуток між двома ортогональними векторами A та B:

$$\begin{pmatrix} A_x \\ A_y \\ A_z \end{pmatrix} \times \begin{pmatrix} B_x \\ B_y \\ B_z \end{pmatrix} = \begin{pmatrix} A_y \cdot B_z - A_z \cdot B_y \\ A_z \cdot B_x - A_x \cdot B_z \\ A_x \cdot B_y - A_y \cdot B_x \end{pmatrix}$$

Як бачите, воно, здавалося, не має сенсу. Однак, якщо ви просто слідуєте цим крокам, ви отримаєте інший вектор, який є ортогональним до ваших вихідних векторів.

Матриці

Тепер, коли ми майже розглянули усе про вектори, прийшов час перейти до матриць! Матриця - це прямокутний масив чисел, символів і/або математичних виразів. Кожен окремий елемент в матриці називається елементом матриці. Нижче показано приклад матриці 2x3:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Матриці індексуються за (i, j), де i - це рядок, а j - стовпець, тому ця матриця називається матрицею 2x3 (3 стовпці і 2 рядки, також відомо як розміри матриці). Це протилежність тому, що ви звикли до індексації 2D графіків як (x, y). Щоб отримати значення 4, нам потрібно індексувати його як (2, 1) (другий рядок, перший стовпець).

Матриці в основному нічим більше не є, просто прямокутні масиви математичних виразів. У них є дуже гарний набір математичних властивостей, і, як і вектори, ми можемо визначити кілька операцій на матрицях, а саме: додавання, віднімання і множення.

Додавання та віднімання

Додавання та віднімання матриць між двома матрицями виконується на елементній основі. Тобто застосовуються ті самі загальні правила, які нам відомі для звичайних чисел, але вони застосовуються до елементів обох матриць з однаковим індексом. Це означає, що додавання та віднімання визначено лише для матриць однакових розмірів. Матрицю 3x2 і матрицю 2x3 (або матрицю 3x3 і матрицю 4x4) не можна

додавати або віднімати одну від одної. Подивимося, як працює додавання матриць на прикладі двох матриць 2x2:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

The same rules apply for matrix subtraction:

$$\begin{bmatrix} 4 & 2 \\ 1 & 6 \end{bmatrix} - \begin{bmatrix} 2 & 4 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 4-2 & 2-4 \\ 1-0 & 6-1 \end{bmatrix} = \begin{bmatrix} 2 & -2 \\ 1 & 5 \end{bmatrix}$$

Матрично-скалярні добутки

Матрично-скалярний добуток множить кожен елемент матриці на скаляр. Наступний приклад ілюструє множення:

$$2 \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 \cdot 1 & 2 \cdot 2 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

Тепер зрозуміло, чому ці числа називаються скалярами. Скаляр, по суті, *масштабує* всі елементи матриці за своїм значенням. У попередньому прикладі всі елементи було масштабовано на 2.

Множення матриці на матрицю

Множення матриць може здатися не складним, але дещо важким для вивчення. Множення матриць фактично означає слідувати набору попередньо визначених правил при множенні. Проте є кілька обмежень:

- Ви можете множити дві матриці лише тоді, коли кількість стовпців у матриці зліва дорівнює кількості рядків у матриці справа.
- Множення матриць не є комутативним, тобто $A \cdot B \neq B \cdot A$.

Давайте розпочнемо з прикладу множення матриць 2 2x2:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Зараз, ймовірно, ви намагаєтеся розібратися, що тут тільки що відбулося? Множення матриць - це поєднання звичайного множення та додавання з використанням рядків матриці зліва та стовпців матриці справа. Давайте спробуємо розглянути це наступним малюнком:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Спочатку ми беремо верхній рядок лівої матриці, а потім беремо стовпець з правої матриці. Рядок і стовпець, які ми вибрали, визначають, яке значення виведемо в результаті розрахунку отриманої матриці розміром 2x2. Якщо ми беремо перший рядок лівої матриці, отримане значення потрапить в перший ряд результативної матриці, потім ми вибираємо стовпець, і якщо це перший стовпець, отримане значення потрапить в перший стовпець результативної матриці. Це відображено червоним шляхом. Для розрахунку значення внизу справа ми беремо нижній рядок першої матриці та праворучній стовпець другої матриці.

Для розрахунку отриманого значення ми множимо разом перший елемент рядка і стовпця, використовуючи звичайне множення, ми робимо те саме для других елементів, третіх, четвертих і так далі. Результати окремих множень потім додаються разом, і ми отримуємо наш результат. Тепер також стає зрозуміло, чому однією з вимог є те, що розмір стовпців лівої матриці і рядків правої матриці повинні бути рівними, інакше ми не можемо завершити операції!

Результат - це матриця розміром (n,m), де n дорівнює кількості рядків лівої матриці, а m дорівнює кількості стовпців правої матриці.

Не хвилюйтесь, якщо ви маєте труднощі уявити множення всередині своєї голови. Просто намагайтеся робити обчислення вручну і повертайтеся на цю сторінку, коли у вас виникнуть труднощі. З часом множення матриць стане для вас звичайним.

Завершимо обговорення множення матриць з більшим прикладом. Спробуйте візуалізувати взірець за допомогою кольорів. Як корисне вправа, спробуйте знайти свою власну відповідь на множення і порівняти її з отриманою матрицею (якщо ви спробуєте вручну виконати множення матриць, ви швидко зрозумієте, як це робити).

$$\begin{bmatrix} 4 & 2 & 0 \\ 0 & 8 & 1 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 4 & 2 & 1 \\ 2 & 0 & 4 \\ 9 & 4 & 2 \end{bmatrix} = \begin{bmatrix} 4 \cdot 4 + 2 \cdot 2 + 0 \cdot 9 & 4 \cdot 2 + 2 \cdot 0 + 0 \cdot 4 & 4 \cdot 1 + 2 \cdot 4 + 0 \cdot 2 \\ 0 \cdot 4 + 8 \cdot 2 + 1 \cdot 9 & 0 \cdot 2 + 8 \cdot 0 + 1 \cdot 4 & 0 \cdot 1 + 8 \cdot 4 + 1 \cdot 2 \\ 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 9 & 0 \cdot 2 + 1 \cdot 0 + 0 \cdot 4 & 0 \cdot 1 + 1 \cdot 4 + 0 \cdot 2 \end{bmatrix} = \begin{bmatrix} 20 & 8 & 12 \\ 25 & 4 & 34 \\ 2 & 0 & 4 \end{bmatrix}$$

Як бачите, множення матриць між собою - це досить незручний процес і дуже схильний до помилок (тому ми зазвичай дозволяємо це робити комп'ютерам), і це стає проблемою, коли матриці стають більшими. Якщо ви все ще хочете дізнатися більше і цікавитеся деякими іншими математичними властивостями матриць, я рекомендую переглянути ці відео Khan Academy про матриці.

Загалом, тепер, коли ми знаємо, як множити матриці, ми можемо перейти до цікавіших речей.

Множення матриці на вектор

До цього ми мали досить багато векторів. Ми використовували їх для представлення позицій, кольорів і навіть текстурних координат. Давайте рухатися трохи глибше в цю тему і скажемо вам, що вектор в основному є матрицею Nx1, де N - це кількість компонент вектора (також відомий як N-мірний вектор). Якщо подумати про це, це має сенс. Вектори, подібно до матриць, є масивом чисел, але з однією колонкою. Таким чином, як ця нова інформація допоможе нам? Ну, якщо у нас є матриця розміром MxN, ми можемо перемножити цю матрицю нашим вектором Nx1, оскільки кількість стовпців матриці дорівнює кількості рядків вектора, отже множення матриці визначено.

Але чому нам це цікаво, якщо ми можемо перемножити матрицю на вектор? Добре, виявляється, що існує багато цікавих 2D/3D перетворень, які ми можемо розмістити всередині матриці, і множення цієї матриці на вектор трансформує цей вектор. У разі, якщо ви все ще трохи плутані, давайте почнемо з декількох прикладів, і ви незабаром побачите, що маємо на увазі.

Матриця тотожності

У OpenGL ми зазвичай працюємо з 4x4 матрицями трансформацій з декількома причинами, і одна з них полягає в тому, що більшість векторів мають розмірність 4. Найпростіша матриця трансформації, яку ми можемо уявити, - це матриця тотожності. Матриця тотожності - це матриця NxN, в якій є лише 0, за винятком діагоналі. Як ви побачите, ця матриця трансформації не змінює вектор.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 \\ 1 \cdot 2 \\ 1 \cdot 3 \\ 1 \cdot 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

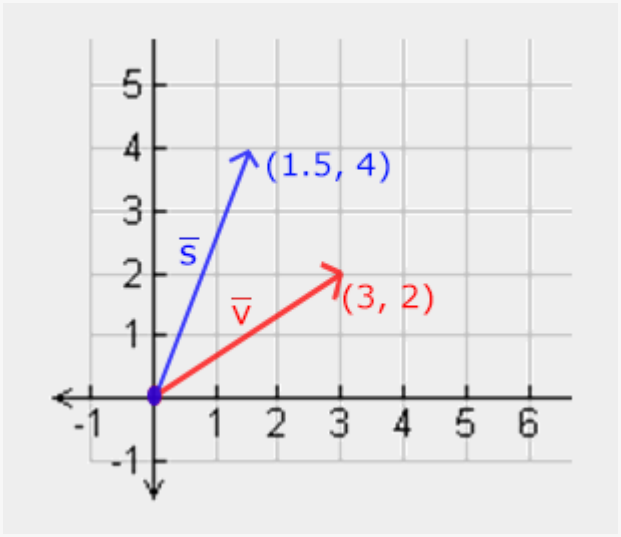
Вектор залишається абсолютно незмінним. Це стає очевидним з правил множення: перший елемент результату - це кожен окремий елемент першого рядка матриці, помножений на кожен елемент вектора. Оскільки всі елементи рядка дорівнюють 0, за винятком першого, ми отримуємо: $1 \cdot 1+0 \cdot 2+0 \cdot 3+0 \cdot 4=1$, і це ж саме стосується і до інших 3 елементів вектора.

Ви, можливо, запитуєтеся, яке використання трансформаційної матриці, яка не трансформує? Матриця тотожності, як правило, є початковою точкою для генерації інших матриць трансформацій. І якщо ми заглибимося ще глибше в лінійну алгебру, ця матриця виявляється дуже корисною для доведення теорем та розв'язання лінійних рівнянь.

Масштабування

Коли ми масштабуємо вектор, ми збільшуємо довжину стрілки на бажану величину, зберігаючи її напрям незмінним. Оскільки ми працюємо в 2 або 3 вимірах, ми можемо визначити масштабування за допомогою вектора з 2 або 3 масштабуючими змінними, кожна з яких масштабує одну зі шкал (x, y або z).

Спробуємо масштабувати вектор $\bar{v}=(3,2)$. Ми масштабуємо вектор вздовж вісі x в 0.5 рази, зробивши його вдвічі меншим, і масштабуємо вектор в 2 рази вздовж вісі y, роблячи його вдвічі вищим. Подивімося, як виглядає вектор після масштабування на (0.5,2) як $\bar{s}$:



Запам'ятайте, що OpenGL, як правило, працює в тривимірному просторі, тому для цього 2D випадку ми могли б встановити масштабування по вісі z рівним 1, залишивши його незмінним. Операція масштабування, яку ми щойно виконали, є нерівномірним масштабуванням, оскільки коефіцієнт масштабування не однаковий для кожної зі шкал. Якщо скаляр буде однаковий для всіх трьох осей, його називатимуть однаковим масштабуванням.

Давайте розпочнемо побудову матриці трансформації, яка робить масштабування для нас. Ми бачили з матриці тотожності, що кожен з діагональних елементів помножений на відповідний елемент вектора. Що, якщо б ми замінили одиниці в матриці тотожності на 3? В цьому випадку ми б помножили кожен з елементів вектора на значення 3 і, отже, ефективно масштабували б вектор на 3. Якщо ми представимо масштабуючі змінні як (S1, S2, S3), ми можемо визначити матрицю масштабування для будь-якого вектора (x, y, z) так:

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_1 \cdot x \\ S_2 \cdot y \\ S_3 \cdot z \\ 1 \end{pmatrix}$$

Зверніть увагу, що ми зберігаємо 4-е значення масштабування рівним 1. Компонент w використовується для інших цілей, які ми розглянемо пізніше.

Переміщення

Переміщення - це процес додавання іншого вектора до початкового вектора для отримання нового вектора з іншим положенням, тобто переміщення вектора на підставі вектора переміщення. Ми вже обговорили додавання векторів, тому це не повинно бути надто новим.

Точно так само, як і матриця масштабування, на 4x4 матриці є кілька місць, де ми можемо виконати певні операції, і для здійснення переміщення це верхні 3 значення 4-го стовпця. Якщо ми представимо вектор переміщення як (Tx, Ty, Tz), ми можемо визначити матрицю переміщення за допомогою наступного виразу:

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

Це працює тому, що всі значення переміщення множаться на стовпчик w вектора та додаються до початкових значень вектора (пам'ятайте правила множення матриць). Це було б неможливим з матрицею розміром 3 на 3.

Однорідні координати

Компонент w вектора також відомий як однорідна координата. Щоб отримати 3D вектор з однорідного вектора, ми ділимо координати x, y і z на його координату w. Зазвичай ми цього не помічаємо, оскільки компонент w майже завжди дорівнює 1,0. Використання однорідних координат має кілька переваг: це дозволяє нам виконувати матричні трансформації на 3D векторах (без компоненти w ми не можемо переміщати вектори), і в наступній главі ми використовуватимемо значення w для створення 3D-перспективи.

Крім того, коли однорідна координата дорівнює 0, вектор відомий як вектор напрямку, оскільки вектор з компонентою w рівною 0 не може бути переміщений.

За допомогою матриці переміщення ми можемо переміщувати об'єкти в будь-якому із трьох напрямів (x, y, z), що робить її дуже корисною матрицею трансформації в нашому наборі трансформаційних інструментів.

Обертання

Останні декілька перетворень були досить легкими для розуміння і візуалізації в просторі 2D або 3D, але обертання дещо складніше. Якщо вам потрібно точно знати, як ці матриці побудовані, я рекомендую переглянути розділи про обертання у відео з лінійної алгебри Khan Academy.

Спочатку давайте визначимо, що насправді представляє обертання вектора. Обертання в 2D або 3D представлене кутом. Кут може бути в градусах або радіанах, де вся кругова рівнина має 360 градусів або 2 PI радіана. Я більше вподобовую пояснювати обертання за допомогою градусів, оскільки ми загалом більше звикли до них.

Більшість функцій обертання вимагають кута в радіанах, але на щастя градуси легко перетворюються в радіани:

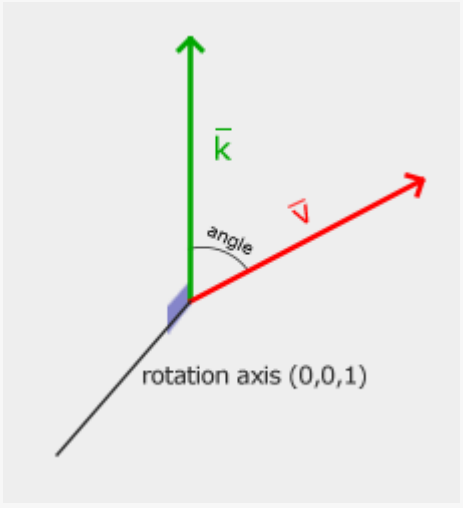
кут в градусах = кут в радіанах * (180 / PI)

кут в радіанах = кут в градусах * (PI / 180)

де PI дорівнює (заокруглено) 3.14159265359.

Обертаючи наполовину круга, ми обертаємо на 360/2 = 180 градусів, і обертаючи наполовину вправо, ми обертаємо на 360/5 = 72 градуси вправо. Це демонструється на прикладі базового 2D вектора, де $\bar{v}$

обертається на 72 градуси вправо, або за годинниковою стрілкою, відносно $\bar{k}$.



Обертання в 3D визначаються кутом та віслю обертання. Зазначений кут буде обертати об'єкт вздовж заданої вісі обертання. Спробуйте візуалізувати це, обертаючи голову на певний кут і при цьому постійно дивлячись вниз по одній вісі обертання. Наприклад, коли обертаєте 2D вектори в 3D світі, ви встановлюєте вісь обертання вздовж вісі z (спробуйте візуалізувати це).

З використанням тригонометрії можна перетворити вектори в нові обертані вектори за допомогою кута. Зазвичай це робиться шляхом розумного поєднання функцій синус і косинус (зазвичай скорочено sin і cos). Обговорення того, як генеруються матриці обертання, виходить за межі цього розділу.

Матриця обертання визначена для кожної одиниці в просторі 3D, де кут представлений символом тета θ.

Rotation around the X-axis:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ \cos\theta \cdot y - \sin\theta \cdot z \\ \sin\theta \cdot y + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

Rotation around the Y-axis:

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x + \sin\theta \cdot z \\ y \\ -\sin\theta \cdot x + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

Rotation around the Z-axis:

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x - \sin\theta \cdot y \\ \sin\theta \cdot x + \cos\theta \cdot y \\ z \\ 1 \end{pmatrix}$$

За допомогою матриць обертання ми можемо перетворити наші вектори положення навколо однієї з трьох одиниць вісей. Щоб обертати навколо довільної 3D вісі, ми можемо поєднувати всі три обертання, спершу навколо вісі X, потім Y і потім Z, наприклад. Проте це швидко призводить до проблеми, яка називається блокуванням Гімбала. Ми не будемо обговорювати деталі, але кращим рішенням є обертання навколо довільної одиничної вісі, наприклад, (0,662,0,2,0,722) (зауважте, що це одиничний вектор), нате ще при використанні відповідної матриці:

[Rx,Ry,Rz]

де Rx, Ry, Rz - кути обертання навколо довільної вісі.

$$\begin{bmatrix} \cos\theta + {R_x}^2(1-\cos\theta) & R_xR_y(1-\cos\theta) - R_z\sin\theta & R_xR_z(1-\cos\theta) + R_y\sin\theta & 0 \\ R_yR_x(1-\cos\theta) + R_z\sin\theta & \cos\theta + {R_y}^2(1-\cos\theta) & R_yR_z(1-\cos\theta) - R_x\sin\theta & 0 \\ R_zR_x(1-\cos\theta) - R_y\sin\theta & R_zR_y(1-\cos\theta) + R_x\sin\theta & \cos\theta + {R_z}^2(1-\cos\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Математичний обговорення генерації такої матриці виходить за рамки цього розділу. Запам'ятайте, що навіть ця матриця не повністю усуває блокування Гімбала (навіть якщо це стає значно складніше). Щоб дійсно усунути блокування Гімбала, ми повинні представляти обертання за допомогою кватерніонів, які не лише безпечніше, але й більш видають обчислювано дружніми. Проте обговорення кватерніонів виходить за рамки цього розділу.

Поєднання матриць

Справжня сила використання матриць для перетворень полягає в тому, що ми можемо поєднувати кілька перетворень в одній матриці завдяки множенню матриць на матриці. Давайте подивимося, чи можемо ми згенерувати матрицю перетворення, яка поєднує декілька перетворень. Скажімо, ми маємо вектор (x, y, z) і хочемо змінити його розмір у 2 рази, а потім перемістити його на (1, 2, 3). Нам потрібна матриця перетворення для наших необхідних дій. Результуюча матриця перетворення буде виглядати так:

$$Trans.Scale = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Зверніть увагу, що спочатку ми робимо перетворення зсуву, а потім масштабування при множенні матриць. Множення матриць не комутативне, що означає, що їх порядок має важливе значення. При множенні матриць праву матрицю спочатку множать на вектор, тому ви повинні читати множення з права наліво. Рекомендується спочатку виконувати операції масштабування, потім обертання і, нарешті, зсуви при об'єднанні матриць, інакше вони можуть (негативно) впливати одне на одне. Наприклад, якщо ви спершу зробите зсув, а потім масштабування, вектор зсуву також піддасться масштабуванню!

Виконання остаточної матриці перетворення нашого вектора призводить до наступного результату:

$$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 2x+1 \\ 2y+2 \\ 2z+3 \\ 1 \end{bmatrix}$$

Відмінно! Вектор спочатку масштабується у два рази, а потім зсувається на (1,2,3).

Практика

Тепер, коли ми пояснили всю теорію перетворень, настав час побачити, як ми можемо використовувати ці знання на користь нашого проекту. У бібліотеці OpenGL немає жодних вбудованих засобів для матриць або векторів, тому нам потрібно визначити власні математичні класи та функції. У цій книзі ми більше б хотіли відсторонитися від всіх дрібних математичних деталей і просто використовувати готові математичні бібліотеки. На щастя, існує легко використовувана і спеціально адаптована для OpenGL бібліотека математики під назвою GLM.

GLM

GLM означає OpenGL Mathematics і є бібліотекою, яка складається лише з заголовочних файлів, що означає, що нам потрібно лише включити відповідні заголовочні файли, і це все; немає необхідності в посиланні та компіляції. GLM можна завантажити з їхнього веб-сайту. Скопіюйте кореневий каталог заголовочних файлів в каталог з інклюдами і давайте рушати вперед.

Більшість функціональності GLM, яку нам потрібно, можна знайти в трьох заголовках, які ми включимо таким чином:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
```

Давайте перевіримо, чи можемо використовувати наші знання про трансформації, щоб перенести вектор (1,0,0) на (1,1,0) (зауважте, що ми визначаємо його як glm::vec4 із встановленою однорідною координатою 1.0):

```cpp
glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
vec = trans * vec;
std::cout << vec.x << vec.y << vec.z << std::endl;
```

Спочатку ми визначаємо вектор з ім'ям vec, використовуючи вбудований в GLM клас векторів. Далі ми визначаємо mat4 і явно ініціалізуємо його як одиничну матрицю, ініціалізуючи діагоналі матриці значенням 1.0. Якщо ми не ініціалізуємо її як одиничну матрицю, то матриця буде нульовою матрицею (усі елементи 0), і всі наступні операції з матрицею також закінчаться нульовою матрицею.

Наступним кроком є створення матриці трансформації, передаючи нашу одиничну матрицю функції glm::translate разом із вектором трансляції (дана матриця потім множиться на матрицю трансляції, і отримана матриця повертається).

Потім ми множимо наш вектор на матрицю трансформації та виводимо результат. Якщо ми все ще пам'ятаємо, як працює матрична трансляція, то отриманий вектор повинен бути (1+1,0+1,0+0), що дорівнює (2,1,0). Цей фрагмент коду виводить 210, так що матриця трансляції виконала свою роботу.

Давайте зробимо щось цікавіше і масштабувати та обертати об'єкт контейнера з попереднього розділу.

```
glm::mat4 trans = glm::mat4(1.0f);

trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));

trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));
```

Спочатку ми масштабуємо контейнер на 0.5 по кожній з осей, а потім обертаємо контейнер на 90 градусів навколо вісі Z. GLM очікує кутові значення в радіанах, тому ми перетворюємо градуси в радіани за допомогою glm::radians. Зауважте, що текстурований прямокутник знаходиться в площині XY, тому ми хочемо обертати навколо вісі Z. Пам'ятайте, що вісь, навколо якої ми обертаємося, повинна бути одиничним вектором, тому будьте впевнені, що нормалізуєте вектор, якщо ви не обертаєтеся навколо вісей X, Y або Z. Оскільки ми передаємо матрицю кожній з функцій GLM, GLM автоматично множить матриці між собою, що призводить до матриці трансформації, яка об'єднує всі трансформації.

Наступним важливим питанням є: як ми передаємо матрицю трансформації у шейдери? Ми вже згадували раніше, що у GLSL також є тип mat4. Таким чином, ми адаптуємо вершинний шейдер для прийому матриці mat4 як змінну uniform і множимо вектор позиції на цю матрицю uniform:

```
#version 330 core

layout (location = 0) in vec3 aPos;

layout (location = 1) in vec2 aTexCoord;
```

```
out vec2 TexCoord;

uniform mat4 transform;

void main()

{

    gl_Position = transform * vec4(aPos, 1.0f);

    TexCoord = vec2(aTexCoord.x, aTexCoord.y);

}
```

GLSL також має типи mat2 і mat3, які дозволяють виконувати операції, схожі на swizzling, так само, як у векторів. Усі вищезазначені математичні операції (наприклад, множення скаляра на матрицю, множення матриці на вектор і множення матриць на матрицю) дозволені для типів матриць. Де би ми не використовували спеціальні операції над матрицями, ми обов'язково пояснимо, що відбувається.

Ми додали uniform і помножили вектор позиції на матрицю трансформації перед передачею його в gl_Position. Наш контейнер тепер має бути вдвічі меншим і обернутим на 90 градусів (нахиленим вліво). Однак нам все ще потрібно передати матрицю трансформації в шейдер:

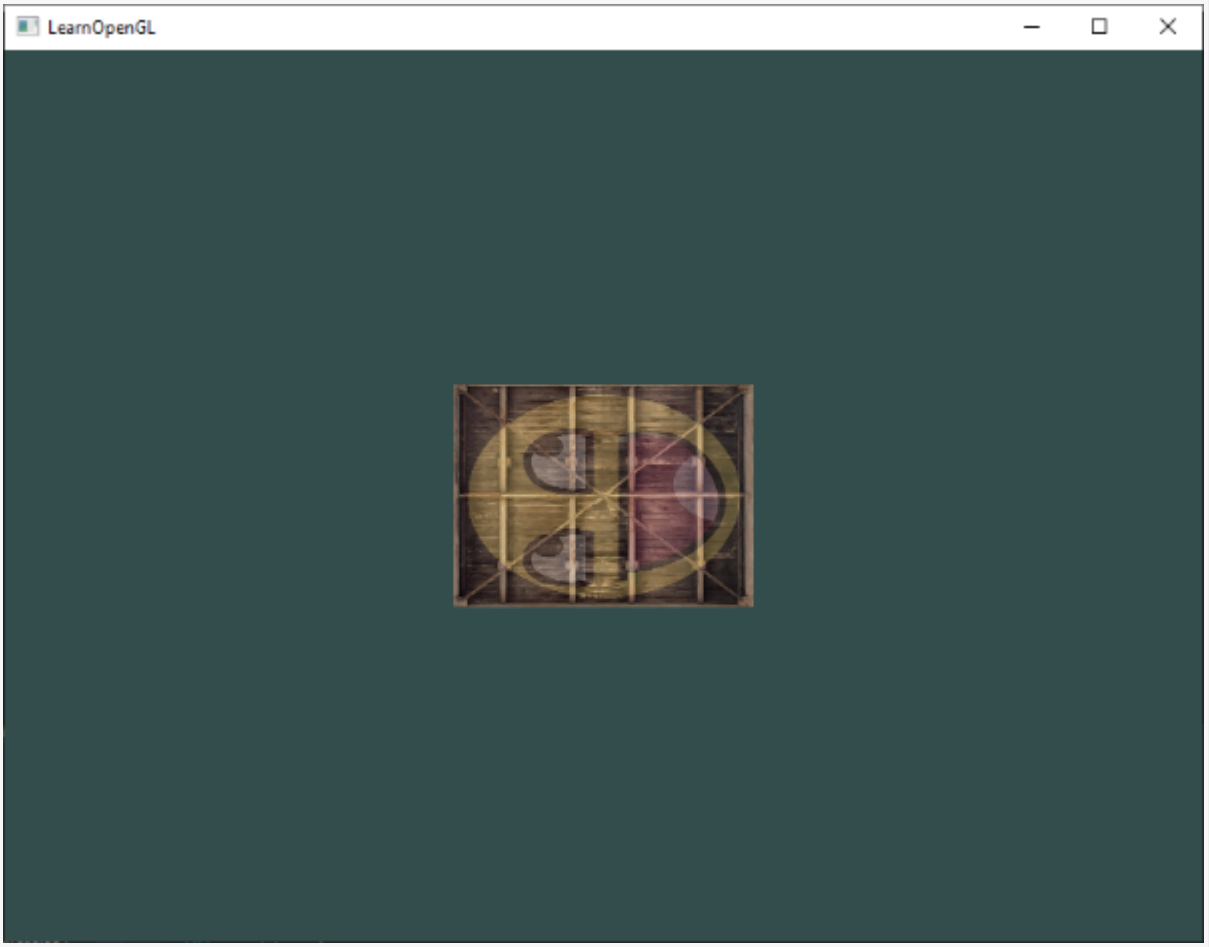
```
unsigned int transformLoc = glGetUniformLocation(ourShader.ID, "transform");

glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(trans));
```

Ми спочатку запитуємо розташування змінної uniform і потім надсилаємо дані матриці в шейдери за допомогою glUniform з постфіксом Matrix4fv. Перший аргумент вже має вам бути знайомим, це розташування uniform. Другий аргумент говорить OpenGL, скільки матриць ми хочемо відправити, це 1. Третій аргумент питає нас, чи бажаємо ми переставити нашу матрицю, тобто поміняти місцями стовпці та рядки. Розробники OpenGL часто використовують внутрішню структуру матриць, викликану column-major ordering, яка є стандартною структурою матриць у GLM, тому немає потреби переставляти матриці; ми можемо залишити це на GL_FALSE. Останнім параметром є фактичні дані матриці, але GLM зберігає дані своїх матриць так, що вони не завжди відповідають очікуванням OpenGL, тому спочатку ми конвертуємо дані за допомогою вбудованої функції GLM value_ptr.

Ми створили матрицю трансформації, оголосили uniform в вершинному шейдері і відправили матрицю в шейдери, де ми трансформували координати вершин. Результат має виглядати приблизно так:

![Результат обертання і масштабування](https://learnopengl.com/img/transformations_basic.png)



Чудово! Наш контейнер дійсно нахилений вліво і вдвічі зменшений, отже, трансформація пройшла успішно. Давайте станемо трохи функціональнішими і подивимося, чи можемо маючи надто часу, обертати контейнер, і для розваги ми також перепозиціонуємо контейнер в нижньому правому куті вікна. Для того, щоб обертати контейнер протягом певного часу, нам потрібно оновлювати матрицю трансформації в циклі рендерингу, оскільки вона повинна оновлюватися кожен кадр. Ми використовуємо функцію часу GLFW для отримання кута з плином часу:

```
glm::mat4 trans = glm::mat4(1.0f);

trans = glm::translate(trans, glm::vec3(0.5f, -0.5f, 0.0f));

trans = glm::rotate(trans, (float)glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));
```

Зауважте, що в попередньому випадку ми могли оголосити матрицю трансформації будь-де, але тепер нам потрібно створити її кожної ітерації, щоб постійно оновлювати

обертання. Це означає, що ми повинні перестворювати матрицю трансформації в кожній ітерації циклу рендерингу. Зазвичай під час відтворення сцен ми маємо декілька матриць трансформацій, які перестворюються з новими значеннями кожен кадр.

Ось спочатку ми обертаємо контейнер навколо початкової точки (0,0,0), а як тільки він обертається, ми пересуваємо його обернену версію в нижній правий кут екрану. Пам'ятайте, що порядок трансформацій повинен бути зчитаним зворотно: навіть якщо в коді ми спочатку перекладаємо, а потім обертаємо, фактичні трансформації спочатку застосовують обертання, а потім переклад. Зрозуміти всі ці комбінації трансформацій і те, як вони застосовуються до об'єктів, складно. Спробуйте та експериментуйте з такими трансформаціями, і ви швидко зрозумієте це.

Якщо ви все зробили правильно, то повинні отримати наступний результат:

І ось вам результат. Контейнер, який переноситься та обертається з часом, все це виконується однією матрицею трансформації! Тепер ви можете бачити, чому матриці є такою потужною конструкцією в області графіки. Ми можемо визначити нескінченну кількість трансформацій і об'єднати їх у одній матриці, яку ми можемо використовувати скільки завгодно разів. Використання трансформацій, подібних до цієї у вершинному шейдері, зберігає нас від зусиль перевизначення даних вершин і також економить наш час обробки, оскільки нам не потрібно надсилати дані кожного разу (що досить повільно); все, що нам потрібно, це оновити матрицю трансформації.

Якщо ви не отримали належного результату або ви застрягли в іншому місці, перегляньте вихідний код і оновлений клас шейдера .

У наступному розділі ми обговоримо, як ми можемо використовувати матриці для визначення різних координатних просторів для наших вершин. Це буде наш перший крок до 3D-графіки!

Додаткова література

- Essence of Linear Algebra : серія чудових відеоуроків від Гранта Сандерсона про основну математику перетворень і лінійну алгебру.

Вправи

- Використовуючи останню трансформацію в контейнері, спробуйте змінити порядок, спочатку обертаючи, а потім переводячи. Подивіться, що відбувається, і спробуйте зрозуміти, чому це відбувається: рішення .
- Спробуйте намалювати другий контейнер з іншим викликом `glDrawElements` але розмістіть його в іншому місці, використовуючи **лише** перетворення . Переконайтеся, що цей другий контейнер розміщено у верхньому лівому куті вікна, і замість того, щоб обертати, масштабуйте його з часом (`sin` тут корисно використовувати функцію; зауважте, що використання `sin` спричинить інвертування об’єкта, щойно застосовано від’ємний масштаб) : рішення .