Лекція 4

Лабораторна робота 4

Текстури

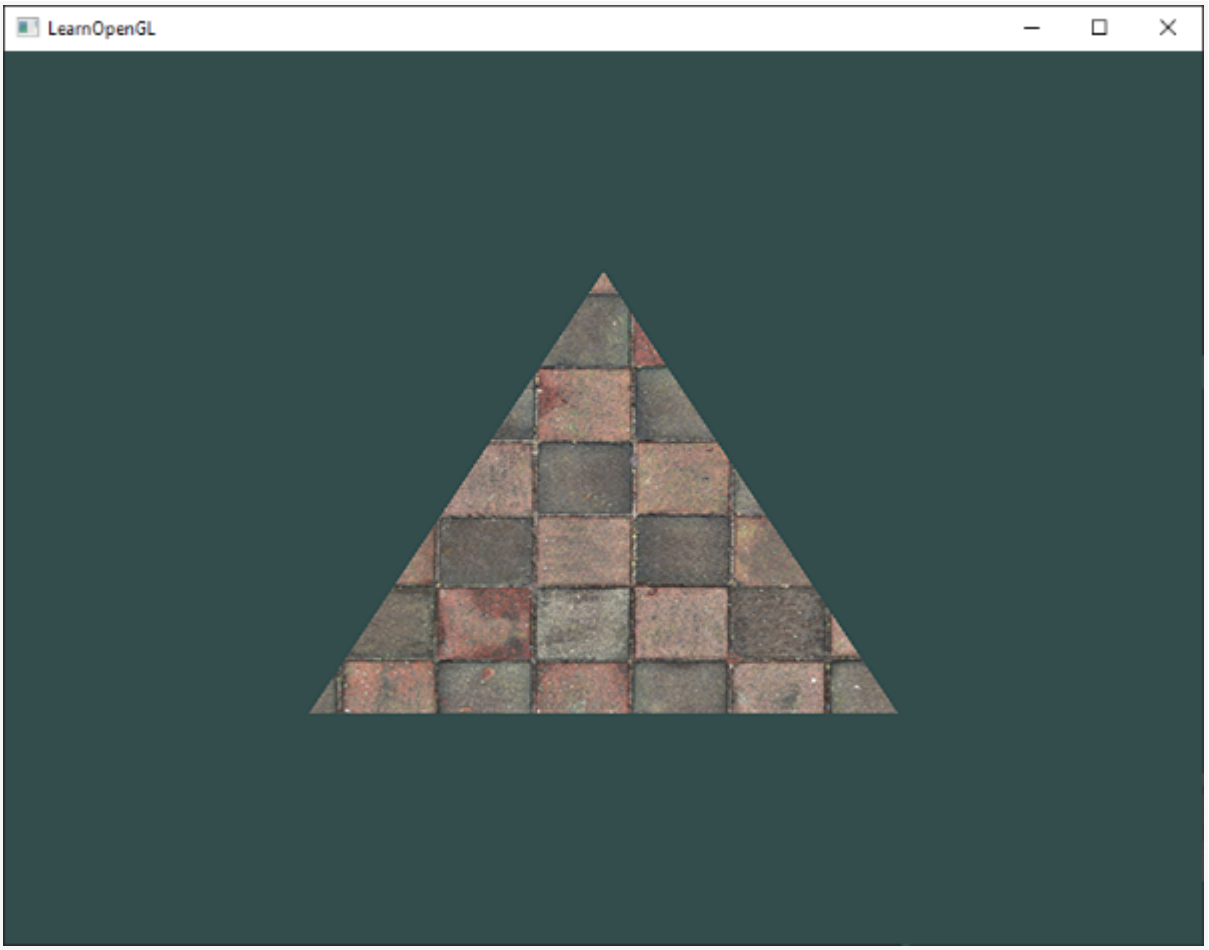
Ми дізналися, що щоб додати більше деталей нашим об’єктам, ми можемо використовувати кольори для кожної вершини, щоб створити цікаві зображення. Однак, щоб отримати неабияку реалістичність, нам потрібно мати багато вершин, аби ми могли вказати багато кольорів. Це вимагає значної кількості додаткових витрат, оскільки кожна модель потребує набагато більше вершин і для кожної вершини також колірний атрибут.

Художники та програмісти зазвичай віддають перевагу використанню текстур.

Текстура — це двовимірне зображення (існують навіть одновимірні та тривимірні текстури), яке використовується для додавання деталей до об’єкта; Подумайте про текстуру як про аркуш паперу з гарним зображенням цегли (наприклад), акуратно складений поверх вашого 3D-будиночка, щоб виглядало, ніби ваш будинок має кам’яний зовнішній вигляд. Оскільки ми можемо вставити багато деталей в одне зображення, ми можемо створювати ілюзію надзвичайно детального об’єкта без необхідності вказувати додаткові вершини.

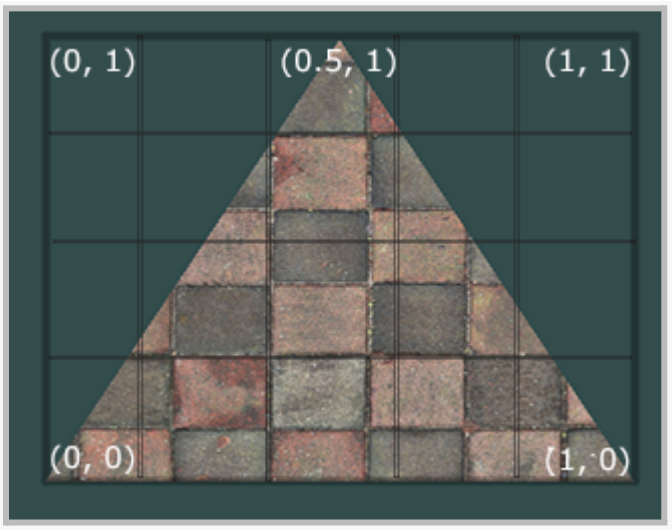
> Окрім зображень, текстури також можна використовувати для зберігання великої колекції довільних даних для надсилання до шейдерів, але ми залишимо це для іншої теми.

Нижче ви побачите зображення текстури цегляна стіна відображається на трикутнику з попереднього розділу.



Щоб відобразити текстуру на трикутнику, нам потрібно повідомити кожній вершині трикутника, якій частині текстури вона відповідає. Таким чином, кожна вершина повинна мати координати текстури пов'язані з ними, які визначають, з якої частини зображення текстури взяти вибірку. Потім інтерполяція фрагментів зробить все інше для інших фрагментів.

Координати текстури в діапазоні від `0` до`1` (пам'ятайте, що ми використовуємо двовимірні зображення текстур). Викликається отримання кольору текстури за допомогою координат текстури. Координати текстури починаються з`(0,0)` для нижнього лівого кута зображення текстури`(1,1)` для верхнього правого кута зображення текстури. На наступному зображенні показано, як ми наносимо координати текстури на трикутник:



Для трикутника вказуємо 3 координатні точки текстури. Ми хочемо, щоб нижня ліва сторона трикутника відповідала нижній лівій стороні текстури, тому ми використовуємо `(0,0)` координата текстури для нижньої лівої вершини трикутника. Те саме стосується нижнього правого боку з а `(1,0)` координата текстури. Верхня частина трикутника повинна відповідати верхньому центру зображення текстури, тому ми беремо `(0,5,1,0)` як його координата текстури. Нам потрібно лише передати 3 координати текстури у вершинний шейдер, який потім передає їх у фрагментний шейдер, який акуратно інтерполює всі координати текстури для кожного фрагмента.

Тоді отримані координати текстури виглядатимуть так:

```
float texCoords[] = {
 0.0f, 0.0f, // lower-left corner
1.0f, 0.0f, // lower-right corner
0.5f, 1.0f // top-center corner
};
```

Вибірка текстури має вільну інтерпретацію та може бути виконана різними способами. Таким чином, наша робота — розповісти OpenGL, як це має бути .

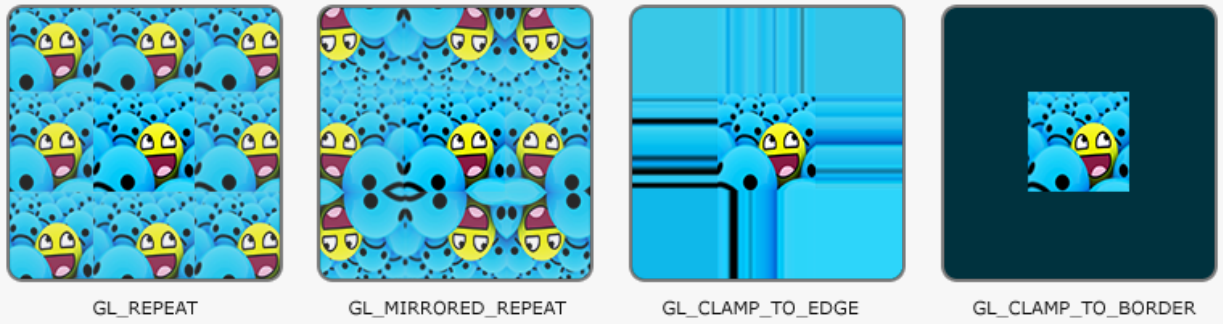
Обгортка текстури

Координати текстури зазвичай коливаються від `(0,0)` до `(1,1)` але що станеться, якщо ми вкажемо координати за межами цього діапазону? Поведінка OpenGL за замовчуванням полягає в повторенні зображень текстури (ми фактично ігноруємо

цілу частину координати текстури з плаваючою комою), але OpenGL пропонує більше варіантів:

- `GL_REPEAT`: типова поведінка для текстур. Повторює зображення текстури.
- `GL_MIRRORED_REPEAT`: Такий же, як `GL_REPEAT` але відображає зображення з кожним повторенням.
- `GL_CLAMP_TO_EDGE`: фіксує координати між 0 і 1. Результатом є те, що вищі координати стають затиснутими до краю, що призводить до розтягнутого малюнка до краю.
- `GL_CLAMP_TO_BORDER`: Координати за межами діапазону тепер отримують визначений користувачем колір межі.

Кожен із параметрів має різний візуальний вихід при використанні координат текстури поза діапазоном за замовчуванням. Давайте подивимося, як вони виглядають на прикладі зображення текстури (оригінальне зображення Хольгера Резенде):



Кожен із зазначених вище параметрів можна встановити для кожної координатної осі

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);
```

Перший аргумент визначає тип текстури; ми працюємо з 2D-текстурами, тому цільовою текстурою є `GL_TEXTURE_2D`. Другий аргумент вимагає від нас вказати, який параметр ми хочемо встановити та для якої осі текстури; ми хочемо налаштувати його для обох `вісей`. Останній аргумент вимагає від нас увімкнути потрібний режим обгортання текстурою, і в цьому випадку OpenGL встановить опцію обгортання текстури для поточної активної текстури за допомогою `GL_MIRRORED_REPEAT`.

Якщо ми вибираємо опцію GL_CLAMP_TO_BORDER, ми також повинні вказати колір межі. Це робиться за допомогою еквівалента fv функції glTexParameter з опцією GL_TEXTURE_BORDER_COLOR, в яку ми передаємо масив float з значенням кольору межі:

```
float borderColor[] = { 1.0f, 1.0f, 0.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

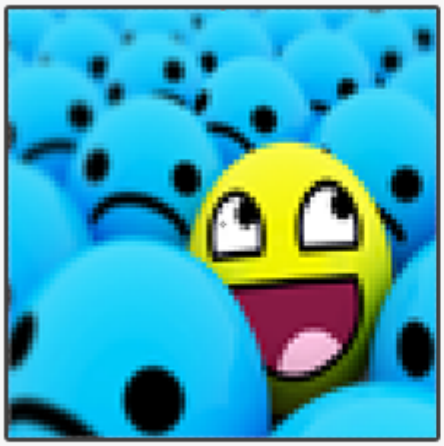
Фільтрація текстури

Текстурні координати не залежать від роздільної здатності, а можуть бути будь-яким значенням з плаваючою комою, тому OpenGL повинен визначити, до якого текстурного пікселя (також відомого як текстурний піксель або тексель) відобразити текстурну координату. Це стає особливо важливим, якщо у вас є дуже великий об'єкт та текстура низької роздільної здатності. Ви, ймовірно, вже здогадалися, що в OpenGL є також опції для цьої фільтрації текстур. Існують різні доступні опції, але наразі ми розглянемо найважливіші опції: GL_NEAREST та GL_LINEAR.

GL_NEAREST (також відомий як фільтрація за найближчим сусідом або точкова фільтрація) - це метод фільтрації текстур, заданий за замовчуванням в OpenGL. Коли встановлено GL_NEAREST, OpenGL вибирає тексель, до центру якого текстурна координата найближча. Нижче ви можете побачити 4 пікселя, де крестик позначає точну текстурну координату. Лівий верхній тексель має свій центр, найближчий до текстурної координати, і тому обирається в якості вибраного кольору для вибірки:

GL_LINEAR (також відомий як (бі)лінійна фільтрація) використовує інтерпольоване значення з сусідніх текселів текстури, наближаючи колір між текселями. Чим менше відстань від текстурної координати до центра текселя, тим більше внесок внесе колір цього текселя у вибране значення кольору. Нижче ми можемо побачити, що повертається змішаний колір сусідніх пікселів:

Але які візуальні ефекти такого методу фільтрації текстури? Подивімося, як працюють ці методи при використанні текстури з низькою роздільною здатністю на великому об'єкті (текстура тому масштабується вгору, і окремі тексели стають помітними):



GL_NEAREST

GL_LINEAR

GL_NEAREST призводить до блокових малюнків, де ми чітко бачимо пікселі, що утворюють текстуру, тоді як GL_LINEAR створює більш плавний малюнок, де окремі пікселі менше видно. GL_LINEAR створює більш реалістичний результат, але деякі розробники віддають перевагу більш "8-бітовому" вигляду і, як результат, вибирають опцію GL_NEAREST.

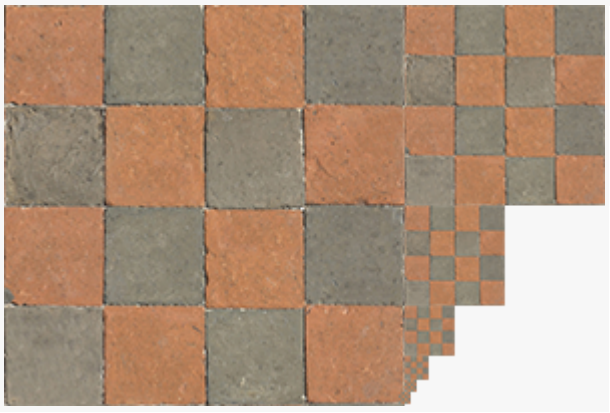
Фільтрацію текстури можна налаштовувати для операцій масштабування (якщо збільшується або зменшується розмір текстури), тому ви, наприклад, можете використовувати фільтрацію за найближчим сусідом, коли текстури зменшуються, і лінійну фільтрацію для збільшених текстур. Тому вам потрібно вказати метод фільтрації для обох опцій за допомогою `glTexParameteri*`. Код повинен схоже виглядати на налаштування методу обгортання:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Mipmaps(Міпмапи)

Уявімо, що у нас є велика кімната з тисячами об'єктів, кожен з приєднаною текстурою. Є об'єкти, які дуже далеко від глядача, але мають прикріплену текстуру високої роздільної здатності так само, як і об'єкти близько до глядача. Оскільки об'єкти знаходяться далеко і, ймовірно, вони створюють лише кілька фрагментів, OpenGL має труднощі у виборі правильного кольору для свого фрагмента з текстури високої роздільної здатності, оскільки йому потрібно вибрати кольорове значення текстури для фрагмента, який охоплює велику частину текстури. Це призводить до видимих артефактів на малих об'єктах, не кажучи вже про втрату пропускної здатності пам'яті при використанні текстур високої роздільної здатності на малих об'єктах.

Для вирішення цієї проблеми OpenGL використовує концепцію, що називається міпмапами, яка по суті представляє собою колекцію текстурних зображень, де кожне наступне зображення у два рази менше, ніж попереднє. Ідея міпмапів має бути простою: після певного порогу від глядача OpenGL буде використовувати іншу міпмап-текстуру, яка найкраще підходить для відстані до об'єкта. Оскільки об'єкт знаходиться далеко, менша роздільна здатність не буде помітною для користувача. OpenGL може зразувати правильні текселі, і при цьому залучено менше кеш-пам'яті під час вибору частини міпмапів. Давайте поглянемо на те, як виглядає текстура з міпмапами:



Створення колекції текстур з міпмапами для кожного текстурного зображення вручну є рутинною роботою, але на щастя, OpenGL може виконати всю роботу за нас одним викликом `glGenerateMipmap` після створення текстури.

Під час перемикання між рівнями міпмапів під час рендерингу OpenGL може показати артефакти, такі як гострі краї поміж двома шарами міпмапів. Так само, як звичайна фільтрація текстури, можливо також встановити фільтрацію між рівнями міпмапів за допомогою методів NEAREST та LINEAR для перемикання між рівнями міпмапів. Щоб вказати метод фільтрації між рівнями міпмапів, ми можемо замінити оригінальні методи фільтрації одним із чотирьох наступних варіантів:

- GL_NEAREST_MIPMAP_NEAREST: бере найближчий рівень міпмапу, який відповідає розміру пікселя і використовує інтерполяцію за методом найближчого сусіда для вибірки текстури.
- GL_LINEAR_MIPMAP_NEAREST: бере найближчий рівень міпмапу та вибирає цей рівень, використовуючи лінійну інтерполяцію.
- GL_NEAREST_MIPMAP_LINEAR: лінійно інтерполюється між двома міпмапами, які найбільше відповідають розміру пікселя, та вибирається інтерпольований рівень за методом найближчого сусіда.
- GL_LINEAR_MIPMAP_LINEAR: лінійно інтерполюється між двома найближчими міпмапами та вибирається інтерпольований рівень за методом лінійної інтерполяції.

Так само, як з фільтрацією текстури, ми можемо встановити метод фільтрації на один з чотирьох вищезазначених методів за допомогою `glTexParameteri`:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Розповсюдженою помилкою є встановлення одного з параметрів фільтрації міпмапів як фільтрації масштабування. Це не має жодного впливу, оскільки міпмапи в основному використовуються для зменшення розмірів текстур: масштабування текстур не використовує міпмапи, і встановлення параметра фільтрації міпмапів для масштабування генерує помилку OpenGL GL_INVALID_ENUM.

Завантаження та створення текстур

Завантаження та створення текстур Перша річ, яку нам потрібно зробити, щоб дійсно використовувати текстури, це завантажити їх у нашу програму. Зображення текстур можуть бути збережені в десятках форматах файлів, кожен зі своєю структурою та порядком даних, тому як ми можемо отримати ці зображення у нашій програмі? Одним рішенням було б вибрати формат файлу, який ми хочемо використовувати, наприклад .PNG, і написати власний завантажувач зображень для конвертації формату зображення в великий масив байтів. Навіть якщо написати власний завантажувач зображень не так важко, це все ще незручно, і якщо ви хочете підтримувати більше форматів файлів, вам доведеться писати завантажувач зображень для кожного формату, який ви хочете підтримувати.

Ще одним рішенням, і, можливо, хорошим, є використання бібліотеки для завантаження зображень, яка підтримує кілька популярних форматів та виконує всю важку роботу за нас. Бібліотека, така як stb_image.h.

stb_image.h stb_image.h - це дуже популярна однорядкова бібліотека для завантаження зображень від Шона Барретта, яка може завантажувати більшість популярних форматів файлів і легко інтегрується у ваші проекти. stb_image.h можна завантажити звідси. Просто завантажте однорядковий файл шапки, додайте його до свого проекту як stb_image.h і створіть додатковий файл C++ з наступним кодом:

stb_image.h

`stb_image.h` є дуже популярною бібліотекою для завантаження зображень із одним заголовкомШон Барретт який здатний завантажувати більшість популярних форматів файлів і його легко інтегрувати у ваш проект(и).`stb_image.h` можна завантажити зтут. Просто завантажте єдиний файл заголовка та додайте його до свого проекту як`stb_image.h`і створіть додатковий файл C++ із таким кодом:

```
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
```

визначивши STB_IMAGE_IMPLEMENTATION, препроцесор модифікує файл заголовка так, що він містить лише відповідний вихідний код визначення, фактично перетворюючи файл заголовка у .cpp файл, і це все. Просто включіть stb_image.h десь у своїй програмі і скомпілюйте її.

Для наступних розділів про текстури ми будемо використовувати зображення дерев'яного контейнера. Щоб завантажити зображення за допомогою stb_image.h, ми використовуємо його функцію stbi_load:

```
int width, height, nrChannels; unsigned char *data =
stbi_load("container.jpg", &width, &height, &nrChannels, 0);
```

Функція спершу приймає розташування файлу зображення в якості вхідних даних. Потім вона очікує, що ви надасте три цілі числа в якості її другого, третього та четвертого аргументів, які stb_image.h заповнить шириною, висотою та кількістю каналів кольору результатуючого зображення. Нам потрібні ширина і висота зображення для створення текстур пізніше.

Генерація текстури

Як і будь-які попередні об'єкти в OpenGL, текстури ідентифікуються з використанням ідентифікатора; давайте створимо один:

```
unsigned int texture;
glGenTextures(1, &texture);
```

Функція glGenTextures спершу приймає вхідним параметром кількість текстур, які ми хочемо згенерувати, і зберігає їх в масиві unsigned int, переданому як її другий аргумент (у нашому випадку, це просто одне беззнакове ціле число). Так само, як і з іншими об'єктами, нам потрібно зв'язати текстуру, щоб будь-які подальші команди текстури налаштовували поточну зв'язану текстуру:

```
glBindTexture(GL_TEXTURE_2D, texture);
```

Тепер, коли текстура зв'язана, ми можемо почати генерувати текстуру, використовуючи раніше завантажені дані зображення. Текстури генеруються за допомогою функції `glTexImage2D`:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
glGenerateMipmap(GL_TEXTURE_2D);
```

Ця функція має багато параметрів, тому розглянемо їх по кроках:

> Перший аргумент вказує ціль текстури; встановлення його на GL_TEXTURE_2D означає, що ця операція буде генерувати текстуру на поточно зв'язаному об'єкті текстури з такою ж цілью (тобто будь-які текскури, зв'язані з цілями GL_TEXTURE_1D або GL_TEXTURE_3D, не будуть змінюватися).
> Другий аргумент вказує рівень міпмапа, для якого ми хочемо створити текстуру, якщо ви хочете встановити кожний рівень міпмапа вручну, але ми залишимо його на базовому рівні, який дорівнює 0.
> Третій аргумент повідомляє OpenGL, в якому форматі ми хочемо зберегти текстуру. Наше зображення має лише значення RGB, тому ми також збережемо текстуру зі значеннями RGB.
> Четвертий та п'ятий аргументи встановлюють ширину та висоту отриманої текстури. Ми зберегли їх раніше під час завантаження зображення, тому ми використаємо відповідні змінні.
> Наступний аргумент повинен завжди дорівнювати 0 (деякі застарілі речі).
> Сьомий та восьмий аргументи визначають формат і тип даних джерела зображення. Ми завантажили зображення зі значеннями RGB та зберегли їх як символи (байти), тому ми передамо відповідні значення.
> Останнім аргументом є фактичні дані зображення. Після виклику glTexImage2D, на поточно зв'язаному об'єкті текстури тепер зображення текстури. Однак наразі завантажений тільки базовий рівень зображення текстури, і якщо ми хочемо використовувати міпмапи, нам потрібно вручну вказати всі різні зображення (постійно збільшуючи другий аргумент) або можемо викликати glGenerateMipmap після генерації текстури. Це автоматично генерує всі необхідні міпмапи для поточно зв'язаної текстури.

Після завершення генерації текстури та відповідних міпмапів доброю практикою є звільнити пам'ять зображення.

```
stbi_image_free(data);
```

Таким чином, весь процес створення текстури виглядає приблизно так:

```
unsigned int texture;
 glGenTextures(1, &texture);
 glBindTexture(GL_TEXTURE_2D, texture);
 // set the texture wrapping/filtering options (on the currently bound
texture object)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
 // load and generate the texture
int width, height, nrChannels;
 unsigned char *data = stbi_load("container.jpg", &width, &height,
&nrChannels, 0);
 if (data) {
      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
GL_RGB,GL_UNSIGNED_BYTE, data);
       glGenerateMipmap(GL_TEXTURE_2D);
 }
 else {
 std::cout << "Failed to load texture" << std::endl;
 }
 stbi_image_free(data);
```

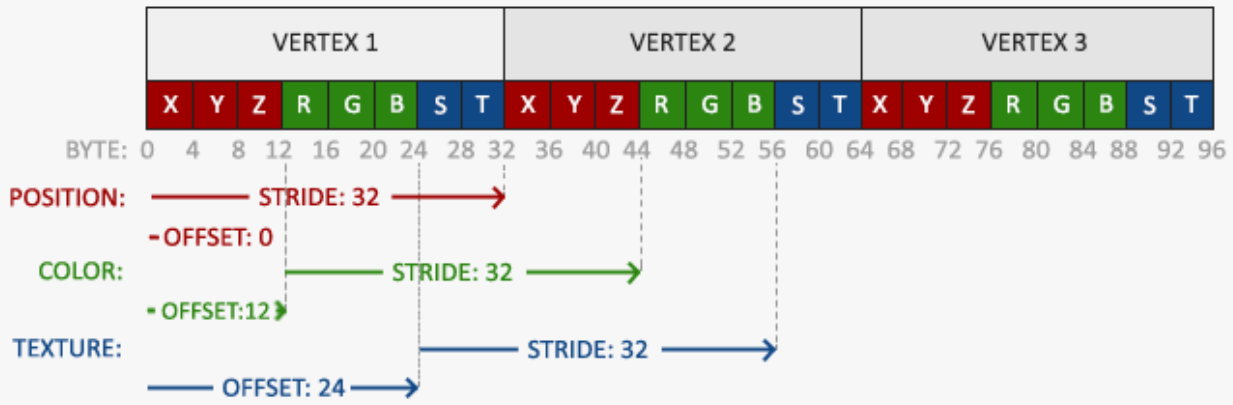
Нанесення текстур

Для наступних розділів ми будемо використовувати прямокутну форму, намальовану за допомогою glDrawElements з заключної частини розділу "Hello Triangle". Нам потрібно повідомити OpenGL, як відбувається взяття зразка текстури, тому нам потрібно оновити дані вершин текстури координатами:

```
float vertices[] = {
```

```
 // positions // colors // texture coords
0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, // top right
0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // bottom right
-0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, // bottom left
-0.5f, 0.5f, 0.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f // top left };
```

Оскільки ми додали додатковий атрибут вершини, нам знову потрібно повідомити OpenGL про новий формат вершини:




```
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float),
(void*)(6 * sizeof(float)));
glEnableVertexAttribArray(2);
```

Зверніть увагу, що нам також потрібно відкоригувати параметр кроку (stride) для попередніх двох атрибутів вершин на 8 * sizeof(float).

Далі нам потрібно змінити вершинний шейдер, щоб він приймав координати текстури як атрибут вершин, а потім передавав ці координати в фрагментний шейдер:

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aTexCoord;
out vec3 ourColor;
```

```
out vec2 TexCoord;
void main() {
	gl_Position = vec4(aPos, 1.0);
	ourColor = aColor;
	TexCoord = aTexCoord;
}
```

Фрагментний шейдер повинен приймати змінну виведення TexCoord як вхідну змінну.

Фрагментний шейдер також повинен мати доступ до об'єкта текстури, але як нам передати об'єкт текстури в фрагментний шейдер? У GLSL є вбудований тип даних для об'єктів текстури, який називається семплер (sampler) і має постфікс типу текстури, який нам потрібен, наприклад, sampler1D, sampler3D або у нашому випадку sampler2D. Потім ми можемо додати текстуру до фрагментного шейдера, просто оголосивши єдино семплер sampler2D, який ми пізніше призначимо нашій текстурі.

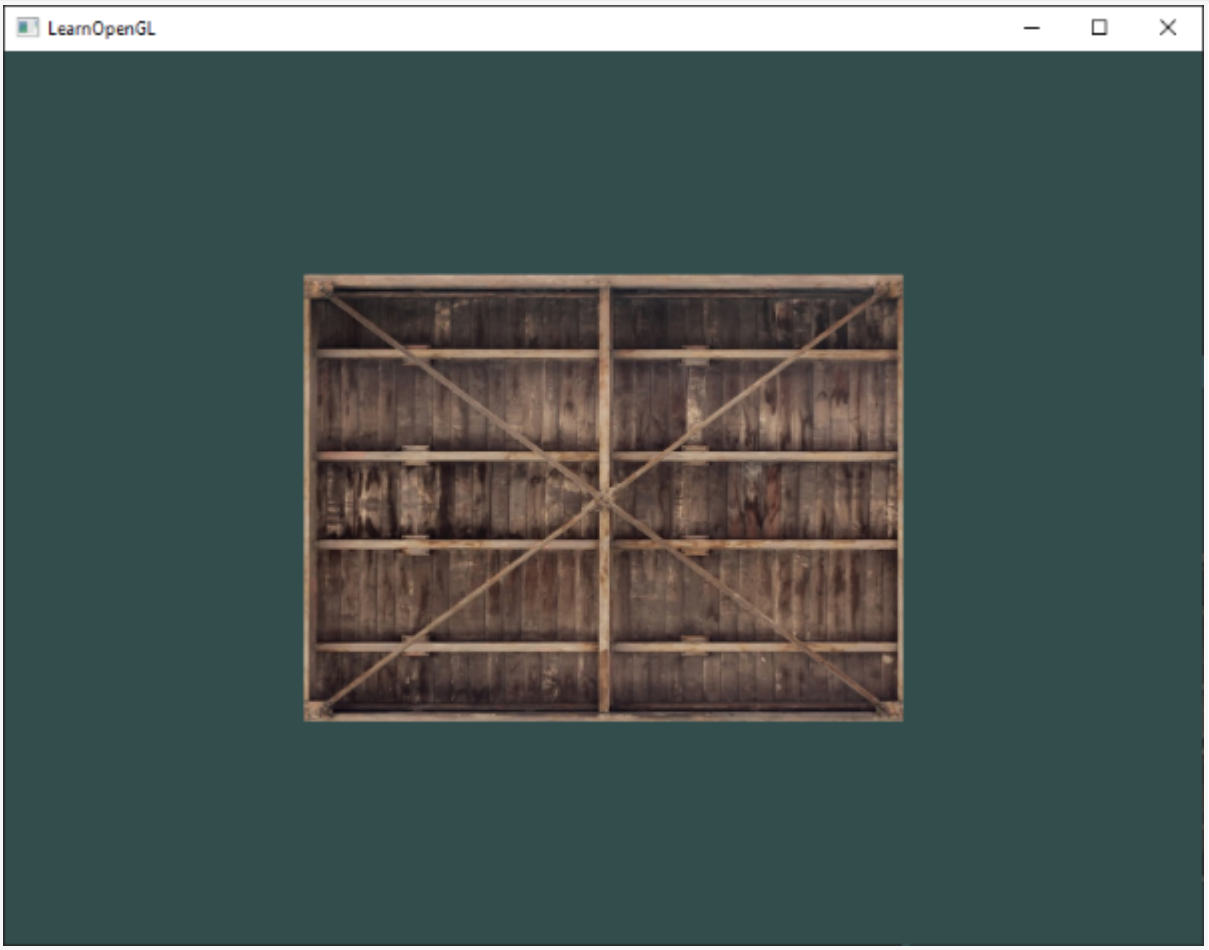
```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;
in vec2 TexCoord;
uniform sampler2D ourTexture;
void main() {
	FragColor = texture(ourTexture, TexCoord);
}
```

Для взяття зразка кольору текстури ми використовуємо вбудовану функцію texture мови GLSL, яка приймає як свій перший аргумент семплер текстури і як свій другий аргумент відповідні координати текстури. Функція texture зразкує відповідне значення кольору з використанням параметрів текстури, які ми встановили раніше. Вихід цього фрагментного шейдера - це (фільтрований) колір текстури в (інтерпольованих) координатах текстури.

Все, що залишилося зробити зараз, це зв'язати текстуру перед викликом glDrawElements, і вона автоматично призначить текстуру семплеру фрагментного шейдера:

```
glBindTexture(GL_TEXTURE_2D, texture);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Якщо ви все зробили правильно, ви повинні побачити таке зображення:



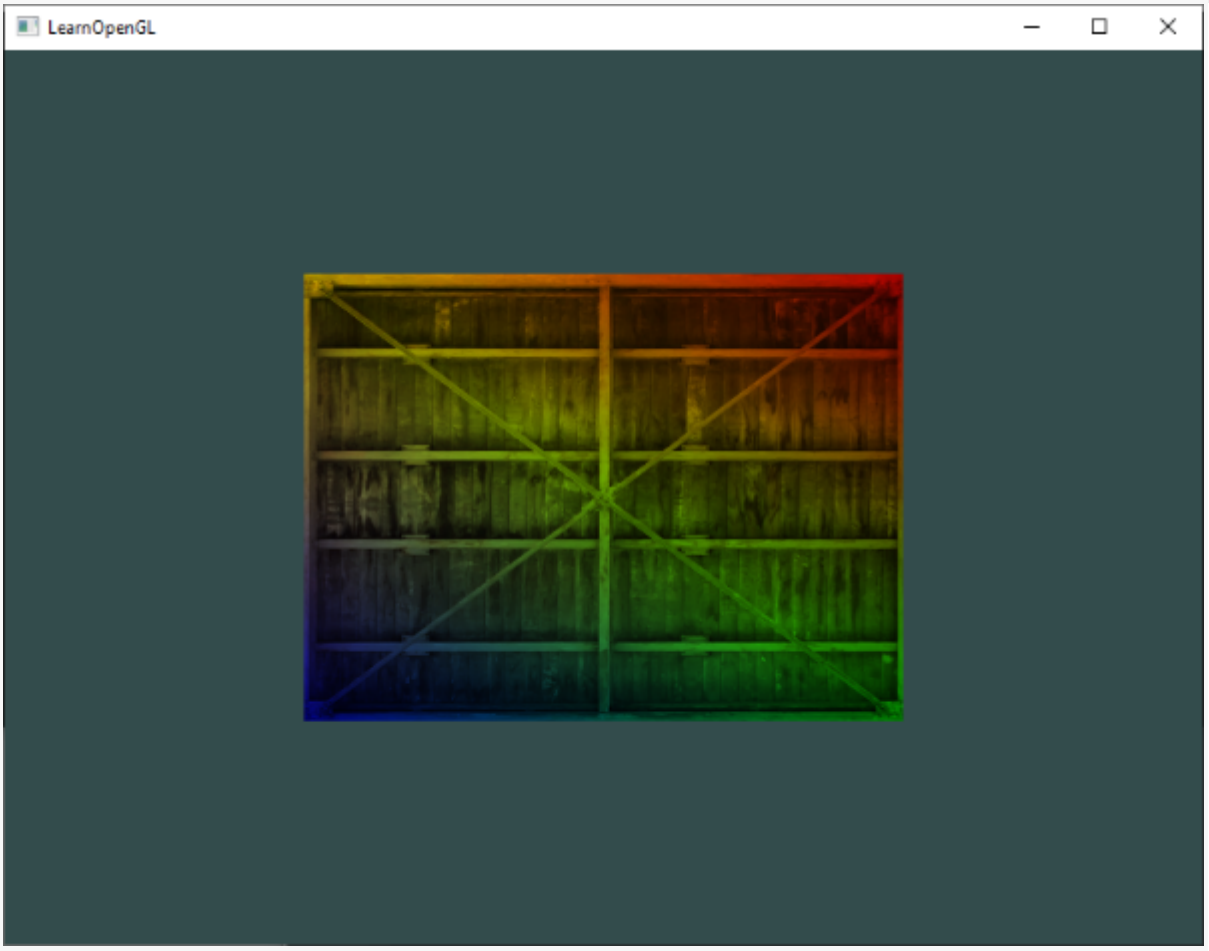
Якщо ваш прямокутник повністю білий або чорний, можливо, ви допустили помилку на шляху. Перевірте журнали шейдера та спробуйте порівняти ваш код із вихідним кодом програми.

Якщо ваш код текстури не працює або відображається як повністю чорний, продовжуйте читати і дійте до останнього прикладу, який повинен працювати. На деяких драйверах необхідно призначити текстурний блок для кожного семплеру, що є темою подальших розділів цієї глави.

Щоб додати трохи ефектності, ми також можемо змішувати отриманий колір текстури з кольорами вершин. Ми просто множимо отриманий колір текстури на колір вершини в фрагментному шейдері, щоб змішати обидва кольори:

```
FragColor = texture(ourTexture, TexCoord) * vec4(ourColor, 1.0);
```

У результаті має вийти суміш кольору вершини та кольору текстури:



Можна сказати, що наш контейнер любить дискотеку.

Текстурні одиниці

Ви, напевно, запитуєтеся, чому змінна sampler2D є uniform, якщо ми навіть не призначаємо їй значення за допомогою glUniform. Використовуючи glUniform1i, насправді ми можемо призначити значення розташування семплера текстури, щоб ми могли встановити декілька текстур одночасно в фрагментному шейдері. Це розташування текстури більш загально відомо як текстурний блок (texture unit). Текстурний блок за замовчуванням для текстури - 0, який є блоком текстури за замовчуванням, і тому нам не потрібно було призначати розташування в попередньому розділі; варто зазначити, що не всі графічні драйвери надають текстурний блок за замовчуванням, тому попередній розділ може не відобразитися для вас.

Головною метою текстурних блоків є дозвіл на використання більш як 1 текстури в наших шейдерах. Призначаючи текстурні блоки семплерам, ми можемо в'язати до декількох текстур одночасно, якщо спершу активуємо відповідний текстурний блок. Так само, як і glBindTexture, ми можемо активувати текстурні блоки за допомогою glActiveTexture, передаючи номер текстурного блоку, який ми хочемо використовувати:

```
glActiveTexture(GL_TEXTURE0); // activate the texture unit first before binding texture
glBindTexture(GL_TEXTURE_2D, texture);
```

Після активації текстурної одиниці і прив'язки текстури до цієї одиниці вам дійсно потрібно редагувати фрагментний шейдер, щоб додати інші семплери для інших текстур. У вашому фрагментному шейдері ви вже маєте семплер ourTexture, але для використання іншої текстури вам потрібно додати ще один семплер і прив'язати його до іншої текстурної одиниці.

Ваш фрагментний шейдер може виглядати, наприклад, так:

```
#version 330 core
out vec4 FragColor;

in vec3 ourColor;
in vec2 TexCoord;

uniform sampler2D ourTexture; // Перша текстура
uniform sampler2D anotherTexture; // Друга текстура

void main()
{
    // Семплуємо обидві текстури
    vec4 textureColor1 = texture(ourTexture, TexCoord);
    vec4 textureColor2 = texture(anotherTexture, TexCoord);

    // Змішуємо колір обох текстур, наприклад, використовуючи середнє значення
```

```
    vec4 finalColor = (textureColor1 + textureColor2) / 2.0;

    FragColor = finalColor;
}
```

Тепер ваш фрагментний шейдер використовує дві різні текстури (ourTexture і anotherTexture) і змішує їх кольори для кінцевого результату.

Не забудьте також встановити значення для anotherTexture за допомогою glUniform1i, щоб вказати, яку текстурну одиницю використовувати для другої текстури.

Завершене кольорове виведення тепер представляє собою комбінацію двох операцій вибору текстур. Вбудована у GLSL функція mix приймає два значення як вхідні дані і лінійно інтерполює їх на основі третього аргументу. Якщо третє значення - 0.0, вона повертає перше вхідне значення; якщо воно 1.0, воно повертає друге вхідне значення. Значення 0.2 поверне 80% першого вхідного кольору і 20% другого, що призведе до змішування обох текстур.

Зараз ми хочемо завантажити і створити іншу текстуру. Ви вже повинні бути знайомі з процесом. Переконайтеся, що створюєте ще один об'єкт текстури, завантажте зображення та створіть кінцеву текстуру, використовуючи `glTexImage2D`. Для другої текстури ми використовуватимемо зображення вашого обличчя, коли ви вивчаєте OpenGL.

```
unsigned char *data = stbi_load("awesomeface.png", &width, &height, &nrChannels, 0);
if (data) {
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);
 glGenerateMipmap(GL_TEXTURE_2D);
}
```

Зверніть увагу, що зараз ми завантажуємо .png-зображення, яке включає альфа-канал (прозорість). Це означає, що ми тепер повинні вказати, що дані зображення також містять альфа-канал, використовуючи GL_RGBA; інакше OpenGL неправильно інтерпретуватиме дані зображення.

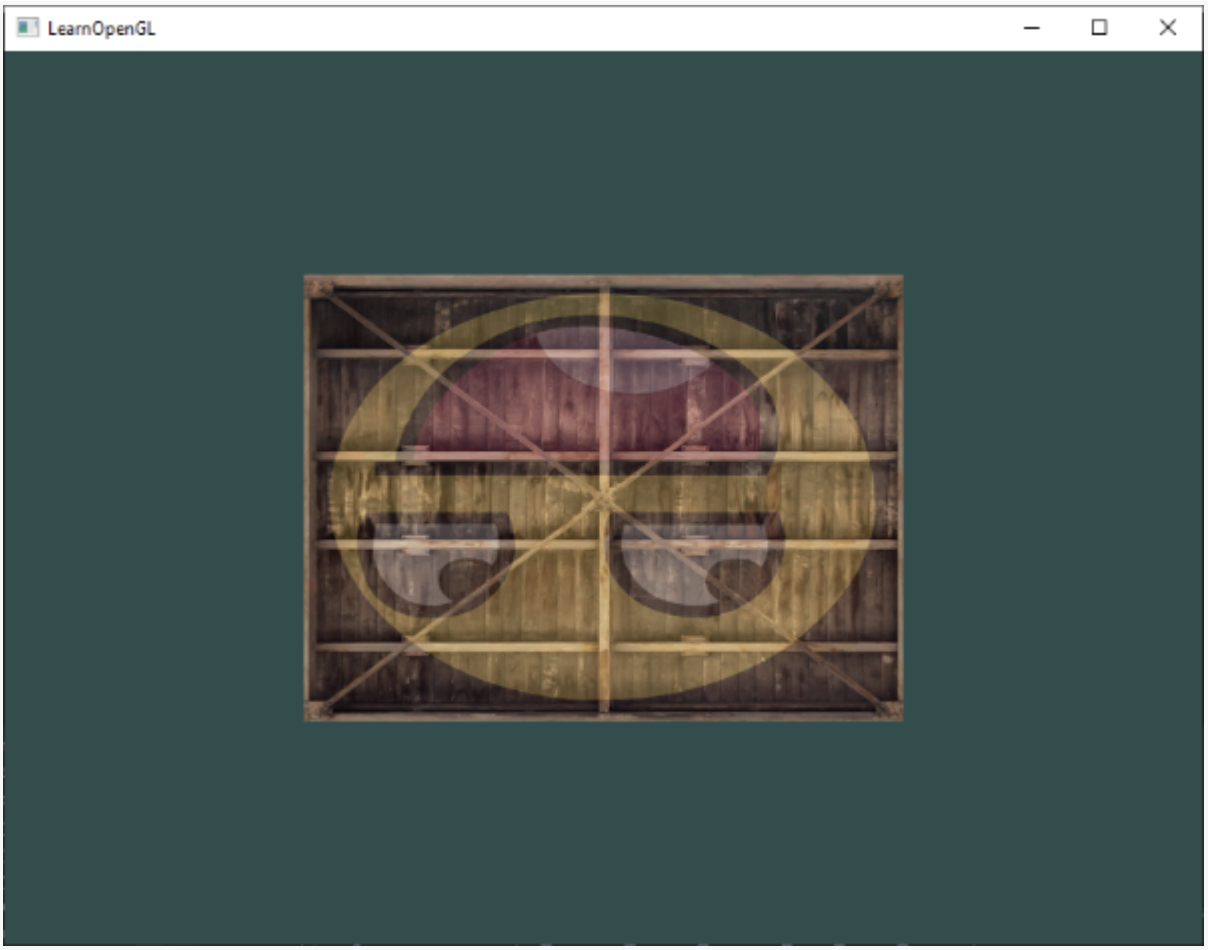
Для використання другої текстури (і першої текстури) нам потрібно слід трохи змінити процедуру відображення, прив'язуючи обидві текстури до відповідних текстурних одиниць:

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture1);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, texture2);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Ми також повинні повідомити OpenGL, до якої текстурної одиниці належить кожний семплер шейдера, встановлюючи кожний семплер за допомогою glUniform1i. Це потрібно встановити лише один раз, тому ми можемо зробити це перед входженням у цикл відображення:

```
ourShader.use();
 // don't forget to activate the shader before setting uniforms!
glUniform1i(glGetUniformLocation(ourShader.ID, "texture1"), 0);
 // set it manually ourShader.setInt("texture2", 1);
 // or with shader class
while(...)
 {
 [...]
}
```

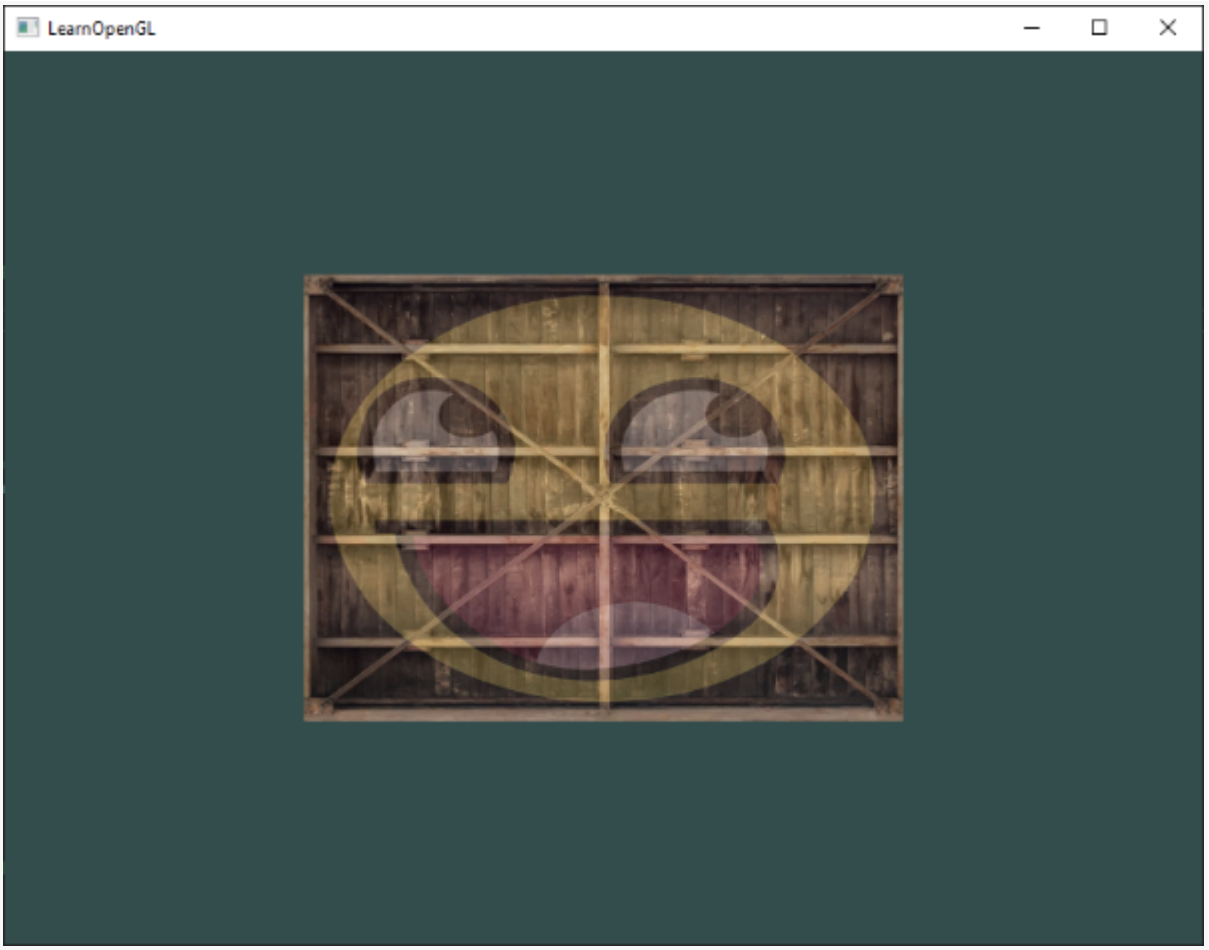
Встановивши семплери за допомогою glUniform1i, ми переконуємося, що кожен uniform семплер відповідає правильній текстурній одиниці. В результаті ви повинні отримати наступний результат:



Ви, напевно, помітили, що текстура перевернута догори дном! Це відбувається тому, що OpenGL очікує, що координата 0.0 на вісі y буде знаходитися на нижній стороні зображення, але зображення зазвичай мають 0.0 у верхній частині вісі y. На щастя, бібліотека stb_image.h може перевернути вісь y під час завантаження зображення, додавши такий рядок перед завантаженням будь-якого зображення:

```
stbi_set_flip_vertically_on_load(true);
```

Після того, як ви вказали stb_image.h перевертати вісь y під час завантаження зображень, ви повинні отримати наступний результат:



Якщо ви бачите один щасливий контейнер, ви зробили все правильно. Ви можете порівняти його з вихідним кодом .

Вправи

Щоб зручніше працювати з текстурами, радимо попрацювати над цими вправами, перш ніж продовжувати.

- Переконайтеся, що **лише** щасливе обличчя дивиться в іншому/зворотному напрямку, змінивши фрагментний шейдер: рішення .
- Експериментуйте з різними методами обтікання текстурою, вказуючи координати текстури в діапазоні `0.0f` до `2.0f` замість `0.0f` до `1.0f`. Подивіться, чи можна відобразити 4 смайлики на одному зображенні контейнера, затиснутому на його краю: рішення , результат . Подивіться, чи можете ви також поекспериментувати з іншими методами обгортання.
- Спробуйте відобразити лише центральні пікселі зображення текстури на прямокутнику таким чином, щоб окремі пікселі ставали видимими шляхом зміни координат текстури. Спробуйте встановити метод фільтрації текстури на `GL_NEAREST` , щоб чіткіше бачити пікселі: рішення .
- Використовуйте змінну-уніформу як третій параметр функції mix, щоб змінювати ступінь видимості двох текстур. Використовуйте клавіші зі стрілками вгору і вниз, щоб змінювати, наскільки видимий контейнер або смайлик:: рішення .