Лекція 7

Лабораторна робота 7

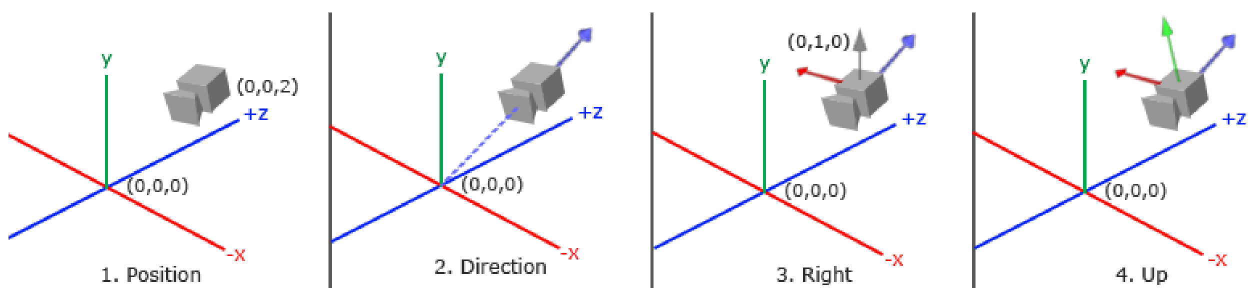
# Камера

У попередньому розділі ми обговорювали матрицю огляду та те, як ми можемо використовувати її для переміщення по сцені (ми трошки рухалися назад). OpenGL сам по собі не знає про концепцію камери, але ми можемо намагатися симулювати її, переміщуючи всі об'єкти на сцені в зворотньому напрямку, створюючи ілюзію руху.

У цьому розділі ми розглянемо, як ми можемо налаштувати камеру в OpenGL. Ми розглянемо камеру у стилі "літаючої", яка дозволяє вам вільно переміщатися в тривимірній сцені. Ми також обговоримо введення з клавіатури та миші, і завершимо створенням власного класу камери.

## Простір камери/огляду

Коли ми говоримо про простір камери/огляду, ми маємо на увазі всі координати вершин, які бачить камера з її точки зору як початку сцени: матриця огляду перетворює всі світові координати в координати огляду, які є відносно положення та напрямку камери. Для визначення камери нам потрібна її позиція в світовому просторі, напрямок, в якому вона дивиться, вектор, що вказує вправо, і вектор, що вказує вгору від камери. Уважний читач може помітити, що фактично ми створюємо систему координат із трьома перпендикулярними одиничними вісями, де позиція камери є початком.




## Позиція камери

Отримати позицію камери легко. Позиція камери - це вектор у світовому просторі, який вказує на позицію камери. Ми встановлюємо камеру в тій самій позиції, що і в попередньому розділі:

```
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
```

Не забувайте, що позитивна вісь z проходить через ваш екран до вас, тому якщо ми хочемо, щоб камера рухалася назад, ми рухаємося вздовж позитивної вісі z.

## Напрямок камери

Наступний необхідний вектор - це напрямок камери, тобто в якому напрямку вона дивиться. Наразі ми дозволяємо камері вказувати на початок нашої сцени: (0,0,0). Пам'ятайте, що якщо ми віднімемо один вектор від іншого, ми отримаємо вектор, який є різницею цих двох векторів? Віднімання вектора позиції камери від вектора початку сцени призводить до вектора напрямку, який нам потрібен. Для системи координат матриці огляду ми хочемо, щоб її вісь z була позитивною, і, враховуючи конвенцію (в OpenGL), камера спрямовується в напрямку негативної вісі z, тому ми хочемо інвертувати вектор напрямку. Якщо ми змінимо порядок віднімання, отримаємо вектор, який вказує на позитивну вісь z камери:

```
glm::vec3 cameraTarget = glm::vec3(0.0f, 0.0f, 0.0f);
glm::vec3 cameraDirection = glm::normalize(cameraPos - cameraTarget);
```

Назва "вектор напрямку" не є найкраще обраною, оскільки фактично він вказує у зворотньому напрямку від того, на що він спрямований.

## Вектор правої вісі

Наступний вектор, який нам потрібен, - це вектор правої вісі, який представляє позитивну вісь x у просторі камери. Щоб отримати вектор правої вісі, ми використовуємо трохи хитрий прийом, спочатку вказавши вектор, що вказує вгору (у світовому просторі). Потім ми виконуємо векторний добуток між вектором, що вказує вгору, та вектором напрямку з кроку 2. Оскільки результат векторного добутку - це вектор, перпендикулярний обом векторам, ми отримаємо вектор, що вказує в напрямку позитивної вісі x (якщо ми

змінимо порядок векторного добутку, отримаємо вектор, що вказує в напрямку негативної вісі x).

```
glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);

glm::vec3 cameraRight = glm::normalize(glm::cross(up, cameraDirection));
```

## Вісь вгору

Тепер, коли у нас є вектор вісі x і вектор вісі z, отримати вектор, що вказує на позитивну вісь y камери, досить легко: ми беремо векторний добуток правого та напрямного векторів:

cpp

Copy code

```
glm::vec3 cameraUp = glm::cross(cameraDirection, cameraRight);
```

За допомогою векторного добутку та кількох хитрощів нам вдалося створити всі вектори, які формують простір огляду/камери. Для більш математично нахилених читачів цей процес відомий як процес Грама-Шмідта в лінійній алгебрі. З використанням цих векторів камери ми тепер можемо створити матрицю LookAt, яка дуже корисна для створення камери.

# Look At

Чудова річ у матрицях полягає в тому, що, якщо ви визначите простір координат за допомогою 3 перпендикулярних (або нелінійних) вісей, ви можете створити матрицю з цими 3 вісями плюс вектором перекладу, і ви можете перетворити будь-який вектор до цього простору координат, перемноживши його на цю матрицю. Це точно те, що робить матриця LookAt, і тепер, коли у нас є 3

перпендикулярні вісі та вектор позиції для визначення простору камери, ми можемо створити свою власну матрицю LookAt:

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

cpp

Copy code

```cpp
// Код для створення матриці LookAt

glm::mat4 viewMatrix;



// Визначення векторів R, U, D та P

glm::vec3 R = cameraRight;

glm::vec3 U = cameraUp;

glm::vec3 D = -cameraDirection; // Увага: вектор напрямку інвертується для
правильного визначення

glm::vec3 P = cameraPos;



// Заповнення матриці LookAt

viewMatrix[0][0] = R.x;
```

```
viewMatrix[1][0] = R.y;

viewMatrix[2][0] = R.z;

viewMatrix[0][1] = U.x;

viewMatrix[1][1] = U.y;

viewMatrix[2][1] = U.z;

viewMatrix[0][2] = D.x;

viewMatrix[1][2] = D.y;

viewMatrix[2][2] = D.z;

viewMatrix[3][0] = -glm::dot(R, P);

viewMatrix[3][1] = -glm::dot(U, P);

viewMatrix[3][2] = glm::dot(D, P);

viewMatrix[3][3] = 1.0f;
```

Де R - вектор правої вісі, U - вектор вгору, D - вектор напрямку, а P - вектор позиції камери. Зверніть увагу, що частини обертання (ліва матриця) та трансляції (права матриця) інвертовані (транспоновані та негативовані відповідно), оскільки ми хочемо обертати та транслювати світ в протилежному напрямку того, куди ми хочемо переміщати камеру. Використовуючи цю матрицю LookAt як нашу матрицю огляду, ми ефективно перетворюємо всі світові координати в простір огляду, який ми тільки що визначили. Матриця LookAt робить те, що вона обіцяє: створює матрицю огляду, яка дивиться на задану ціль.

На щастя для нас, GLM вже виконує всю цю роботу. Нам просто потрібно вказати позицію камери, позицію цілі та вектор, який представляє вектор вгору в світовому просторі (вектор вгору, який ми використовували для обчислення вектора правої вісі). Після цього GLM створює матрицю LookAt, яку ми можемо використовувати як нашу матрицю виду.

```
glm::mat4 view;

view = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f),

              glm::vec3(0.0f, 0.0f, 0.0f),

              glm::vec3(0.0f, 1.0f, 0.0f));
```

Функція glm::LookAt вимагає вказівки на позицію, ціль та вектор "вгору" відповідно. У цьому прикладі створюється матриця огляду, яка ідентична з тією, яку ми створили в попередньому розділі.

Перш ніж займатися користувацьким введенням, давайте трошки розважимося, обертаючи камеру навколо нашої сцени. Ми тримаємо ціль сцени на позначці (0,0,0). Ми використовуємо трошки тригонометрії, щоб кожного кадру створювати x- та z-координати, які представляють точку на колі, і використовуємо їх для позиції нашої камери. Перераховуючи координати x та z з плином часу, ми обходимо всі точки на колі, і, отже, камера обертається навколо сцени. Ми збільшуємо цей коло на заздалегідь визначений радіус і кожен кадр створюємо нову матрицю огляду за допомогою функції glfwGetTime з GLFW:

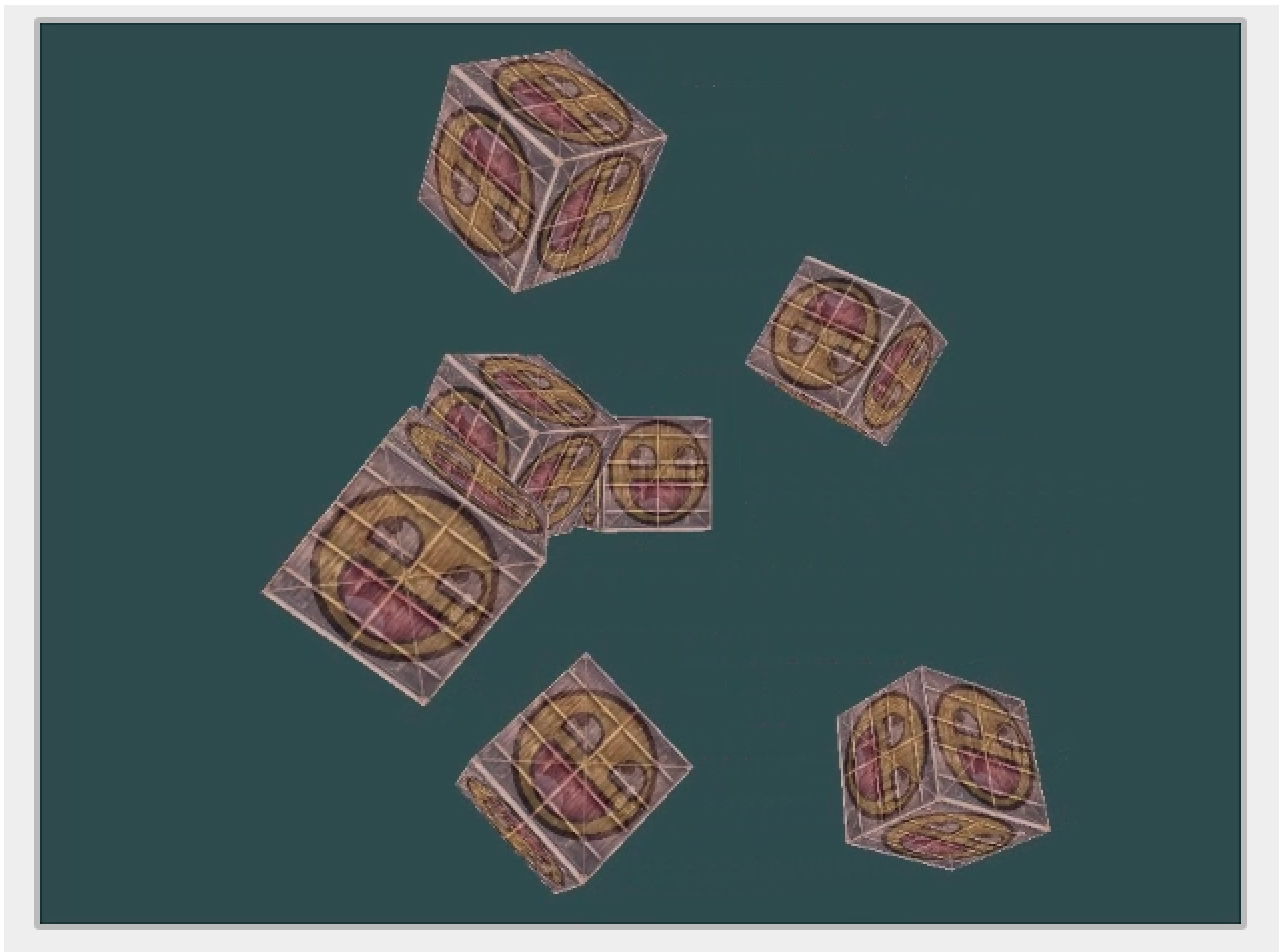
cpp

Copy code

```
const float radius = 10.0f;

float camX = sin(glfwGetTime()) * radius;

float camZ = cos(glfwGetTime()) * radius;

glm::mat4 view;

view = glm::lookAt(glm::vec3(camX, 0.0, camZ), glm::vec3(0.0, 0.0, 0.0),
glm::vec3(0.0, 1.0, 0.0));
```

Якщо ви запустите цей код, ви повинні отримати щось подібне до цього:

[https://learnopengl.com/video/getting-started/camera_circle.mp4](https://learnopengl.com/video/getting-started/camera_circle.mp4)



За допомогою цього короткого уривка коду камера тепер обходить сцену з плином часу. Не соромтеся експериментувати з параметрами радіуса та позиції/напрямком, щоб зрозуміти, як працює ця матриця LookAt. Також перевірте вихідний код, якщо у вас виникли труднощі.

## Пересування навколо

Обертання камери навколо сцени цікаво, але це ще цікавіше робити всі рухи самостійно! Спочатку нам потрібно налаштувати систему камери, тому корисно визначити деякі змінні камери у верхній частині нашої програми:

```cpp
glm::vec3 cameraPos   = glm::vec3(0.0f, 0.0f,  3.0f);

glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);

glm::vec3 cameraUp    = glm::vec3(0.0f, 1.0f,  0.0f);
```

Функція LookAt тепер стає такою:

```cpp
view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
```

Спочатку ми встановлюємо позицію камери на попередньо визначене значення cameraPos. Напрямок - це поточна позиція + вектор напрямку, який ми щойно визначили. Це забезпечує, що незалежно від того, як ми рухаємося, камера продовжує дивитися в напрямок цільового напрямку. Давайте трошки поекспериментуємо з цими змінними, оновлюючи вектор cameraPos, коли ми натискатимемо деякі клавіші.

Ми вже визначили функцію processInput для обробки клавіш клавіатури GLFW, так що додамо кілька додаткових команд клавіш:

```cpp
void processInput(GLFWwindow *window)
{
    ...
    const float cameraSpeed = 0.05f; // відкоригуйте за необхідності
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        cameraPos += cameraSpeed * cameraFront;
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        cameraPos -= cameraSpeed * cameraFront;
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
```

```
        cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;

    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)

        cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;

}
```

Кожного разу, коли ми натискатимемо одну з клавіш WASD, позиція камери відповідно оновлюватиметься. Якщо ми хочемо рухатися вперед або назад, ми додаємо або віднімаємо вектор напрямку від вектора позиції, помножений на деяке значення швидкості. Якщо ми хочемо рухатися в бічному напрямку, ми робимо векторне множення, щоб створити правий вектор, і рухаємося вздовж правого вектора відповідно. Це створює знайому ефект сторінкування при використанні камери.

Зверніть увагу, що ми нормалізуємо отриманий правий вектор. Якщо ми не нормалізуємо цей вектор, отриманий векторний добуток може повертати вектори різного розміру в залежності від змінної cameraFront. Якщо ми не нормалізуємо вектор, ми будемо рухатися повільно чи швидко в залежності від орієнтації камери, а не зі сталою швидкістю руху.

На цьому етапі ви вже повинні вміти пересувати камеру, проте це може залежати від конкретного обладнання, тому вам може знадобитися налаштувати параметр cameraSpeed.

## ШВИДКІСТЬ РУХУ

Наразі ми використовуємо константне значення швидкості руху під час пересування. Теоретично це виглядає добре, але на практиці у різних комп'ютерів у людей різна потужність обробки, і результат полягає в тому, що одні люди можуть відтворювати набагато більше кадрів за секунду, ніж інші. Коли користувач відтворює більше кадрів, він також частіше викликає функцію processInput. Результат полягає в тому, що одні люди рухаються дуже швидко, а інші дуже повільно в залежності від їхнього обладнання. При вивантаженні вашого застосунку ви хочете впевнитися, що він працює однаково на різних обладнаннях.

Графічні застосунки та ігри зазвичай ведуть облік змінної deltatime, яка зберігає час, який був витрачений на відтворення останнього кадру. Потім всі швидкості множаться на це значення deltaTime. Результат полягає в тому, що коли маємо велике значення deltaTime у кадрі, що означає, що останній кадр тривав довше за середній, швидкість для цього кадру також буде трошки вищою, щоб збалансувати все це. При використанні цього підходу не важливо, чи у вас дуже швидкий чи повільний комп'ютер, швидкість камери буде вирівнюватися відповідно, і кожен користувач отримає однаковий досвід.

Для обчислення значення deltaTime ми ведемо облік двох глобальних змінних:

```cpp
float deltaTime = 0.0f;    // Час між поточним кадром та попереднім кадром

float lastFrame = 0.0f;    // Час останнього кадру
```

```

У кожному кадрі ми обчислюємо нове значення deltaTime для подальшого використання:

```cpp
float currentFrame = glfwGetTime();
deltaTime = currentFrame - lastFrame;
lastFrame = currentFrame;
```

Тепер, коли у нас є deltaTime, ми можемо враховувати його при обчисленні швидкостей:

```cpp
void processInput(GLFWwindow *window)
{
    float cameraSpeed = 2.5f * deltaTime;
    [...]
}

```
```

Оскільки ми використовуємо deltaTime, камера тепер буде рухатися зі сталою швидкістю 2.5 одиниць на секунду. Разом із попереднім розділом у нас тепер повинна бути набагато більш плавна та послідовна система камери для руху навколо сцени.

І тепер у нас є камера, яка рухається та дивиться однаково швидко на будь-якій системі. Знову перевірте вихідний код, якщо у вас виникають труднощі. Ви часто будете бачити значення deltaTime в усьому, що пов'язане з рухом.

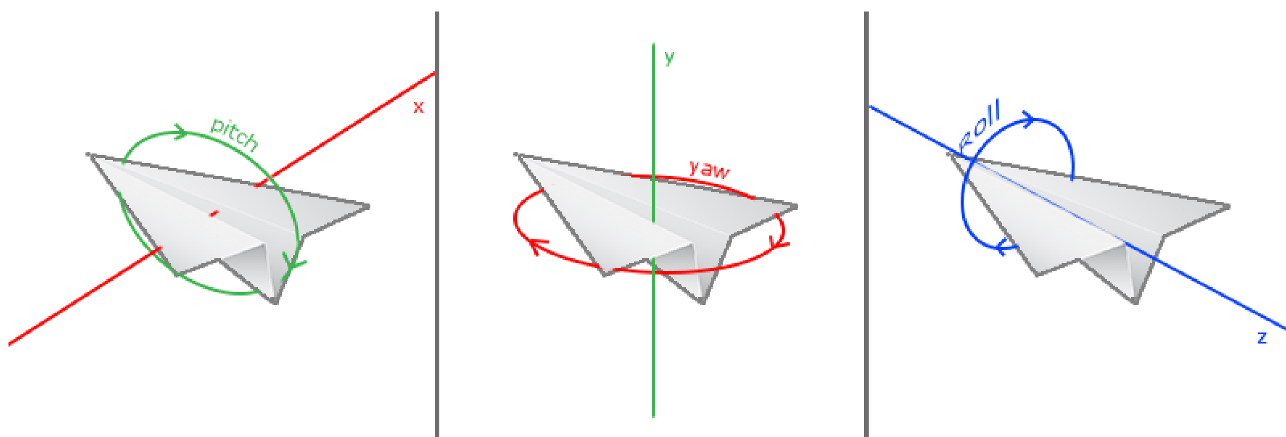
## Огляд навколо

Лише використання клавіш клавіатури для пересування навколо не є дуже цікавим. Особливо, оскільки ми не можемо повертатися, рух є досить обмеженим. Ось де входить миша!

Щоб дивитися навколо сцени, нам потрібно змінити вектор cameraFront на основі введення миші. Однак зміна вектора напрямку на основі обертання миші трошки складніша і вимагає трохи тригонометрії. Якщо ви не розумієте тригонометрію, не переймайтеся, ви можете просто пропустити до розділів коду і вставити їх у свій код; ви завжди можете повертатися назад, якщо захочете дізнатися більше.

## Кути Ейлера

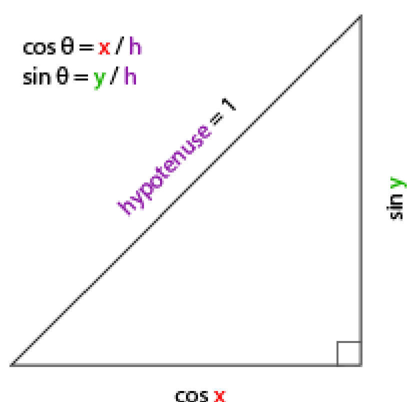
Кути Ейлера - це 3 значення, які можуть представляти будь-яке обертання в 3D, визначене Леонардом Ейлером десь у 1700-х роках. Є 3 кути Ейлера: *pitch*, *yaw* and *roll*. Наступне зображення надає їм візуальне значення:




Кут крена - це кут, який показує, наскільки ми дивимося вгору чи вниз, як видно на першому зображенні. Друге зображення показує значення рискання, яке представляє величину, на яку ми дивимося ліворуч чи праворуч. Рулон визначає, наскільки ми кочуємо, головним чином використовуючи камери для космічних польотів. Кожен з кутів Ейлера представлений одним значенням, і з комбінацією всіх трьох ми можемо розрахувати будь-який вектор обертання в 3D.

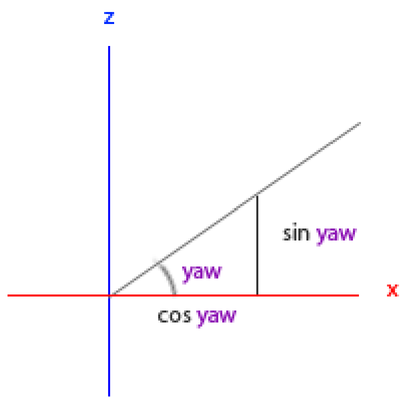
Для нашої системи камери нас цікавлять тільки значення рискання і крена, тому ми тут не будемо обговорювати значення рулону. З заданими значеннями рискання та крена ми можемо конвертувати їх у 3D-вектор, що представляє новий вектор напрямку. Процес конвертації значень рискання та крена в вектор напрямку вимагає трошки тригонометрії, і ми розпочинаємо з базового випадку:

Почнемо з невеликого оновлення та перевірки загального випадку прямокутного трикутника (із одним боком під кутом 90 градусів):




Якщо ми визначимо гіпотенузу довжиною 1, ми знаємо з тригонометрії (soh cah toa), що довжина прилеглої сторони - це cos x/h = cos x/1 = cos x, і що довжина протилежної сторони - це sin y/h = sin y/1 = sin y. Це дає нам загальні формули для отримання довжини як прилеглої, так і протилежної сторін прямокутного трикутника, в залежності від заданого кута. Давайте використовувати це для розрахунку компонентів вектора напрямку.

Представимо цей самий трикутник, але тепер дивимося на нього з верхнього погляду, коли прилеглі та протилежні сторони паралельні вісі x та z сцени (ніби дивимося вздовж вісі y).

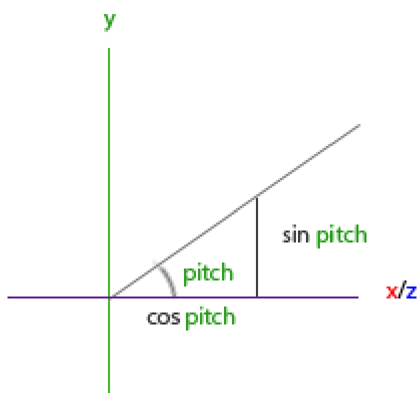



Якщо ми візуалізуємо кут рискання як кут проти годинникової стрілки, починаючи зі сторони x, ми можемо побачити, що довжина сторони x відноситься до cos(yaw). І подібно тому, як довжина сторони z відноситься до sin(yaw).

Якщо ми використовуємо це знання та задане значення рискання, ми можемо використовувати його для створення вектора напрямку камери:

```cpp
glm::vec3 direction;

direction.x = cos(glm::radians(yaw)); // Зверніть увагу, що ми конвертуємо кут в радіани спочатку

direction.z = sin(glm::radians(yaw));
```

Це вирішує, як ми можемо отримати 3D вектор напрямку зі значенням рискання, але треба включити також крен. Тепер давайте розглянемо сторону вісі y, ніби ми сидимо на площині xz:




Аналогічно, з цього трикутника ми бачимо, що компонент y вектора напрямку дорівнює sin(pitch), отже, давайте заповнимо це:

```cpp
direction.y = sin(glm::radians(pitch));
```

Однак з трикутника pitch ми також бачимо, що сторони xz впливають на cos(pitch), тому ми повинні впевнитися, що це також є частиною вектора напрямку. З цим включеним ми отримуємо кінцевий вектор напрямку, перекладений з кутів Ейлера рискання та крена:

```cpp
```

direction.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));

direction.y = sin(glm::radians(pitch));

direction.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));

```

Це дає нам формулу для конвертації значень рискання та крена в 3-вимірний вектор напрямку, який ми можемо використовувати для обертання.

Ми встановили світ сцени так, що все розташовано в напрямку від'ємної вісі z. Однак, якщо ми розглянемо трикутник рискання x та z, то побачимо, що θ 0 призводить до того, що вектор напрямку камери вказує на позитивну вісь x. Щоб забезпечити, що камера за замовчуванням вказує на від'ємну вісь z, ми можемо встановити значення рискання за замовчуванням як обертання за годинниковою стрілкою на 90 градусів. Позитивні градуси обертаються проти годинникової стрілки, тому ми встановлюємо значення рискання за замовчуванням на:

```cpp

yaw = -90.0f;

```

Ви, можливо, вже це задаєтеся питанням: як ми встановлюємо і змінюємо ці значення рискання та крена?
```

Введення з миші

Значення рискання та крена отримуються від руху миші (або контролера/джойстика), де горизонтальний рух миші впливає на рискання, а вертикальний рух миші впливає на крен. Ідея полягає в тому, щоб зберігати позиції миші з минулого кадру і обчислювати, наскільки змінилися значення миші в поточному кадрі. Чим вище різниця у горизонтальному або вертикальному напрямі, тим більше ми оновлюємо значення крена чи рискання, і, отже, тим більше камера повинна рухатися.

Спочатку ми скажемо GLFW, що він повинен приховати курсор і захопити його. Захоплення курсора означає, що, якщо програма має фокус, курсор миші залишається в центрі вікна (крім випадків втрати фокуса або виходу з програми). Ми можемо це зробити одним простим викликом конфігурації:

```cpp
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```

Після цього виклику, куди б ми не рухали мишу, її не буде видно, і вона не повинна залишати вікно. Це ідеально підходить для системи камери у стилі FPS.

Щоб обчислити значення рискання та крена, нам потрібно сказати GLFW слухати події руху миші. Ми робимо це, створивши функцію зворотного виклику із наступним прототипом:

```cpp
void mouse_callback(GLFWwindow* window, double xpos, double ypos);
```

Ось xpos та ypos представляють поточні позиції миші. Як тільки ми зареєструємо функцію зворотного виклику за допомогою GLFW, кожного разу, коли миша рухається, викликається функція mouse_callback:

```cpp
glfwSetCursorPosCallback(window, mouse_callback);
```

При обробці введення миші для камери у стилі "літачої" є кілька кроків, які нам потрібно виконати, перш ніж ми зможемо повністю розрахувати вектор напрямку камери:

1. Розрахувати зміщення миші з минулого кадру.

2. Додати значення зміщення до значень рискання та крена камери.

3. Додати деякі обмеження для мінімальних/максимальних значень крена.

4. Розрахувати вектор напрямку.

Перший крок - розрахувати зміщення миші з останнього кадру. Спочатку нам потрібно зберегти останні позиції миші в програмі, які ми ініціалізуємо в центрі екрана (розмір екрану - 800 на 600) на початку:

```cpp
float lastX = 400, lastY = 300;
```

Потім у функції зворотного виклику миші розраховуємо зміщення між останнім та поточним кадром:

```cpp
float xoffset = xpos - lastX;
float yoffset = lastY - ypos; // зворотнє значення, оскільки координати у змінюються віднизу вгору
lastX = xpos;
lastY = ypos;
```

```
const float sensitivity = 0.1f;

xoffset *= sensitivity;

yoffset *= sensitivity;
```

Зверніть увагу, що ми множимо значення зміщення на чутливість. Якщо ми опустимо це множення, рух миші буде занадто сильним; експериментуйте зі значенням чутливості за своїм смаком.

Далі ми додаємо значення зміщення до глобально визначених значень рискання та крена:

```cpp
yaw   += xoffset;

pitch += yoffset;
```

На третьому кроці ми хочемо додати деякі обмеження камери, щоб користувачі не могли робити дивні рухи камерою (що також викликає обертання при LookAt, коли вектор напрямку паралельний напрямку вгору світу). Рискання повинно бути обмежене так, щоб користувачі не могли дивитися вище 89 градусів (при 90 градусах відбувається обертання LookAt), а також не нижче -89 градусів. Це

забезпечить можливість користувача подивитися на небо або вниз до своїх ніг, але не далі. Обмеження працюють так, що вони замінюють значення Ейлера його значенням обмеження, якщо воно порушує обмеження:

```cpp
if(pitch > 89.0f)
  pitch =  89.0f;
if(pitch < -89.0f)
  pitch = -89.0f;
```

Зверніть увагу, що ми не встановлюємо обмеження для значення рискання, оскільки ми не хочемо обмежувати користувача в горизонтальному обертанні. Однак це також легко додати обмеження для рискання, якщо ви так вирішите.

Четвертий і останній крок - розрахувати фактичний вектор напрямку, використовуючи формулу з попереднього розділу:

```cpp
glm::vec3 direction;
direction.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
```

direction.y = sin(glm::radians(pitch));

direction.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));

cameraFront = glm::normalize(direction);

```

Отриманий цим вектор напрямку вже містить всі обертання, розраховані від руху миші. Оскільки вектор cameraFront вже включений у функцію lookAt glm, ми готові до використання.

Якщо ви зараз запустите код, ви помітите, що камера робить великий раптовий стрибок, коли вікно вперше отримує фокус від курсора миші. Причиною цього раптового стрибка є те, що, як тільки курсор входить у вікно, функція зворотного виклику миші викликається з позначками xpos та ypos, рівними місцю, з якого миша увійшла на екран. Це часто є положенням, яке виявляється значно віддаленим від центру екрана, що призводить до великих зміщень і, отже, великого стрибка руху. Ми можемо обійти цю проблему, визначивши глобальну змінну типу bool для перевірки, чи це перше введення миші. Якщо це перше введення, ми оновлюємо початкові позначки миші новими значеннями xpos та ypos. Отримані потім рухи миші будуть використовувати координати місця, введеного новою позначкою миші, для розрахунку зміщень:

```cpp

if (firstMouse) // спочатку встановлено в true

{

    lastX = xpos;

```
    lastY = ypos;

    firstMouse = false;

}

```

Завершений код тепер виглядає так:

```cpp

void mouse_callback(GLFWwindow* window, double xpos, double ypos)

{

    if (firstMouse)

    {

        lastX = xpos;

        lastY = ypos;

        firstMouse = false;

    }



    float xoffset = xpos - lastX;

    float yoffset = lastY - ypos;
```

```
lastX = xpos;
lastY = ypos;

float sensitivity = 0.1f;
xoffset *= sensitivity;
yoffset *= sensitivity;

yaw   += xoffset;
pitch += yoffset;

if(pitch > 89.0f)
    pitch = 89.0f;
if(pitch < -89.0f)
    pitch = -89.0f;

glm::vec3 direction;
direction.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
direction.y = sin(glm::radians(pitch));
direction.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
```

    cameraFront = glm::normalize(direction);

}

```

Ось і все! Попробуйте тепер і ви побачите, що тепер ми можемо вільно переміщатися в нашій 3D-сцені!

## ZOOM

Як додатковий елемент до системи камери, ми також реалізуємо інтерфейс зумування. У попередньому розділі ми сказали, що поле зору або FOV в значній мірі визначає те, скільки ми можемо бачити сцени. Коли поле зору стає меншим, проектований простір сцени стає меншим. Цей менший простір проектується на той самий NDC (Normalized Device Coordinates), створюючи ілюзію збільшення. Для збільшення, ми будемо використовувати колесо прокрутки миші. Подібно до руху миші та введення з клавіатури, у нас є функція зворотного виклику для прокрутки миші:

```cpp
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)

{

    fov -= (float)yoffset;

    if (fov < 1.0f)

        fov = 1.0f;

    if (fov > 45.0f)

        fov = 45.0f;

}

```

Під час прокручування значення `yoffset` вказує нам, на скільки ми прокрутили вертикально. Коли викликається функція зворотного виклику `scroll_callback`, ми змінюємо вміст глобально визначеної змінної `fov`. Оскільки 45.0 - це значення за замовчуванням для `fov`, ми хочемо обмежити рівень зумування між 1.0 та 45.0.

Тепер нам потрібно завантажити матрицю перспективної проекції на GPU кожен кадр, але цього разу з `fov` як полем зору:

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

І, нарешті, не забудьте зареєструвати функцію зворотного виклику для прокрутки:

```cpp

```
glfwSetScrollCallback(window, scroll_callback);
```

І ось вам це. Ми реалізували просту систему камери, яка дозволяє вільно рухатися в 3D-середовищі.

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

## Клас камери

*У наступних розділах ми завжди будемо використовувати камеру, щоб легко оглядати сцени і бачити результати з різних кутів. Однак, оскільки код камери може займати значну кількість місця в кожному розділі, ми невеличко абстрагуємо його деталі та створюємо свій власний об'єкт камери, який виконає більшість роботи за нас, додавши деякі цікаві фішки. На відміну від розділу про шейдер, ми не будемо докладно вас проводити крізь створення класу камери, але надамо вам (повністю закоментований) вихідний код, якщо ви захочете розібратися в його внутрішніх роботах.*

*Подібно до об'єкта шейдера, ми визначаємо клас камери виключно в одному заголовковому файлі. Ви можете знайти клас камери тут; ви повинні мати можливість зрозуміти код після цього розділу. Рекомендується принаймні один раз перевірити клас як приклад того, як ви можете створити власну систему камери.*

*Систему камери, яку ми представили, є камера, яка підходить для більшості завдань і працює добре з ейлеровими кутами, але будьте обережні, коли створюєте різні системи камер, такі як камера від першої особи (FPS) чи камера для симуляції польоту. Кожна система камери має свої хитрощі та особливості, тому будьте впевнені, що читаєте про них. Наприклад, ця камера не дозволяє значенням крену більшими або рівними 90 градусам, а статичний вектор вгору (0,1,0) не працює, коли ми враховуємо значення кута крену.*

*Оновлену версію вихідного коду із новим об'єктом камери можна знайти тут.*

**Вправи**

1. *Спробуйте перетворити клас камери так, щоб він став справжньою камерою від першої особи (FPS), де ви не можете літати; ви можете лише дивитися навколо, залишаючись на площині xz: [рішення](посилання_на_рішення).*

2. *Спробуйте створити свою власну функцію LookAt, де ви вручну створите видову матрицю, як обговорювалося на початку цього розділу. Замініть функцію LookAt від glm своїм власним виконанням і переконайтеся, що вона все ще працює так само: [рішення](посилання_на_рішення).*