Лекція 8

Лабораторна робота 8

Освітлення

Кольори

Основне освітлення

Матеріали

Карти освітлення

Світлові ролики

Кілька вогнів

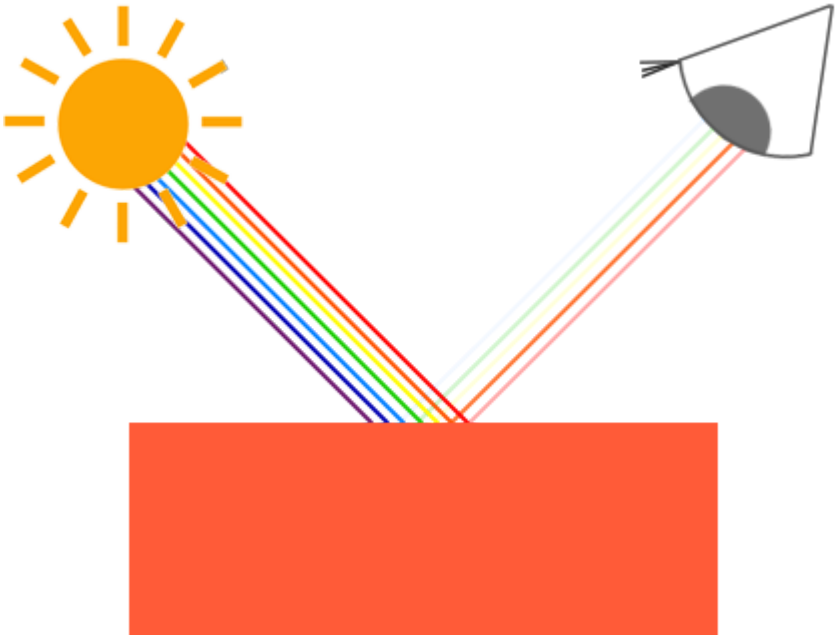
Кольори

У попередніх розділах ми коротко використовували та маніпулювали кольорами, але ніколи не визначали їх належним чином. Тут ми обговоримо, що таке кольори, і почнемо створювати сцену для майбутніх розділів освітлення.

У реальному світі кольори можуть мати будь-яке відоме значення кольору, причому кожен об'єкт має власний колір(и). У цифровому світі нам потрібно зіставити (нескінченну) реальні кольори з (обмеженими) цифровими значеннями, тому не всі реальні кольори можна представити цифровим способом. Кольори представлені цифровим способом за допомогою компонентів `red`, `green` та `blue`, які зазвичай скорочуються як RGB. Використовуючи різні комбінації лише цих 3 значень у діапазоні `[0,1]`, ми можемо представити майже будь-який колір. Наприклад, щоб отримати *кораловий* колір, ми визначаємо колірний вектор як:

```
glm::vec3 coral(1.0f, 0.5f, 0.31f);
```

Колір об’єкта, який ми бачимо в реальному житті, — це не колір, який він має насправді, а колірвідображенівід об'єкта. Кольори, які не поглинаються (відкидаються) об’єктом, є кольором, який ми сприймаємо. Як приклад, світло сонця сприймається як біле світло, яке є сукупністю багатьох різних кольорів (як ви можете бачити на зображенні). Якщо ми посвітимо цим білим світлом синю іграшку, вона поглине всі підкольори білого кольору, крім синього. Оскільки іграшка не поглинає синю частину, вона відбивається. Це відбите світло потрапляє в наше око, створюючи враження, що іграшка має синій колір. На наступному зображенні показано це для іграшки коралового кольору, де вона відображає кілька кольорів із різною інтенсивністю:



Ви бачите, що біле сонячне світло є сукупністю всіх видимих кольорів, і об’єкт поглинає велику частину цих кольорів. Він відображає лише ті кольори, які представляють колір об’єкта, а поєднання цих кольорів є тим, що ми сприймаємо (у цьому випадку кораловий колір).

Технічно це дещо складніше, але ми дійдемо до цього в розділах PBR.

Ці правила відображення кольору застосовуються безпосередньо в графіці. Коли ми визначаємо джерело світла в OpenGL, ми хочемо надати цьому джерелу світла колір. У попередньому параграфі у нас був білий колір, тому ми також надамо джерелу світла білий колір. Якщо ми потім помножимо колір джерела світла на значення кольору об'єкта, отриманий колір буде відбитим кольором об'єкта (і, таким чином, його кольором, який сприймається). Давайте знову розглянемо нашу іграшку (цього разу з кораловим значенням) і подивимось, як ми обчислимо її колір, який сприймається в графічному просторі. Ми отримуємо результуючий вектор кольору, виконуючи покомпонентне множення між векторами кольорів світла та об'єкта:

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

glm::vec3 toyColor(1.0f, 0.5f, 0.31f);

glm::vec3 result = lightColor * toyColor; // = (1.0f, 0.5f, 0.31f);
```

Ми бачимо, що колір іграшки *поглинає* велику частину білого світла, але відображає кілька червоних, зелених і синіх значень на основі його власне значення кольору. Це уявлення про те, як кольори працюватимуть у реальному житті. Таким чином, ми можемо визначити колір об'єкта як *кількість кожного компонента кольору, який він відбиває від джерела світла*. А що станеться, якщо ми використаємо зелене світло?

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

glm::vec3 toyColor(1.0f, 0.5f, 0.31f);
```

```
glm::vec3 result = lightColor * toyColor; // = (0.0f, 0.5f, 0.0f);
```

Як ми бачимо, іграшка не має червоного та синього світла для поглинання та/або відображення. Іграшка також поглинає половину світло-зеленого кольору, але також відбиває половину світло-зеленого кольору. Тоді колір іграшки, який ми сприймаємо, буде темно-зеленуватим. Ми бачимо, що якщо ми використовуємо зелене світло, лише компоненти зеленого кольору можуть бути відбиті і, таким чином, сприйняті; не сприймаються червоний і синій кольори. В результаті кораловий об'єкт раптово стає темно-зеленуватим. Давайте спробуємо ще один приклад із темним оливково-зеленим світлом:

```
glm::vec3 lightColor(0.33f, 0.42f, 0.18f);

glm::vec3 toyColor(1.0f, 0.5f, 0.31f);

glm::vec3 result = lightColor * toyColor; // = (0.33f, 0.21f, 0.06f);
```

Як бачите, ми можемо отримати цікаві кольори від об’єктів, використовуючи різні кольори світла. Творчо підійти до кольорів неважко.

Але досить про кольори, давайте почнемо будувати сцену, де ми можемо експериментувати.

Сцена освітлення

У наступних розділах ми створюватимемо цікаві візуальні ефекти, імітуючи освітлення реального світу, широко використовуючи кольори. Оскільки тепер ми будемо використовувати джерела світла, ми хочемо відобразити їх як

візуальні об’єкти на сцені та додати принаймні один об’єкт для імітації освітлення.

Перше, що нам потрібно, це об’єкт, на який можна пролити світло, і ми використаємо сумно відомий куб-контейнер із попередніх розділів. Нам також знадобиться світловий об’єкт, щоб показати, де знаходиться джерело світла в 3D-сцені. Для простоти ми також представимо джерело світла кубом (ми вже маємо дані вершини, правда?).

Отже, заповнення об’єкта буфера вершин, встановлення покажчиків атрибутів вершин і все це має бути для вас уже знайомим, тому ми не будемо проводити вас через ці кроки. Якщо ви все ще не знаєте, що з ними відбувається, я пропоную вам переглянути попередні розділи та, якщо можливо, виконати вправи, перш ніж продовження.

Отже, перше, що нам знадобиться, це вершинний шейдер, щоб намалювати контейнер. Позиції вершин контейнера залишаються незмінними (хоча цього разу нам не знадобляться координати текстури), тому в коді не повинно бути нічого нового. Ми будемо використовувати скорочену версію вершинного шейдера з останніх розділів:

```
#version 330 core

layout (location = 0) in vec3 aPos;


uniform mat4 model;

uniform mat4 view;

uniform mat4 projection;
```

```
void main()

{

    gl_Position = projection * view * model * vec4(aPos, 1.0);

}
```

Обов’язково оновіть дані вершин і покажчики атрибутів відповідно до нового шейдера вершин (якщо хочете, ви можете залишити дані текстури та покажчики атрибутів активними; ми просто зараз їх не використовуємо).

Оскільки ми також збираємося відобразити куб джерела світла, ми хочемо створити новий VAO спеціально для джерела світла. Ми могли б відобразити джерело світла за допомогою того самого VAO, а потім виконати кілька перетворень положення світла на матриці `model`, але в наступних розділах ми' Дані вершин і покажчики атрибутів об’єкта-контейнера змінюватимуться досить часто, і ми не хочемо, щоб ці зміни поширювалися на об’єкт джерела світла (нас дбають лише про положення вершин світлового куба), тому ми' #39;створю новий VAO:

```
unsigned int lightVAO;

glGenVertexArrays(1, &lightVAO);

glBindVertexArray(lightVAO);

// we only need to bind to the VBO, the container's VBO's data already contains the data.

glBindBuffer(GL_ARRAY_BUFFER, VBO);

// set the vertex attribute
```

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);

glEnableVertexAttribArray(0);
```

Код повинен бути відносно простим. Тепер, коли ми створили і контейнер, і куб джерела світла, залишилося визначити одну річ, а саме фрагментний шейдер як для контейнера, так і для джерела світла:

```
#version 330 core

out vec4 FragColor;


uniform vec3 objectColor;

uniform vec3 lightColor;


void main()

{

    FragColor = vec4(lightColor * objectColor, 1.0);

}
```

Фрагментний шейдер приймає як колір об’єкта, так і світлий колір від єдиної змінної. Тут ми множимо колір світла на колір (відбитого) об’єкта, як ми обговорювали на початку цієї глави. Знову ж таки, цей шейдер має бути

простим для розуміння. Давайте встановимо колір об'єкта на кораловий колір останньої секції з білим світлом:

```
// don't forget to use the corresponding shader program first (to set the uniform)
lightingShader.use();
lightingShader.setVec3("objectColor", 1.0f, 0.5f, 0.31f);
lightingShader.setVec3("lightColor",  1.0f, 1.0f, 1.0f);
```

Залишається зауважити, що коли ми почнемо оновлювати ці *шейдери освітлення* в наступних розділах, це також вплине на куб джерела світла, а це не що ми хочемо. Ми не хочемо, щоб колір об’єкта джерела світла вплинув на розрахунки освітлення, а радше тримаємо джерело світла ізольованим від решти. Ми хочемо, щоб джерело світла мало постійний яскравий колір, на який не впливають інші зміни кольору (це створює враження, ніби куб джерела світла справді є джерелом світла).

Щоб досягти цього, нам потрібно створити другий набір шейдерів, які ми будемо використовувати для малювання куба джерела світла, таким чином захищаючи від будь-яких змін шейдерів освітлення. Вершиний шейдер такий самий, як і вершинний шейдер освітлення, тому ви можете просто скопіювати вихідний код. Фрагментний шейдер куба джерела світла гарантує, що колір куба залишається яскравим, визначаючи постійний білий колір на лампі:

```
#version 330 core
out vec4 FragColor;
```

```
void main()

{

    FragColor = vec4(1.0); // set all 4 vector values to 1.0

}
```

Коли ми хочемо відобразити, ми хочемо відобразити об’єкт-контейнер (або, можливо, багато інших об’єктів) за допомогою шейдера освітлення, який ми щойно визначили, а коли ми хочемо намалювати джерело світла, ми використовуємо шейдери джерела світла. У розділах, присвячених освітленню, ми поступово оновлюватимемо шейдери освітлення, щоб поступово досягти більш реалістичних результатів.

Основне призначення куба джерела світла – показати, звідки виходить світло. Зазвичай ми визначаємо положення джерела світла десь у сцені, але це просто положення, яке не має візуального значення. Щоб показати, де насправді знаходиться джерело світла, ми візуалізуємо куб у тому самому місці джерела світла. Ми візуалізуємо цей куб за допомогою шейдера куба джерела світла, щоб переконатися, що куб завжди залишається білим, незалежно від умов освітлення сцени.

Отже, давайте оголосимо глобальну змінну vec3, яка представляє розташування джерела світла в координатах світового простору:

```
glm::vec3 lightPos(1.2f, 1.0f, 2.0f);
```

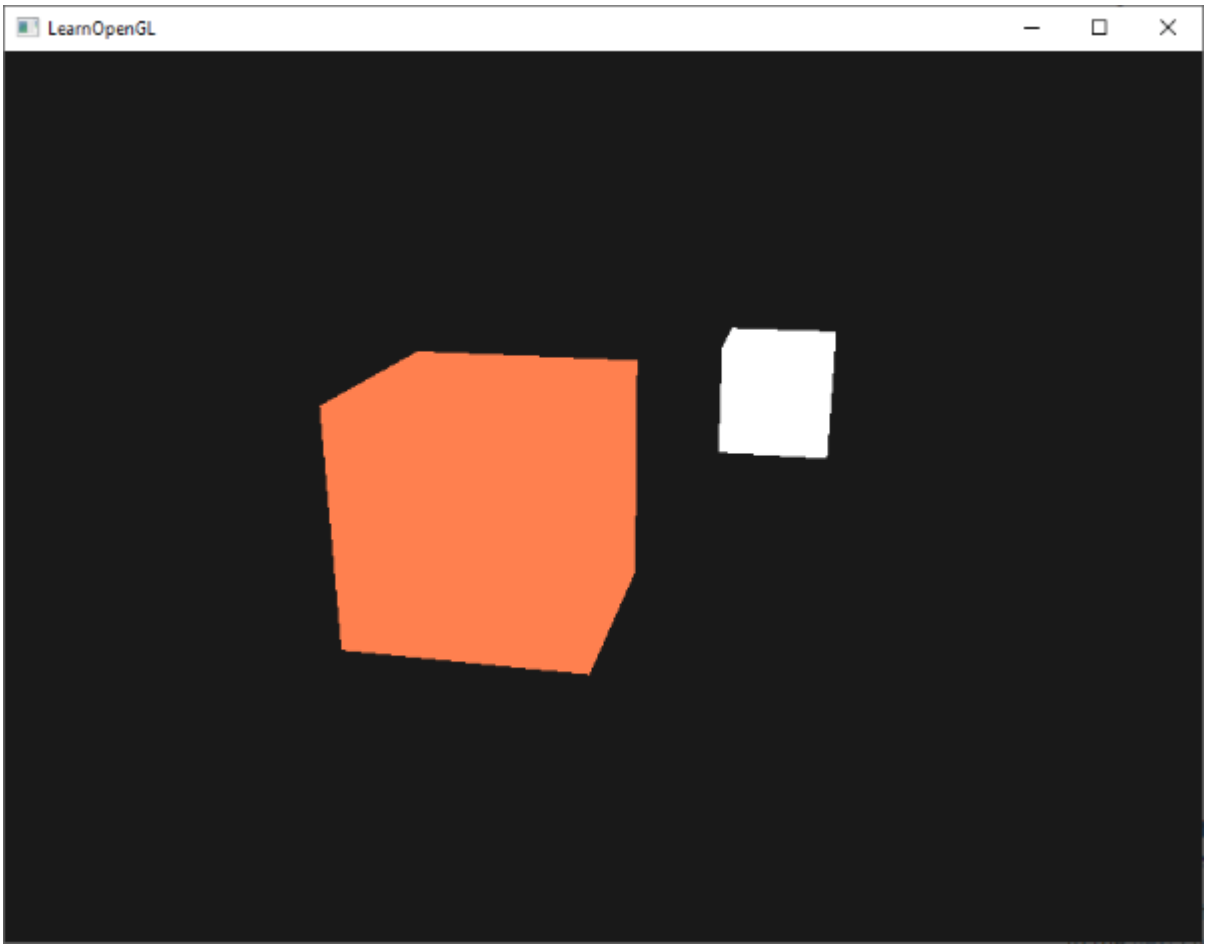
Потім ми переводимо куб джерела світла в положення джерела світла та зменшуємо його масштаб перед відтворенням:

```
model = glm::mat4(1.0f);
model = glm::translate(model, lightPos);
model = glm::scale(model, glm::vec3(0.2f));
```

Отриманий код візуалізації для куба джерела світла має виглядати приблизно так:

```
lightCubeShader.use();
// set the model, view and projection matrix uniforms
[...]
// draw the light cube object
glBindVertexArray(lightCubeVAO);
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Введення всіх фрагментів коду у відповідні місця призведе до чистої програми OpenGL, правильно налаштованої для експериментів з освітленням. Якщо все компілюється, це має виглядати так:



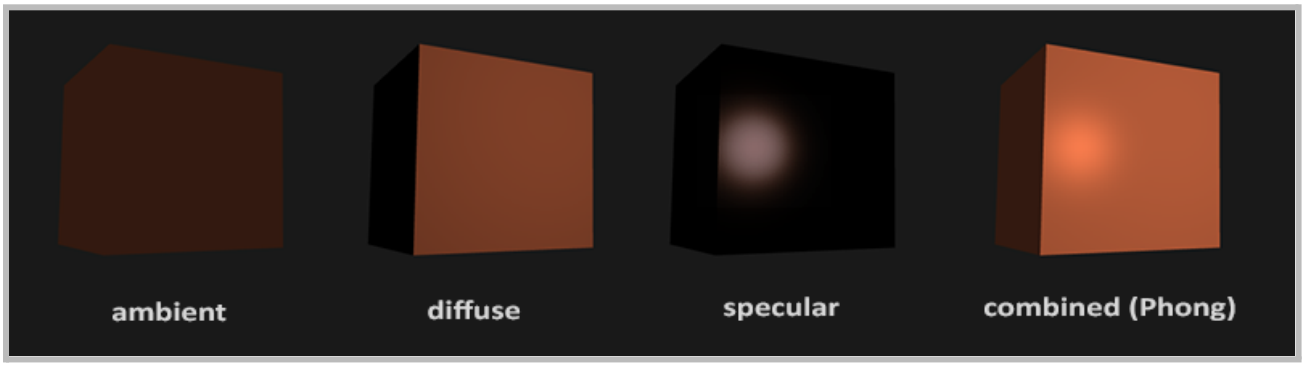
Зараз небагато, на що дивитися, але я обіцяю, що в наступних розділах стане ще цікавіше.

Якщо у вас виникли труднощі з визначенням того, де всі фрагменти коду поєднуються в програмі загалом, перевірте вихідний код тут та уважно пройдіться через код/коментарі.

Тепер, коли ми маємо неабиякі знання про кольори та створили основну сцену для експериментів з освітленням, ми можемо перейти до наступного розділу, де починається справжня магія .

Основне освітлення

Освітлення в реальному світі є надзвичайно складним і залежить від надто багатьох факторів, які ми не можемо собі дозволити розрахувати на обмеженій обчислювальній потужності, яку маємо. Тому освітлення в OpenGL базується на наближеннях до реальності за допомогою спрощених моделей, які набагато легше обробляти та виглядають відносно схожими. Ці моделі освітлення базуються на фізиці світла, як ми її розуміємо. Одна з таких моделей називаєтьсяМодель освітлення Phong. Основні будівельні блоки моделі освітлення Phong складаються з 3 компонентів: навколишнього, дифузного та дзеркального освітлення. Нижче ви можете побачити, як виглядають ці компоненти освітлення окремо та разом:



- **Навколишнє освітлення: навіть коли темно, як правило, десь у світі ще є світло (місяць, далеке світло), тому об'єкти майже ніколи не бувають повністю темними. Щоб імітувати це, ми використовуємо константу навколишнього освітлення, яка завжди надає об'єкту певного кольору.**
- **Розсіяне освітлення: імітує спрямований вплив світлового об'єкта на об'єкт. Це найбільш візуально значущий компонент моделі освітлення. Чим більше частина предмета звернена до джерела світла, тим яскравішим він стає.**
- **Дзеркальне освітлення: імітує яскраву пляму світла, яка з'являється на блискучих об'єктах. Дзеркальні відблиски більше залежать від кольору світла, ніж від кольору об'єкта.**

Щоб створити візуально цікаві сцени, ми хочемо принаймні імітувати ці 3 компоненти освітлення. Ми почнемо з найпростішого: *навколишнє освітлення*.

Навколишнє освітлення

Зазвичай світло виходить не від одного джерела світла, а від багатьох джерел світла, розкиданих навколо нас, навіть якщо їх не видно відразу. Одна з властивостей світла полягає в тому, що воно може розсіюватися та відбиватися в багатьох напрямках, досягаючи плям, які не видно безпосередньо; таким чином світло може *відбиватися* на інших поверхнях і опосередковано впливати на освітлення об'єкта. Алгоритми, які враховують це, називаються глобальне освітленняалгоритми, але вони складні та дорогі для розрахунку.

3

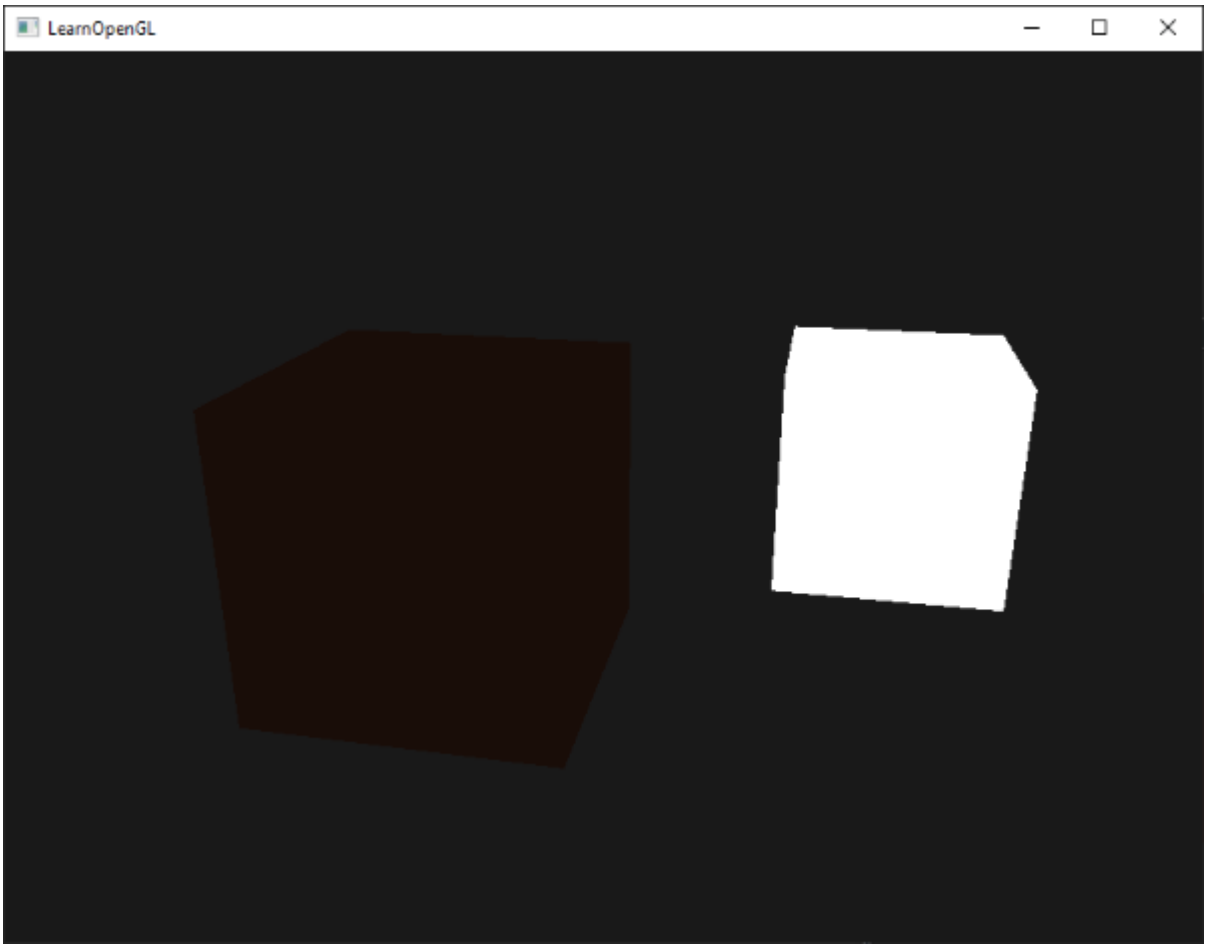
Оскільки ми не є великими прихильниками складних і дорогих алгоритмів, ми почнемо з використання дуже спрощеної моделі глобального освітлення, а саменавколишнє освітлення. Як ви бачили в попередньому розділі, ми використовуємо невеликий постійний (світлий) колір, який ми додаємо до кінцевого кінцевого кольору фрагментів об'єкта, таким чином створюючи вигляд, що розсіяне світло завжди є, навіть коли немає прямого джерела світла.

Додати навколишнє освітлення до сцени дуже просто. Ми беремо колір світла, множимо його на невеликий постійний фактор навколишнього середовища, множимо його на колір об'єкта та використовуємо його як колір фрагмента в шейдері кубічного об'єкта. :

```
void main()

{
```

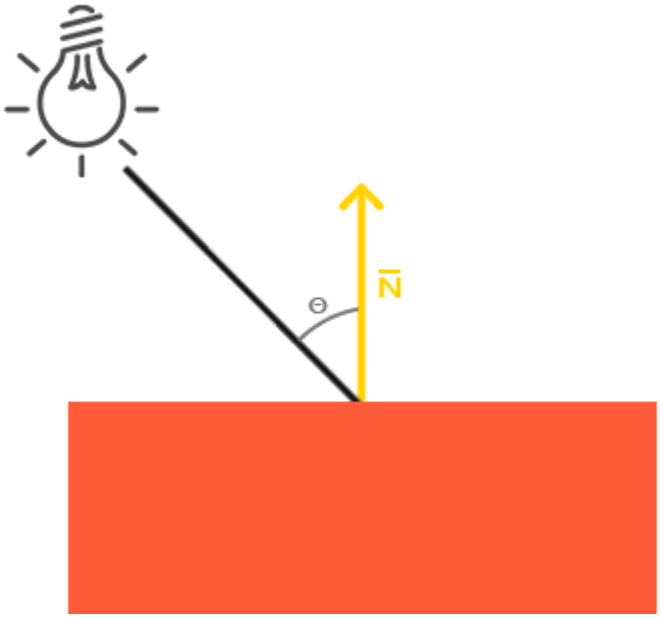
```
    float ambientStrength = 0.1;

    vec3 ambient = ambientStrength * lightColor;



    vec3 result = ambient * objectColor;

    FragColor = vec4(result, 1.0);

}
```

Якщо ви зараз запустите програму, ви помітите, що перший етап освітлення тепер успішно застосовано до об’єкта. Об’єкт досить темний, але не повністю, оскільки застосовано навколишнє освітлення (зверніть увагу, що світловий куб не впливає, оскільки ми використовуємо інший шейдер). Це має виглядати приблизно так:



Розсіяне освітлення

Навколишнє освітлення саме по собі не дає найцікавіших результатів, але розсіяне освітлення почне справляти значний візуальний вплив на об’єкт. Розсіяне освітлення надає об’єкту більшої яскравості, чим ближче його фрагменти розташовані до світлових променів від джерела світла. Щоб краще зрозуміти дифузне освітлення, подивіться на наступне зображення:



Ліворуч знаходимо джерело світла, промінь якого спрямований на окремий фрагмент нашого об'єкта. Нам потрібно виміряти, під яким кутом світловий промінь торкається фрагмента. Якщо промінь світла перпендикулярний до поверхні об'єкта, світло має найбільший вплив. Щоб виміряти кут між світловим променем і фрагментом, ми використовуємо те, що називається анормальний вектор, тобто вектор, перпендикулярний до поверхні фрагмента (тут зображено жовтою стрілкою); ми дійдемо до цього пізніше. Потім кут між двома векторами можна легко обчислити за допомогою скалярного добутку.

Можливо, ви пам’ятаєте з розділу про перетворення, що чим менший кут між двома одиничними векторами, тим більше скалярний добуток нахилений до значення 1. Коли кут між обома векторами дорівнює 90 градусів, скалярний добуток дорівнює 0. Те саме стосується

θ

�: більший

θ

�стає, тим менший вплив має мати світло на колір фрагмента.

Зауважте, що для отримання (лише) косинуса кута між обома векторами ми працюватимемо з *одиничними векторами* (векторами довжини 1), тому нам потрібно переконатися, що всі вектори нормалізовані, інакше

скалярний добуток повертає більше, ніж просто косинус (див. Перетворення).

Таким чином, отриманий скалярний добуток повертає скаляр, який ми можемо використовувати для обчислення впливу світла на колір фрагмента, що призводить до різного освітлення фрагментів залежно від їхньої орієнтації на світло.

Отже, що нам потрібно для розрахунку дифузного освітлення:

- Нормальний вектор: вектор, перпендикулярний до вершини' поверхні.
- Спрямований світловий промінь: напрямний вектор, який є вектором різниці між положенням світла та положенням фрагмента. Щоб обчислити цей світловий промінь, нам потрібен вектор позиції світла та вектор позиції фрагмента.

Нормальні вектори

Нормальний вектор — це (одиничний) вектор, перпендикулярний до поверхні вершини. Оскільки вершина сама по собі не має поверхні (це лише одна точка в просторі), ми отримуємо нормальний вектор, використовуючи його оточуючі вершини, щоб визначити поверхню вершини. Ми можемо використати маленьку хитрість, щоб обчислити вектори нормалей для всіх вершин куба за допомогою перехресного добутку, але оскільки 3D-куб не є складною формою, ми можемо просто вручну додати їх до даних вершин. Оновлений масив даних вершин можна знайти тут. Спробуйте уявити, що нормалі справді є векторами, перпендикулярними до поверхні кожної площини (куб складається з 6 площин).

Оскільки ми додали додаткові дані до масиву вершин, нам слід оновити вершинний шейдер куба:

```
#version 330 core
```

```
layout (location = 0) in vec3 aPos;

layout (location = 1) in vec3 aNormal;

...
```

Тепер, коли ми додали нормальний вектор до кожної з вершин і оновили вершинний шейдер, ми також повинні оновити покажчики атрибутів вершин. Зверніть увагу, що куб джерела світла використовує той самий масив вершин для даних вершин, але шейдер лампи не використовує щойно додані нормальні вектори. Нам не потрібно оновлювати шейдери або конфігурації атрибутів лампи, але ми повинні принаймні змінити покажчики атрибутів вершин, щоб відобразити новий розмір масиву вершин:

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);

glEnableVertexAttribArray(0);
```

Ми хочемо використовувати лише перші 3 плаваючі числа кожної вершини та ігнорувати останні 3 плаваючі числа, тому нам потрібно лише оновити *параметр stride* до 6 розміру `float`, і ми готові.

> Це може виглядати неефективно з використанням вершинних даних, які не повністю використовуються ламповим шейдером, але вершинні дані вже збережені в пам’яті GPU з об’єкта-контейнера, тому нам не потрібно зберігати нові дані в Пам'ять GPU. Це фактично робить його більш ефективним порівняно з виділенням нового VBO спеціально для лампи.

Усі розрахунки освітлення виконуються у фрагментному шейдері, тому нам потрібно переслати вектори нормалей із вершинного шейдера до фрагментного шейдера. Давайте зробимо це:

```
out vec3 Normal;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    Normal = aNormal;
}
```

Що залишилося зробити, це оголосити відповідну вхідну змінну у фрагментному шейдері:

```
in vec3 Normal;
```

Розрахунок дифузного кольору

Тепер у нас є вектор нормалі для кожної вершини, але нам все ще потрібні вектор позиції світла та вектор позиції фрагмента. Оскільки позиція світла є однією статичною змінною, ми можемо оголосити її як уніфіковану у фрагментному шейдері:

```
uniform vec3 lightPos;
```

А потім оновіть уніформу в циклі візуалізації (або поза ним, оскільки вона не змінюється для кадру). Ми використовуємо вектор `lightPos`, оголошений у попередньому розділі, як розташування джерела дифузного світла:

```
lightingShader.setVec3("lightPos", lightPos);
```

Тоді останнє, що нам потрібно, це фактичне положення фрагмента. Ми збираємося виконати всі розрахунки освітлення у світовому просторі, тому нам потрібна позиція вершини, яка спочатку знаходиться у світовому просторі. Ми можемо досягти цього, помноживши атрибут позиції вершини лише на матрицю моделі (а не на матрицю перегляду та проекції), щоб перетворити її на координати світового простору. Це можна легко зробити у вершинному шейдері, тому давайте оголосимо вихідну змінну та обчислимо її координати у світовому просторі:

```
out vec3 FragPos;

out vec3 Normal;


void main()

{
```

```
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = aNormal;
}
```

І, нарешті, додайте відповідну вхідну змінну до фрагментного шейдера:

```
in vec3 FragPos;
```

Ця змінна `in` буде інтерпольована з 3 векторів світової позиції трикутника, щоб сформувати вектор `FragPos`, який є пер. -фрагмент світової позиції. Тепер, коли всі необхідні змінні встановлені, ми можемо почати обчислення освітлення.

Перше, що нам потрібно обчислити, це вектор напрямку між джерелом світла та положенням фрагмента. З попереднього розділу ми знаємо, що вектор напрямку світла є вектором різниці між вектором положення світла та вектором положення фрагмента. Як ви, можливо, пам'ятаєте з розділу про перетворення, ми можемо легко обчислити цю різницю, віднімаючи обидва вектори один від одного. Ми також хочемо переконатися, що всі релевантні вектори закінчуються як одиничні вектори, тому ми нормалізуємо як нормаль, так і результуючий вектор напрямку:

```
vec3 norm = normalize(Normal);
vec3 lightDir = normalize(lightPos - FragPos);
```

При розрахунку освітлення ми зазвичай не дбаємо про величину вектора або їх положення; ми дбаємо лише про їхній напрямок. Оскільки ми дбаємо лише про їх напрямок, майже всі обчислення виконуються з одиничними векторами, оскільки це спрощує більшість обчислень (як скалярний добуток). Тому, виконуючи обчислення освітлення, переконайтеся, що ви завжди нормалізуєте відповідні вектори, щоб переконатися, що вони є фактичними одиничними векторами. Забути нормалізувати вектор — поширена помилка.

Далі нам потрібно обчислити дифузний вплив світла на поточний фрагмент, взявши скалярний добуток між `нормою` та `lightDir` вектори. Потім отримане значення множиться на колір світла, щоб отримати дифузний компонент, у результаті чого дифузний компонент темніший, чим більший кут між обома векторами:

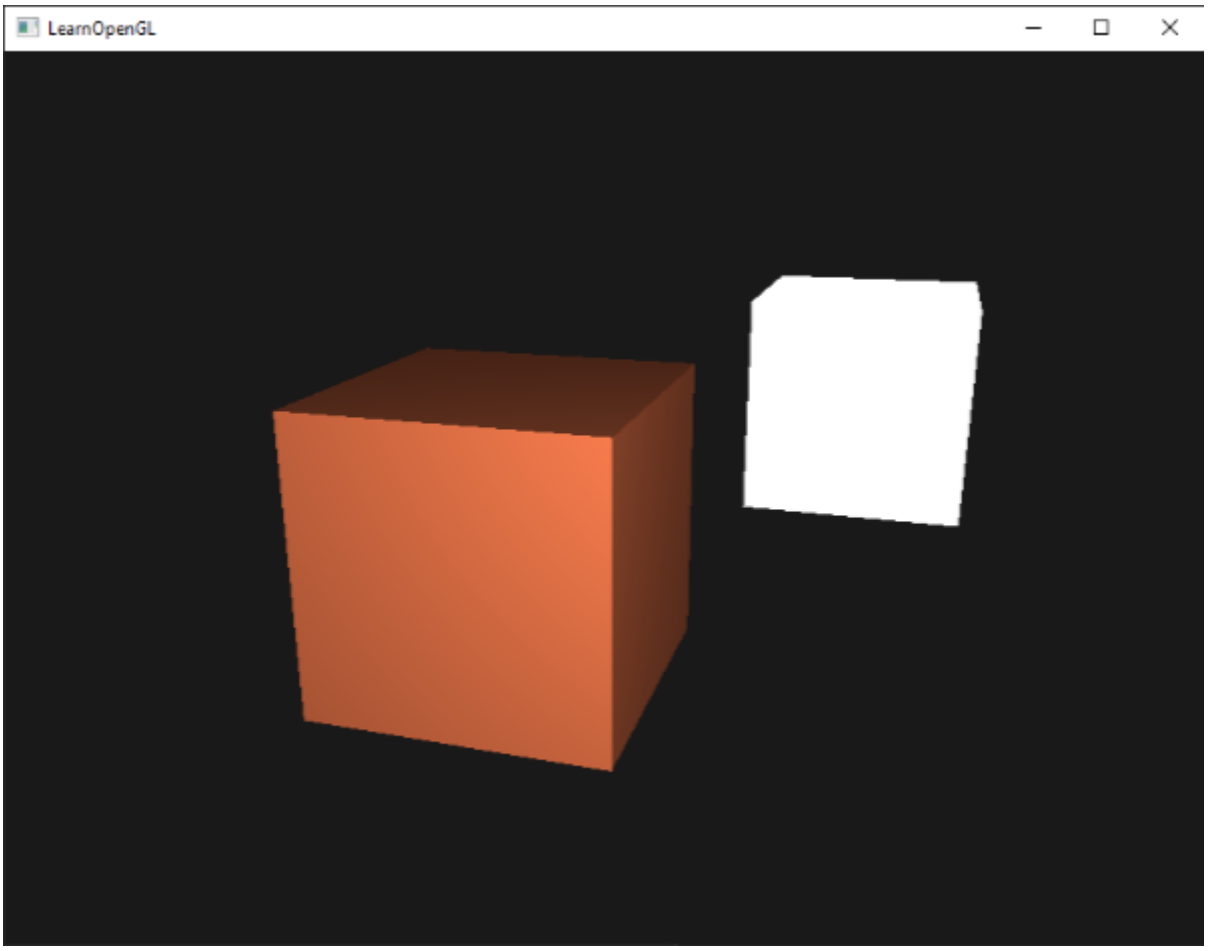
```
float diff = max(dot(norm, lightDir), 0.0);

vec3 diffuse = diff * lightColor;
```

Якщо кут між обома векторами більший за 90 градусів, результат скалярного добутку фактично стане від'ємним, і ми отримаємо від'ємний дифузний компонент. З цієї причини ми використовуємо `макс`функція, яка повертає найвищий з обох своїх параметрів, щоб переконатися, що дифузний компонент (і, отже, кольори) ніколи не стане від'ємним. Освітлення для негативних кольорів насправді не визначено, тому краще триматися подалі від цього, якщо ви не один із тих ексцентричних художників.

Тепер, коли у нас є навколишній і дифузний компоненти, ми додаємо обидва кольори один до одного, а потім множимо результат на колір об’єкта, щоб отримати вихідний колір отриманого фрагмента:

```
vec3 result = (ambient + diffuse) * objectColor;

FragColor = vec4(result, 1.0);
```

Якщо вашу програму (і шейдери) скомпільовано успішно, ви повинні побачити щось на зразок цього:



Ви бачите, що при розсіяному освітленні куб знову починає виглядати як справжній куб. Спробуйте візуалізувати вектори нормалей у своїй голові та

перемістіть камеру навколо куба, щоб побачити, що чим більший кут між вектором нормалі та вектором напрямку світла, тим темнішим стає фрагмент.

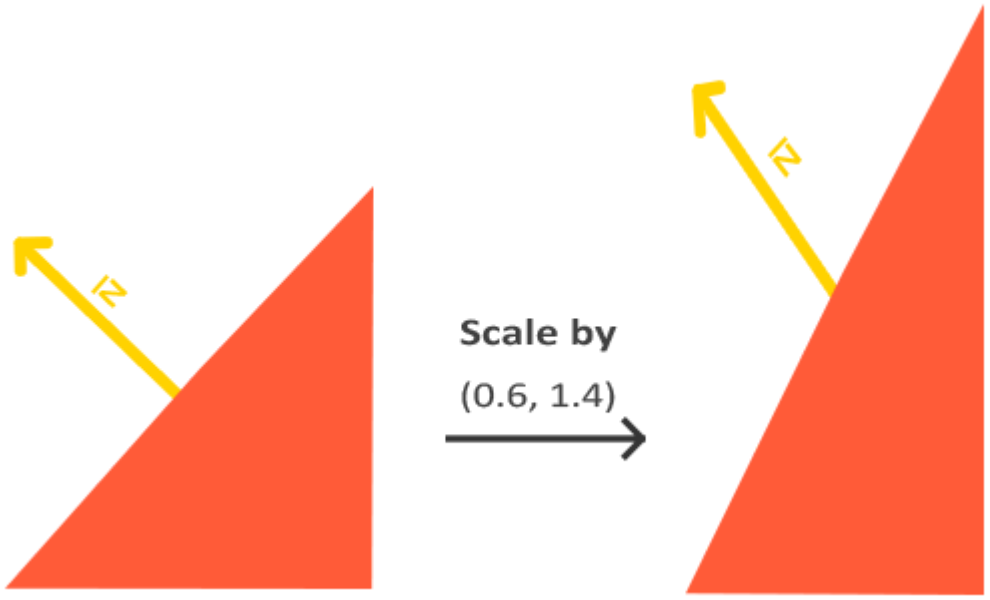
Не соромтеся порівняти свій вихідний код із повним вихідним кодом тут, якщо ви застрягли.

Остання річ

у попередньому розділі ми передали вектор нормалі безпосередньо від вершинного шейдера до фрагментного шейдера. Однак усі обчислення у фрагментному шейдері виконуються у світовому просторі, тож чи не варто нам також трансформувати нормальні вектори у світові просторові координати? Загалом так, але це не так просто, як просто помножити його на модельну матрицю.

По-перше, нормальні вектори є лише векторами напрямків і не представляють певного положення в просторі. По-друге, нормальні вектори не мають однорідної координати (`w` компонента положення вершини). Це означає, що трансляції не повинні впливати на нормальні вектори. Отже, якщо ми хочемо помножити нормальні вектори на модельну матрицю, ми хочемо видалити частину трансляції матриці, взявши верхню ліву 3x3 матрицю модельної матриці (зверніть увагу, що ми також можемо встановити `w` компонент вектора нормалі до `0` та помножити на матрицю 4x4).

По-друге, якщо матриця моделі буде виконувати нерівномірний масштаб, вершини будуть змінені таким чином, що нормальний вектор більше не буде перпендикулярним до поверхні. На наступному зображенні показано вплив такої модельної матриці (з нерівномірним масштабуванням) на нормальний вектор:



Кожного разу, коли ми застосовуємо нерівномірний масштаб (зверніть увагу: рівномірний масштаб змінює лише величину нормалі, а не її напрямок, який легко виправити нормалізацією), вектори нормалей більше не перпендикулярні до відповідної поверхні, що спотворює освітлення.

Хитрість виправити цю поведінку полягає у використанні іншої матриці моделі, спеціально розробленої для нормальних векторів. Ця матриця називаєтьсянормальна матриця і використовує кілька лінійних алгебраїчних операцій, щоб усунути ефект неправильного масштабування нормальних векторів. Якщо ви хочете знати, як обчислюється ця матриця, я пропоную наступну статтю.

Нормальна матриця визначається як 'транспонування оберненої верхньої лівої частини 3x3 матриці моделі'. Фу, це ковток, і якщо ви насправді не розумієте, що це означає, не хвилюйтеся; ми ще не обговорювали інверсні та транспоновані матриці. Зауважте, що більшість ресурсів визначають нормальну матрицю як похідну від матриці моделі-виду, але оскільки ми працюємо у світовому просторі (а не в просторі перегляду), ми виведемо її з матриці моделі.

У вершинному шейдері ми можемо створити нормальну матрицю за допомогоюзворотнийітранспонувати функціонує у вершинному шейдері, який

працює з будь-яким типом матриці. Зауважте, що ми перетворюємо матрицю на матрицю 3x3, щоб гарантувати, що вона втрачає свої властивості перекладу та може множитись на вектор нормалі vec3:

```
Normal = mat3(transpose(inverse(model))) * aNormal;
```

> Інверсія матриць є дорогою операцією для шейдерів, тому, де це можливо, намагайтеся уникати виконання інверсних операцій, оскільки їх потрібно виконувати на кожній вершині вашої сцени. Для цілей навчання це добре, але для ефективної програми ви, ймовірно, захочете обчислити нормальну матрицю на ЦП і надіслати її до шейдерів через уніформу перед малюванням (так само, як матриця моделі).

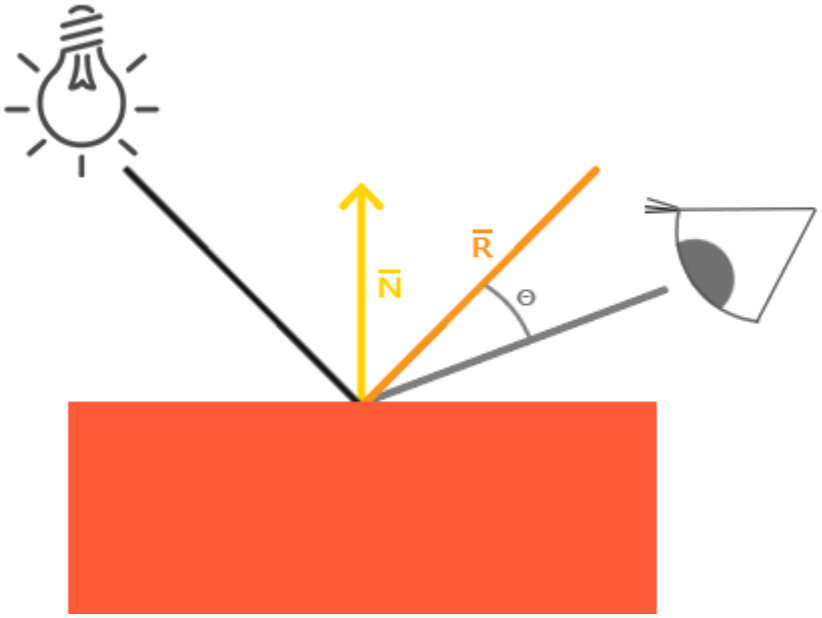
У розділі дифузного освітлення освітлення було нормальним, оскільки ми не робили масштабування об’єкта, тому насправді не було потреби використовувати нормальну матрицю, і ми могли просто помножити нормалі на матрицю моделі . Однак, якщо ви використовуєте нерівномірний масштаб, важливо помножити ваші нормальні вектори на нормальну матрицю.

Дзеркальне освітлення

Якщо ви ще не виснажені всіма розмовами про освітлення, ми можемо почати завершувати модель освітлення Phong, додавши відблиски.

Подібно до дифузного освітлення, дзеркальне освітлення базується на векторі напрямку світла та нормалях об’єкта, але цього разу воно також базується на напрямку огляду, наприклад. з якого боку гравець дивиться на фрагмент. Дзеркальне освітлення засноване на відбивних властивостях поверхонь. Якщо ми розглядаємо поверхню об’єкта як дзеркало, дзеркальне освітлення є

найсильнішим там, де ми бачимо світло, відбите від поверхні. Ви можете побачити цей ефект на наступному зображенні:



Ми обчислюємо вектор відбиття, відбиваючи напрямок світла навколо вектора нормалі. Потім ми обчислюємо кутову відстань між цим вектором відбиття та напрямком огляду. Чим ближчий кут між ними, тим сильніший вплив дзеркального світла. Отриманий ефект полягає в тому, що ми бачимо трохи відблиску, коли дивимося на напрямок світла, відбитого поверхнею.

Вектор огляду — це одна додаткова змінна, яка нам потрібна для дзеркального освітлення, яку ми можемо обчислити, використовуючи позицію глядача у світовому просторі та позицію фрагмента. Потім ми обчислюємо інтенсивність віддзеркалення, множимо її на колір світла та додаємо це до навколишнього та дифузного компонентів.

Ми вирішили виконати розрахунки освітлення у світовому просторі, але більшість людей, як правило, віддає перевагу робити освітлення у просторі видимості. Перевагою простору перегляду є те, що позиція глядача завжди `(0,0,0)`, тому ви вже отримали позицію глядача безкоштовно. Однак я вважаю розрахунок освітлення у світовому просторі більш інтуїтивно зрозумілим для цілей навчання. Якщо ви все ще бажаєте розрахувати освітлення в просторі огляду, вам потрібно також

трансформувати всі відповідні вектори за допомогою матриці огляду (не забудьте також змінити нормальну матрицю).

Щоб отримати світові просторові координати глядача, ми просто беремо вектор позиції об’єкта камери (яким, звичайно, є глядач). Отже, давайте додамо іншу форму до шейдера фрагментів і передамо вектор позиції камери шейдеру:

```
uniform vec3 viewPos;
```

```
lightingShader.setVec3("viewPos", camera.Position);
```

Тепер, коли ми маємо всі необхідні змінні, ми можемо обчислити дзеркальну інтенсивність. Спочатку ми визначаємо значення інтенсивності віддзеркалення, щоб надати відблиску середнього яскравого кольору, щоб воно не справляло надто сильного впливу:

```
float specularStrength = 0.5;
```

Якщо ми встановимо це значення `1.0f`, ми отримаємо справді яскравий дзеркальний компонент, який занадто великий для коралового куба. У наступному розділі ми поговоримо про правильне налаштування всіх цих інтенсивностей освітлення та про те, як вони впливають на об’єкти. Далі обчислюємо вектор напрямку огляду та відповідний вектор відображення вздовж нормальної осі:

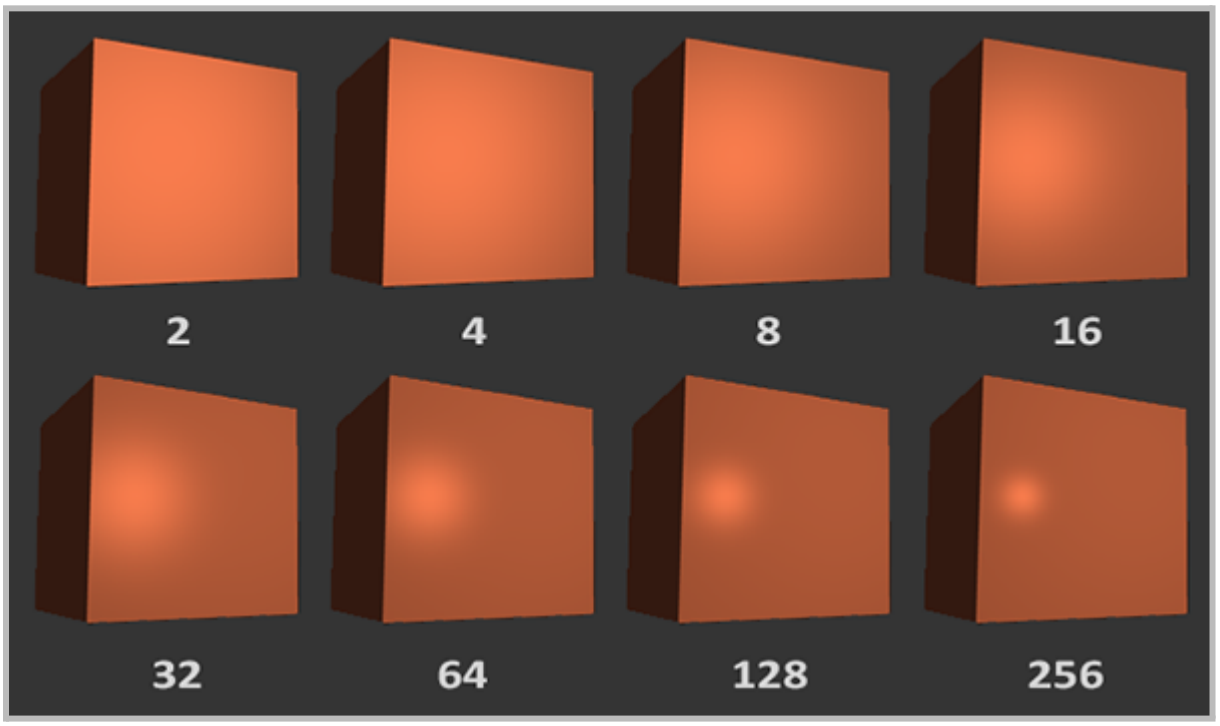
```
vec3 viewDir = normalize(viewPos - FragPos);

vec3 reflectDir = reflect(-lightDir, norm);
```

Зауважте, що ми заперечуємо вектор `lightDir`. Функція `reflect` очікує, що перший вектор буде вказувати від джерела світла до положення фрагмента, але `lightDir` вектор наразі спрямований навпаки: від фрагмента до джерела світла (це залежить від порядку віднімання раніше, коли ми обчислювали вектор `lightDir`). Щоб переконатися, що ми отримуємо правильний вектор `reflect`, ми змінюємо його напрямок, спершу заперечуючи вектор `lightDir`. Другий аргумент передбачає нормальний вектор, тому ми надаємо нормалізований `norm` вектор.

Тоді те, що залишилося зробити, це фактично обчислити дзеркальний компонент. Це досягається за такою формулою:

```
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);

vec3 specular = specularStrength * spec * lightColor;
```

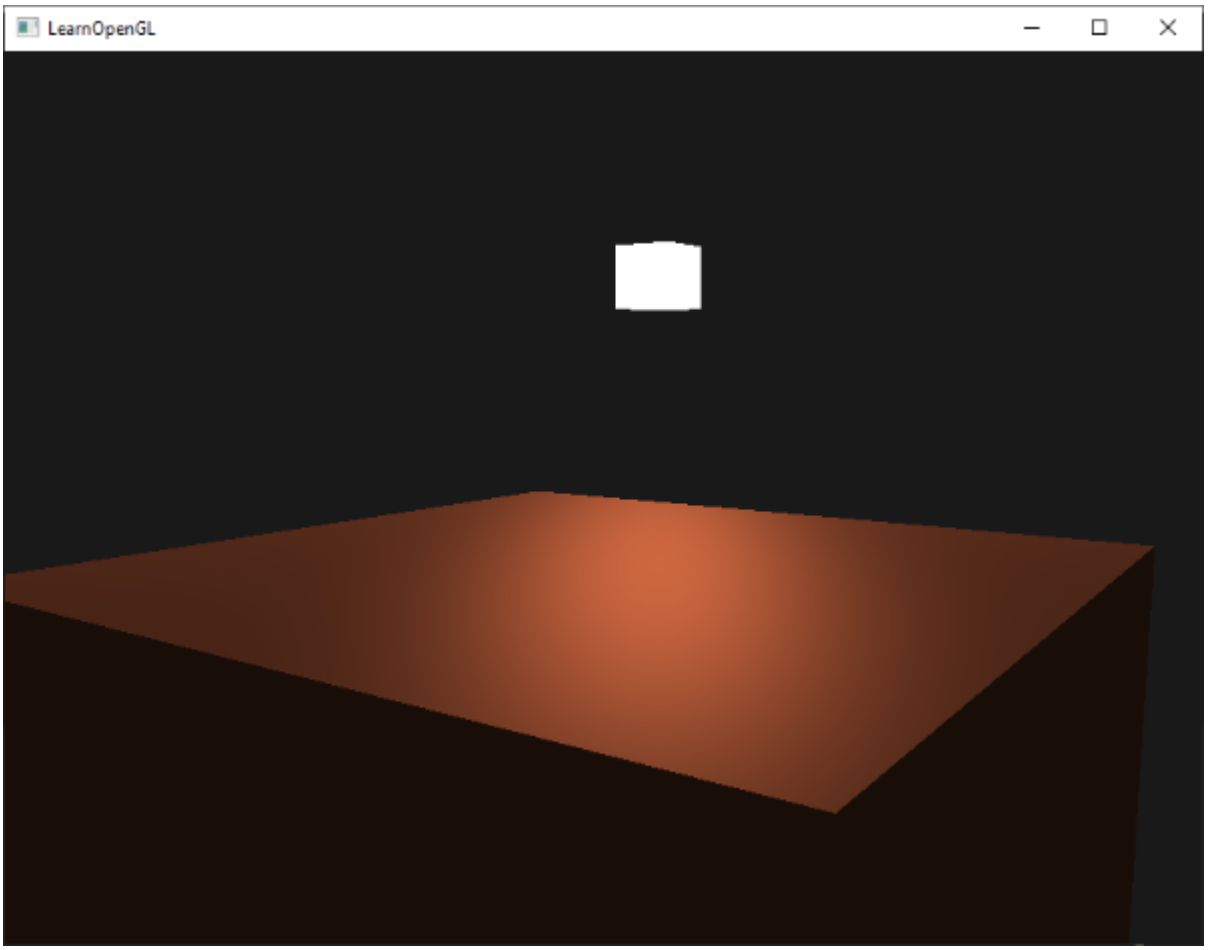
Спочатку ми обчислюємо скалярний добуток між напрямком огляду та напрямком відображення (і переконаємось, що він не є від'ємним), а потім підносимо його до степеня 32. Це 32 значення є **блиск**значення виділення. Чим вище значення блиску об'єкта, тим більше він належним чином відбиває світло, а не розсіює його навколо, і, таким чином, меншим стає світло. Нижче ви можете побачити зображення, яке демонструє візуальний вплив різних значень блиску:



Ми не хочемо, щоб дзеркальний компонент надто відволікав, тому ми залишаємо експоненту 32. Єдине, що залишилося зробити, це додати його до компонентів ambient і diffuse і помножити об'єднаний результат на колір об'єкта:

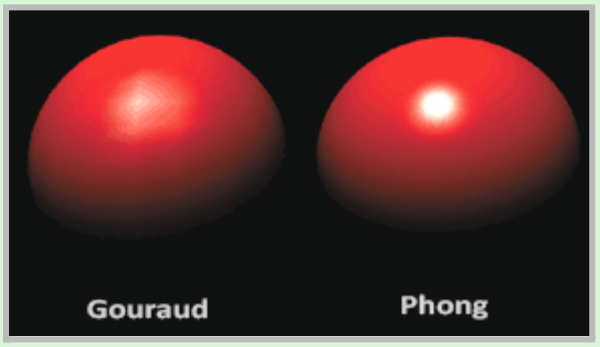
```
vec3 result = (ambient + diffuse + specular) * objectColor;

FragColor = vec4(result, 1.0);
```

Тепер ми розрахували всі компоненти освітлення моделі освітлення Фонга. Виходячи з вашої точки зору, ви повинні побачити щось на зразок цього:



Ви можете знайти повний вихідний код програми [тут](#).

На початку шейдерів освітлення розробники реалізовували модель освітлення Phong у вершинному шейдері. Перевага створення освітлення у вершинному шейдері полягає в тому, що це набагато ефективніше, оскільки, як правило, є набагато менше вершин порівняно з фрагментами, тому (дорогі) обчислення освітлення виконуються рідше. Однак кінцеве значення кольору у вершинному шейдері є кінцевим кольором освітлення лише цієї вершини, а значення кольорів оточуючих фрагментів є результатом інтерпольованих кольорів освітлення. В результаті освітлення було не дуже реалістичним, якщо не використовувати велику кількість вершин:



Коли модель освітлення Phong реалізована у вершинному шейдері, вона викликаєтьсяШтрихування ГуразамістьФонг затінення. Зауважте, що через інтерполяцію освітлення виглядає дещо неправильним. Затінення Phong дає більш плавне освітлення.

На даний момент ви повинні почати бачити, наскільки потужні шейдери. Маючи невелику кількість інформації, шейдери можуть обчислити, як освітлення впливає на кольори фрагментів усіх наших об'єктів. У наступних розділах ми глибше розглянемо, що ми можемо робити з моделлю освітлення.

вправи

- На даний момент джерелом світла є нудне статичне джерело світла, яке не рухається. Спробуйте з часом перемістити джерело світла навколо сцени, використовуючи будь-яке з них`grix`або`cos`. Спостереження за зміною освітлення з часом дає вам гарне розуміння моделі освітлення Фонга: рішення.
- Пограйте з різними навколишніми, дифузними та дзеркальними силами та подивіться, як вони впливають на результат. Також поекспериментуйте з коефіцієнтом блиску. Спробуйте зрозуміти, чому певні значення мають певний візуальний результат.
- Виконуйте затінення Phong у просторі перегляду замість світового простору: рішення.
- Застосувати затінення Гуро замість затінення Фонга. Якщо ви зробили все правильно, освітлення має виглядати дещо нерівним (особливо

відблиски) з об’єктом-кубом. Спробуйте пояснити, чому це виглядає так дивно: рішення.

Матеріали

У реальному світі кожен об’єкт по-різному реагує на світло. Сталеві предмети часто блискучіші, ніж, наприклад, глиняна ваза, а дерев’яний контейнер не реагує на світло так само, як сталевий контейнер. Деякі об’єкти відбивають світло без значного розсіювання, що призводить до невеликих дзеркальних відблисків, а інші сильно розсіюються, надаючи відблиску більший радіус. Якщо ми хочемо симулювати декілька типів об’єктів у OpenGL, ми маємо визначитиматеріалвластивості, характерні для кожної поверхні.

У попередньому розділі ми визначили об’єкт і колір світла, щоб визначити візуальний вихід об’єкта в поєднанні з компонентом навколишньої та дзеркальної інтенсивності. Описуючи поверхню, ми можемо визначити колір матеріалу для кожного з 3 компонентів освітлення: навколишнього, дифузного та дзеркального освітлення. Вказавши колір для кожного з компонентів, ми маємо точний контроль над кольором поверхні. Тепер додайте компонент блиску до цих 3 кольорів, і ми отримаємо всі необхідні властивості матеріалу:

```
#version 330 core

struct Material {
```

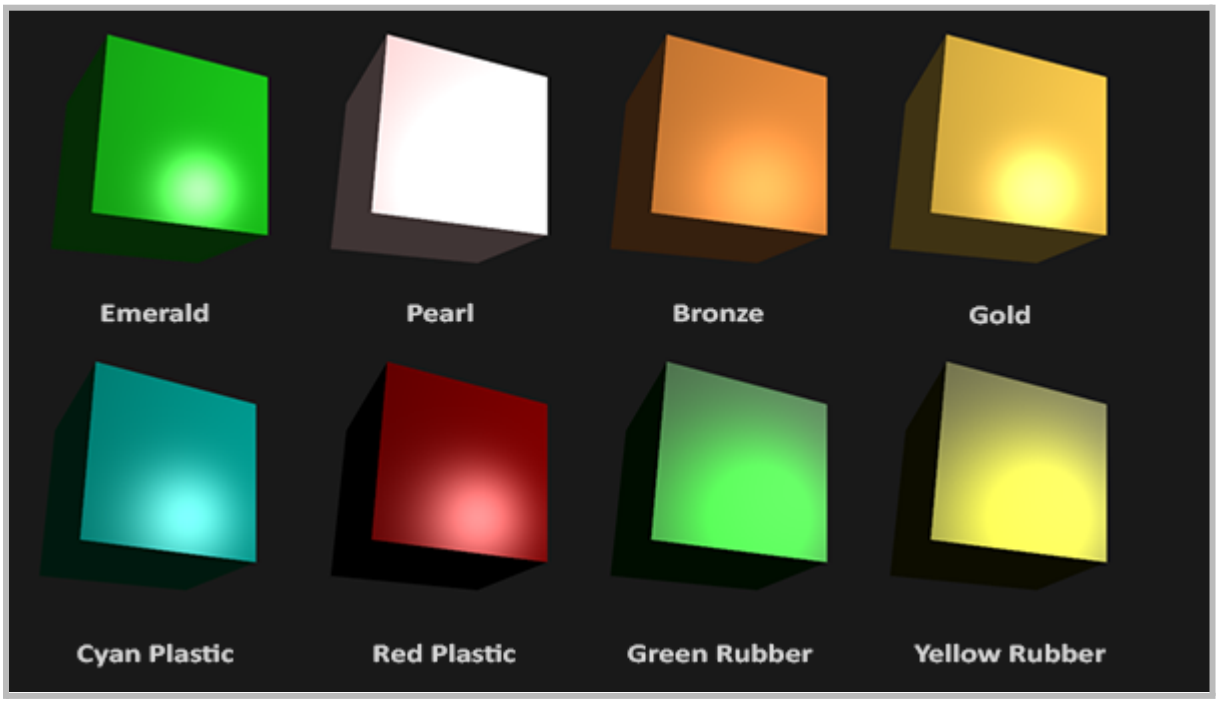
```
    vec3 ambient;

    vec3 diffuse;

    vec3 specular;

    float shininess;

};


uniform Material material;
```

У шейдері фрагментів ми створюємо `struct` для зберігання властивостей матеріалу поверхні. Ми також можемо зберігати їх як окремі уніфіковані значення, але якщо зберігати їх як структуру, це буде більш організовано. Спочатку ми визначаємо макет структури, а потім просто оголошуємо уніфіковану змінну з новоствореною структурою як її тип.

Як бачите, ми визначаємо колірний вектор для кожного з компонентів освітлення Phong. Вектор матеріалу `ambient` визначає, який колір відбиває поверхня під зовнішнім освітленням; зазвичай він збігається з кольором поверхні. `дифузний` вектор матеріалу визначає колір поверхні під дифузним освітленням. Розсіяний колір (як і навколишнє освітлення) встановлюється на потрібний колір поверхні. Вектор `specular` матеріалу встановлює колір дзеркального відблиску на поверхні (або, можливо, навіть відбиває колір, характерний для поверхні). Нарешті, `блиск` впливає на розсіювання/радіус дзеркального відблиску.

За допомогою цих 4 компонентів, які визначають матеріал об'єкта, ми можемо імітувати багато матеріалів реального світу. Таблиця, яку можна знайти на сайті devernay.free.fr, показує список властивостей матеріалів, які імітують

реальні матеріали, знайдені у зовнішньому світі. На наступному зображенні показано, як деякі з цих реальних матеріальних цінностей справляють на наш куб:



Як бачите, правильне визначення властивостей матеріалу поверхні змінює наше сприйняття об’єкта. Ефекти чітко помітні, але для більш реалістичних результатів нам потрібно буде замінити куб чимось складнішим. У розділах Завантаження моделі ми обговоримо більш складні форми.

З'ясувати правильні параметри матеріалу для об'єкта - важкий подвиг, який здебільшого вимагає експериментів і великого досвіду. Це не така рідкість, коли неправильно розміщений матеріал повністю руйнує візуальну якість об’єкта.

Давайте спробуємо реалізувати таку систему матеріалів у шейдерах.

Накладні матеріали

2

Ми створили однорідну структуру матеріалу у фрагментному шейдері, тому далі ми хочемо змінити обчислення освітлення відповідно до нових

властивостей матеріалу. Оскільки всі змінні матеріалу зберігаються у структурі, ми можемо отримати до них доступ із уніформи `material`:

```
void main()

{

    // ambient

    vec3 ambient = lightColor * material.ambient;

    

    // diffuse

    vec3 norm = normalize(Normal);

    vec3 lightDir = normalize(lightPos - FragPos);

    float diff = max(dot(norm, lightDir), 0.0);

    vec3 diffuse = lightColor * (diff * material.diffuse);



    // specular

    vec3 viewDir = normalize(viewPos - FragPos);

    vec3 reflectDir = reflect(-lightDir, norm);

    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);

    vec3 specular = lightColor * (spec * material.specular);
```

```
    vec3 result = ambient + diffuse + specular;

    FragColor = vec4(result, 1.0);

}
```

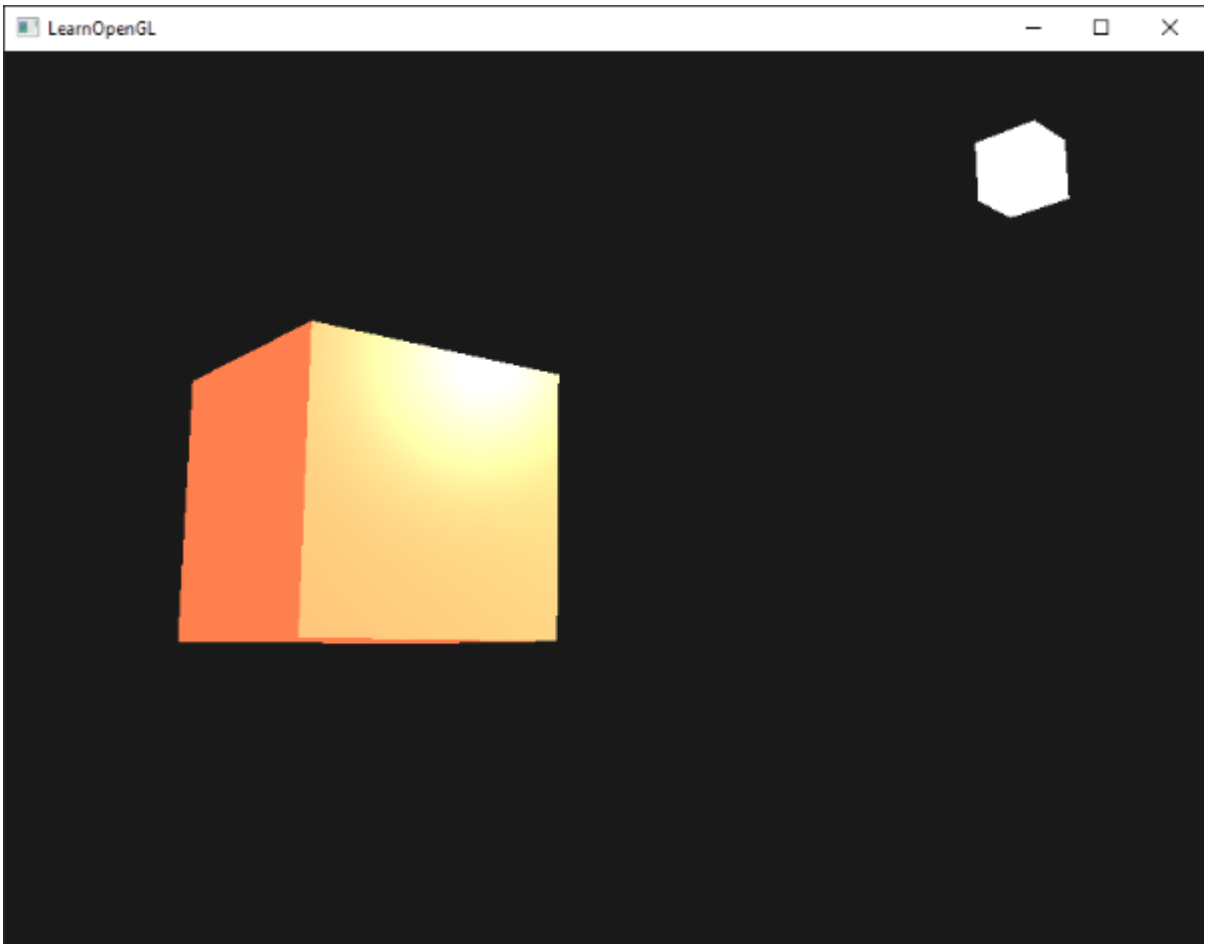
Як бачите, тепер ми отримуємо доступ до всіх властивостей структури матеріалу, де вони нам потрібні, і цього разу обчислюємо кінцевий вихідний колір за допомогою кольорів матеріалу. Кожен із матеріальних атрибутів об’єкта помножується на відповідні компоненти освітлення.

Ми можемо встановити матеріал об’єкта в додатку, встановивши відповідні уніформи. Однак структура в GLSL не є особливою в будь-якому відношенні під час встановлення уніформи; структура насправді діє лише як простір імен уніфікованих змінних. Якщо ми хочемо заповнити структуру, нам доведеться встановити індивідуальну форму, але з префіксом назви структури:

```
lightingShader.setVec3("material.ambient", 1.0f, 0.5f, 0.31f);

lightingShader.setVec3("material.diffuse", 1.0f, 0.5f, 0.31f);

lightingShader.setVec3("material.specular", 0.5f, 0.5f, 0.5f);

lightingShader.setFloat("material.shininess", 32.0f);
```

Ми встановлюємо навколишній і дифузний компонент кольору, який ми хотіли б мати, а для дзеркального компонента об’єкта встановлюємо середньо-яскравий колір; ми не хочемо, щоб дзеркальна складова була надто сильною. Ми також зберігаємо блиск 32.

Тепер ми можемо легко впливати на матеріал об’єкта з програми. Запуск програми дає щось на зразок цього:



Хоча це виглядає не так?

Світлові властивості

Об’єкт надто яскравий. Причина надто яскравого об’єкта полягає в тому, що навколишні, розсіяні та дзеркальні кольори відбиваються з повною силою від будь-якого джерела світла. Джерела світла також мають різну інтенсивність для навколишнього, дифузного та дзеркального компонентів відповідно. У попередньому розділі ми вирішували це, змінюючи навколишню та дзеркальну інтенсивності за допомогою значення сили. Ми хочемо зробити щось подібне, але цього разу, вказавши вектори інтенсивності для кожного з компонентів освітлення. Якби ми візуалізували `lightColor` як `vec3(1.0)`, код виглядав би так:

```
vec3 ambient  = vec3(1.0) * material.ambient;

vec3 diffuse  = vec3(1.0) * (diff * material.diffuse);

vec3 specular = vec3(1.0) * (spec * material.specular);
```

Отже, кожна матеріальна властивість об’єкта повертається з повною інтенсивністю для кожного компонента світла. На ці `vec3(1.0)` значення також можна впливати окремо для кожного джерела світла, і зазвичай це те, чого ми хочемо. Зараз на колір куба повною мірою впливає навколишній компонент об’єкта. Компонент навколишнього середовища не повинен справді мати такий великий вплив на кінцевий колір, тому ми можемо обмежити колір навколишнього середовища, встановивши нижче значення інтенсивності навколишнього освітлення:

```
vec3 ambient = vec3(0.1) * material.ambient;
```

Таким же чином ми можемо впливати на дифузну та дзеркальну інтенсивність джерела світла. Це дуже схоже на те, що ми робили в попередньому розділі; можна сказати, що ми вже створили деякі властивості світла, щоб впливати на кожен компонент освітлення окремо. Ми хочемо створити щось подібне до структури матеріалу для властивостей світла:

```
struct Light {

    vec3 position;
```

```
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

uniform Light light;
```

Джерело світла має різну інтенсивність для `навколишнього`, `розсіяного` та < /span> світиться з повною інтенсивністю. Зверніть увагу, що ми також додали вектор позиції світла до структури. компоненти. Навколишнє освітлення зазвичай має низьку інтенсивність, оскільки ми не хочемо, щоб колір навколишнього середовища був надто домінуючим. Розсіяний компонент джерела світла зазвичай налаштований на точний колір, який ми хотіли б мати; часто яскраво-білого кольору. Дзеркальний компонент зазвичай `дзеркальні`vec3(1.0)

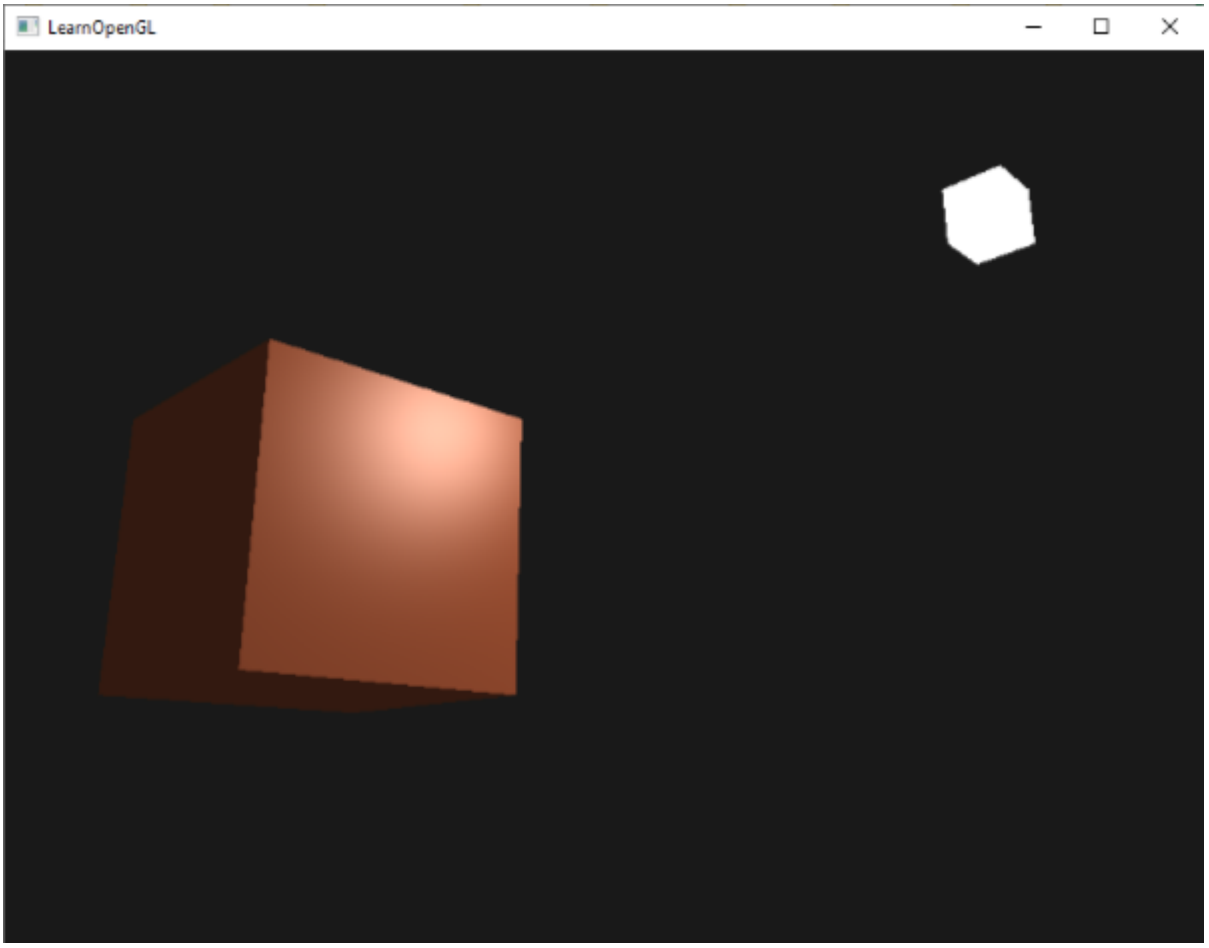
Як і у випадку з уніформою матеріалу, нам потрібно оновити фрагментний шейдер:

```
vec3 ambient  = light.ambient * material.ambient;
vec3 diffuse  = light.diffuse * (diff * material.diffuse);
vec3 specular = light.specular * (spec * material.specular);
```

Потім ми хочемо встановити інтенсивність світла в додатку:

```
lightingShader.setVec3("light.ambient",  0.2f, 0.2f, 0.2f);

lightingShader.setVec3("light.diffuse",  0.5f, 0.5f, 0.5f); // darken diffuse light a bit

lightingShader.setVec3("light.specular", 1.0f, 1.0f, 1.0f);
```

Тепер, коли ми модулювали вплив світла на матеріал об’єкта, ми отримали візуальний результат, який дуже схожий на результат попереднього розділу. Однак цього разу ми отримали повний контроль над освітленням і матеріалом об’єкта:



Змінити візуальні аспекти об’єктів зараз відносно легко. Давайте трохи оживимо ситуацію!

Різні світлі кольори

Досі ми використовували світлі кольори лише для того, щоб варіювати інтенсивність їх окремих компонентів, вибираючи кольори в діапазоні від білого до сірого та чорного, не впливаючи на фактичні кольори об’єкта (лише на його інтенсивність). Оскільки ми тепер маємо легкий доступ до властивостей світла, ми можемо змінювати їх кольори з часом, щоб отримати справді цікаві ефекти. Оскільки все вже налаштовано у фрагментному шейдері, змінити кольори світла легко та одразу створювати деякі дивовижні ефекти:

Як ви бачите, інший колір світла сильно впливає на вихід кольору об’єкта. Оскільки колір світла безпосередньо впливає на те, які кольори може відображати об’єкт (як ви, можливо, пам’ятаєте з розділу Кольори), він має значний вплив на візуальний результат.

Ми можемо легко змінювати кольори світла з часом, змінюючи кольори навколишнього світла та дифузні кольори за допомогою`rpix`і`glfwGetTime`:

```
glm::vec3 lightColor;

lightColor.x = sin(glfwGetTime() * 2.0f);

lightColor.y = sin(glfwGetTime() * 0.7f);

lightColor.z = sin(glfwGetTime() * 1.3f);



glm::vec3 diffuseColor = lightColor   * glm::vec3(0.5f);

glm::vec3 ambientColor = diffuseColor * glm::vec3(0.2f);
```

```
lightingShader.setVec3("light.ambient", ambientColor);

lightingShader.setVec3("light.diffuse", diffuseColor);
```

Спробуйте поекспериментувати з декількома значеннями освітлення та матеріалів і подивіться, як вони впливають на візуальний результат. Ви можете знайти вихідний код програми тут.

вправи

- **Чи можете ви зробити так, щоб зміна кольору світла змінювала колір світлового куба?**
- **Чи можете ви змоделювати деякі об’єкти реального світу, визначивши їхні відповідні матеріали, як ми бачили на початку цього розділу? Зауважте, що значення навколишнього середовища таблиці не збігаються зі значеннями дифузії; вони не враховували інтенсивність світла. Щоб правильно встановити їхні значення, вам потрібно встановити всі інтенсивності світла на `vec3(1.0)`, щоб отримати той самий результат: рішення блакитного пластикового контейнера.**

Карти освітлення

У попередньому розділі ми обговорювали можливість того, що кожен об’єкт має власний унікальний матеріал, який по-різному реагує на світло. Це чудово підходить для надання кожному об’єкту унікального вигляду порівняно з іншими об’єктами, але все ще не надає великої гнучкості щодо візуального виводу об’єкта.

У попередньому розділі ми визначили матеріал для всього об'єкта в цілому. Однак об’єкти в реальному світі зазвичай складаються не з одного матеріалу, а з кількох матеріалів. Подумайте про автомобіль: його екстер’єр складається з блискучої тканини, він має вікна, які частково відображають навколишнє середовище, його шини майже блискучі, тому на них немає дзеркальних відблисків, і він має надзвичайно блискучі диски (якщо ви фактично добре помив вашу машину). Автомобіль також має дифузні та навколишні кольори, які не однакові для всього об’єкта; автомобіль відображає багато різних навколишніх/дифузних кольорів. Загалом, такий об’єкт має різні властивості матеріалу для кожної з його різних частин.

Отже, системи матеріалів у попередньому розділі недостатньо для всіх моделей, крім найпростіших, тому нам потрібно розширити систему, ввівши *diffuse* та < /span> карти. Це дозволяє нам впливати на дифузний (і опосередковано на навколишній компонент, оскільки вони все одно повинні бути однаковими) і дзеркальний компонент об’єкта з набагато більшою точністю. *зеркальні*

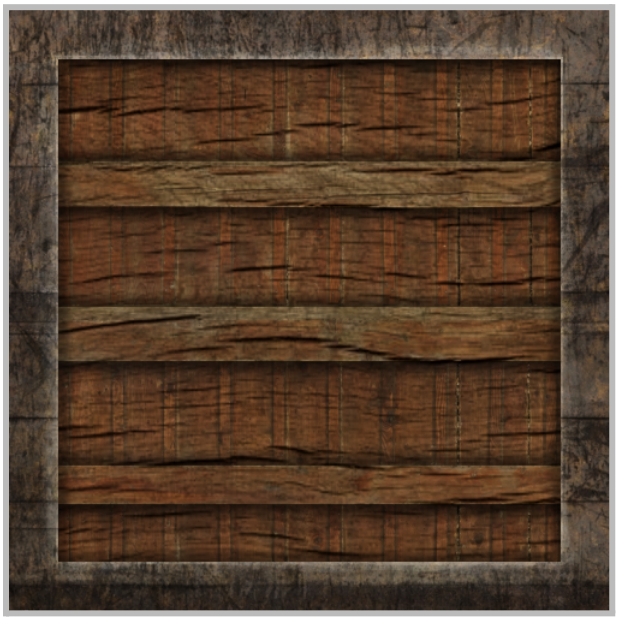
Дифузні карти

Те, що ми хочемо, це якимось чином встановити розсіяні кольори об’єкта для кожного окремого фрагмента. Якась система, де ми можемо отримати значення кольору на основі положення фрагмента на об’єкті?

Напевно, все це має здатися знайомим, і ми вже деякий час використовуємо таку систему. Це схоже на *текстури*, які ми докладно обговорювали в одному з попередніх розділі, і це в основному саме це: текстура. Ми просто

використовуємо іншу назву для того самого основного принципу: використовуємо зображення, обернуте навколо об'єкта, яке ми можемо індексувати для унікальних значень кольорів на фрагмент. У освітлених сценах це зазвичай називається дифузна карта(зазвичай так їх називають 3D-художники до PBR), оскільки зображення текстури представляє всі дифузні кольори об'єкта.

Щоб продемонструвати дифузні карти, ми використаємо наведене нижче зображення дерев'яного контейнера зі сталевим бортиком:



Використання дифузної карти в шейдерах точно так само, як ми показали в розділі про текстури. Однак цього разу ми зберігаємо текстуру як `sampler2D` всередині `материал` структура. Ми замінюємо раніше визначений `vec3` вектор дифузного кольору дифузною картою.

> **Пам'ятайте, що `sampler2D` є так званим непрозорий типце означає, що ми не можемо створювати ці типи, а лише визначаємо їх як уніформи. Якщо структура буде створена не як однорідний (як параметр функції), GLSL може викликати дивні помилки; те саме стосується будь-якої структури, що містить такі непрозорі типи.**

Ми також видаляємо вектор кольору навколишнього матеріалу, оскільки колір навколишнього середовища все одно дорівнює дифузному кольору тепер, коли ми керуємо навколишнім середовищем за допомогою світла. Тому немає необхідності зберігати його окремо:

```
struct Material {
    sampler2D diffuse;
    vec3      specular;
    float     shininess;
};
...
in vec2 TexCoords;
```

> Якщо ви трохи вперті й все одно бажаєте встановити інші значення кольорів навколишнього середовища (окрім значення дифузії), ви можете залишити навколишні кольори vec3, але тоді кольори навколишнього середовища все одно будуть залишаються однаковими для всього об'єкта. Щоб отримати різні значення середовища для кожного фрагмента, вам доведеться використовувати іншу текстуру лише для значень середовища.

Зверніть увагу, що нам знову знадобляться координати текстури у фрагментному шейдері, тому ми оголосили додаткову вхідну змінну. Потім ми просто вибираємо з текстури, щоб отримати значення розсіяного кольору фрагмента:

```
vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
```

Крім того, не забудьте встановити колір навколишнього матеріалу таким же, як і колір дифузного матеріалу:

```
vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
```

І це все, що потрібно для використання дифузної карти. Як бачите, у цьому немає нічого нового, але він забезпечує різке підвищення якості зображення. Щоб це запрацювало, нам потрібно оновити дані вершин за допомогою координат текстури, передати їх як атрибути вершин у фрагментний шейдер, завантажити текстуру та прив'язати текстуру до відповідного блоку текстури.

Оновлені дані про вершини можна знайти тут. Дані про вершини тепер включають позиції вершин, вектори нормалей і координати текстури для кожної з вершин куба. Давайте оновимо вершинний шейдер, щоб приймати координати текстури як атрибут вершини та пересилати їх у фрагментний шейдер:

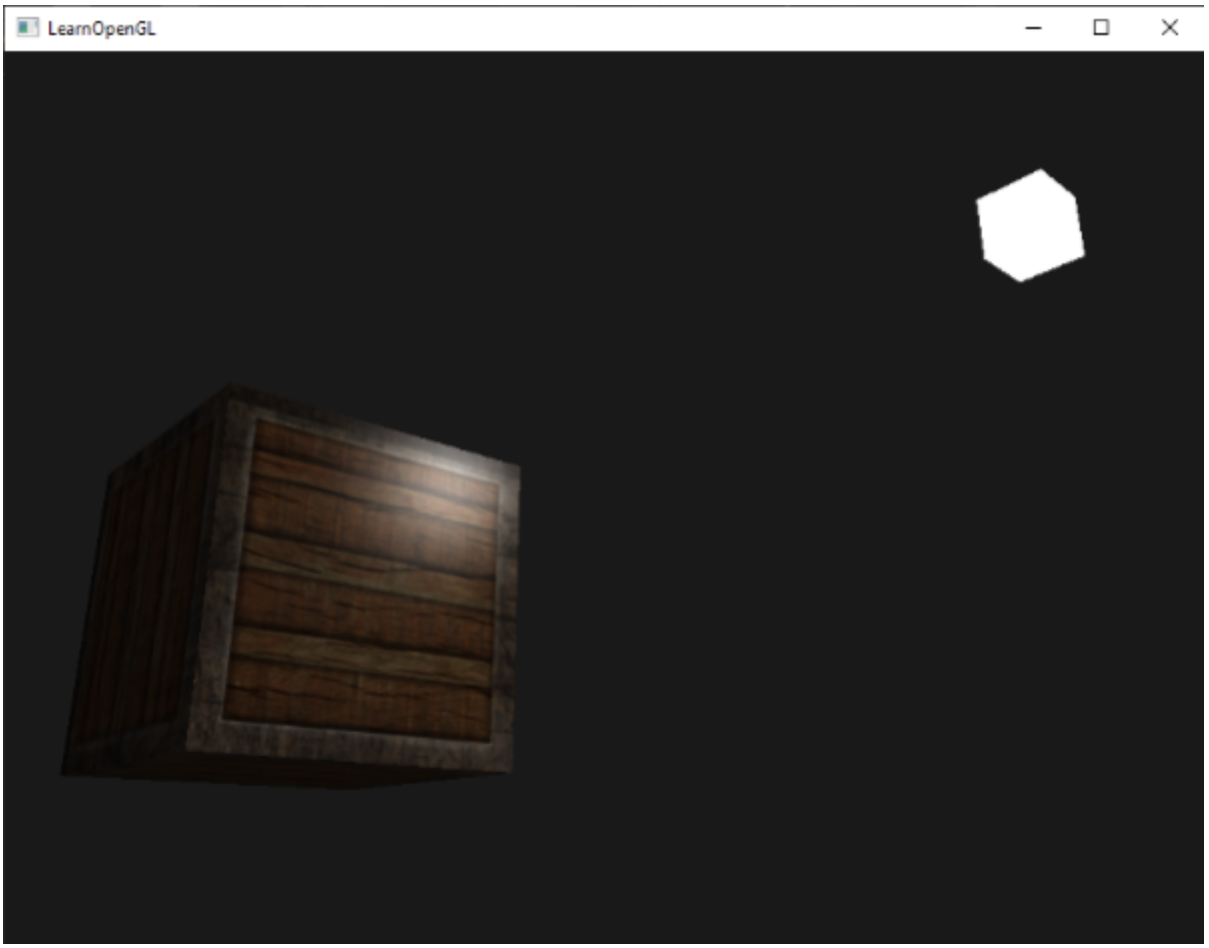
```
#version 330 core

layout (location = 0) in vec3 aPos;

layout (location = 1) in vec3 aNormal;

layout (location = 2) in vec2 aTexCoords;
```

```
...

out vec2 TexCoords;



void main()

{

    ...

    TexCoords = aTexCoords;

}
```

Обов’язково оновіть покажчики атрибутів вершин обох VAO, щоб відповідати новим даним вершин і завантажте зображення контейнера як текстуру. Перед рендерингом куба ми хочемо призначити потрібну текстурну одиницю однорідному семплеру `material.diffuse` і прив’язати контейнерну текстуру до цієї текстурної одиниці:

```
lightingShader.setInt("material.diffuse", 0);

...

glActiveTexture(GL_TEXTURE0);

glBindTexture(GL_TEXTURE_2D, diffuseMap);
```

Тепер, використовуючи дифузну карту, ми знову отримуємо величезний приріст деталей, і цього разу контейнер дійсно починає сяяти (буквально). Тепер ваш контейнер виглядає приблизно так:

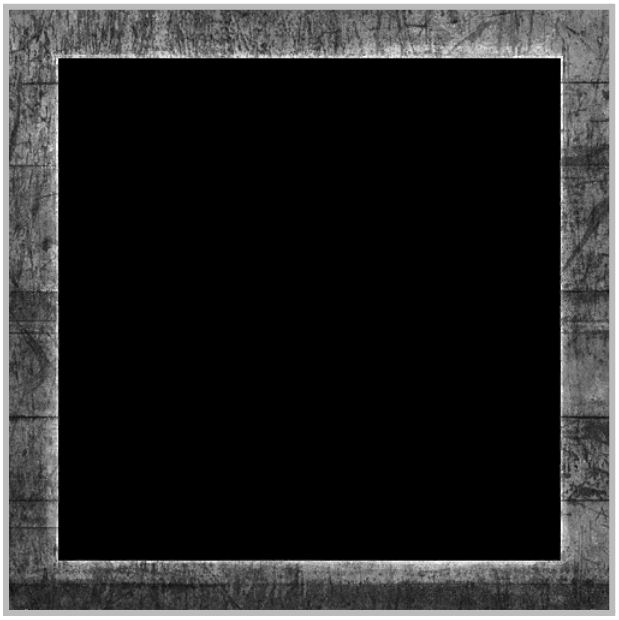


Ви можете знайти повний вихідний код програми тут.

Дзеркальні карти

Ви, мабуть, помітили, що дзеркальне відблискування виглядає дещо дивно, оскільки об'єкт є контейнером, який переважно складається з деревини, а деревина не має таких відблисків. Ми можемо виправити це, встановивши для дзеркального матеріалу об'єкта значення `vec3(0.0)`, але це означатиме, що сталеві межі контейнера також перестануть відображати відблиски, а сталь має відображати дзеркальні відблиски. Ми хотіли б контролювати, які частини об'єкта мають відображати відблиски із різною інтенсивністю. Це проблема, яка звучить знайомо. Випадковість? Я думаю, НЕ.

Ми також можемо використовувати карту текстури лише для відблисків. Це означає, що нам потрібно створити чорно-білу (або кольорову, якщо хочете) текстуру, яка визначає інтенсивність відображення кожної частини об'єкта. Прикладом оглядової карти є таке зображення:



Інтенсивність відблиску залежить від яскравості кожного пікселя зображення. Кожен піксель дзеркальної карти можна відобразити як кольоровий вектор, де чорний колір представляє колірний вектор `vec3(0.0)`, а сірий колірний вектор `vec3(0.5)`, наприклад. Потім у фрагментному шейдері ми вибираємо відповідне значення кольору та множимо це значення на інтенсивність віддзеркалення світла. Чим більше 'білого' піксель, тим вищий результат множення і, отже, тим яскравішим стає дзеркальний компонент об'єкта.

Оскільки контейнер переважно складається з деревини, а дерево як матеріал не повинно мати дзеркальних відблисків, усю дерев'яну частину розсіяної текстури було перетворено на чорний: чорні секції не мають відблисків. Сталевий бордюр контейнера має різну інтенсивність дзеркального відображення, причому сама сталь відносно чутлива до дзеркальних відблисків, тоді як тріщини – ні.

> Технічно деревина також має дзеркальні відблиски, хоча зі значно нижчим значенням блиску (більше розсіювання світла) і меншим впливом, але з

метою навчання ми можемо просто прикинутися, що деревина не реагує на дзеркальне світло.

Використовуючи такі інструменти, як *Photoshop* або *Gimp*, можна відносно легко трансформувати дифузний текстури до такого дзеркального зображення, як це, вирізаючи деякі частини, перетворюючи їх на чорно-білі та збільшуючи яскравість/контраст.

Вибірка дзеркальних карт

Дзеркальна карта схожа на будь-яку іншу текстуру, тому код схожий на код дифузної карти. Переконайтеся, що правильно завантажено зображення та створено об’єкт текстури. Оскільки ми використовуємо інший семплер текстури в тому самому фрагментному шейдері, нам потрібно використовувати інший блок текстури (див. Текстури) для дзеркальної карти, тому давайте прив’яжемо його до відповідного блоку текстури перед рендерингом:

```
lightingShader.setInt("material.specular", 1);

...

glActiveTexture(GL_TEXTURE1);

glBindTexture(GL_TEXTURE_2D, specularMap);
```

Потім оновіть властивості матеріалу фрагментного шейдера, щоб прийняти `sampler2D` як його дзеркальний компонент замість `vec3`:

```
struct Material {
```

```
    sampler2D diffuse;
    sampler2D specular;
    float     shininess;
};
```

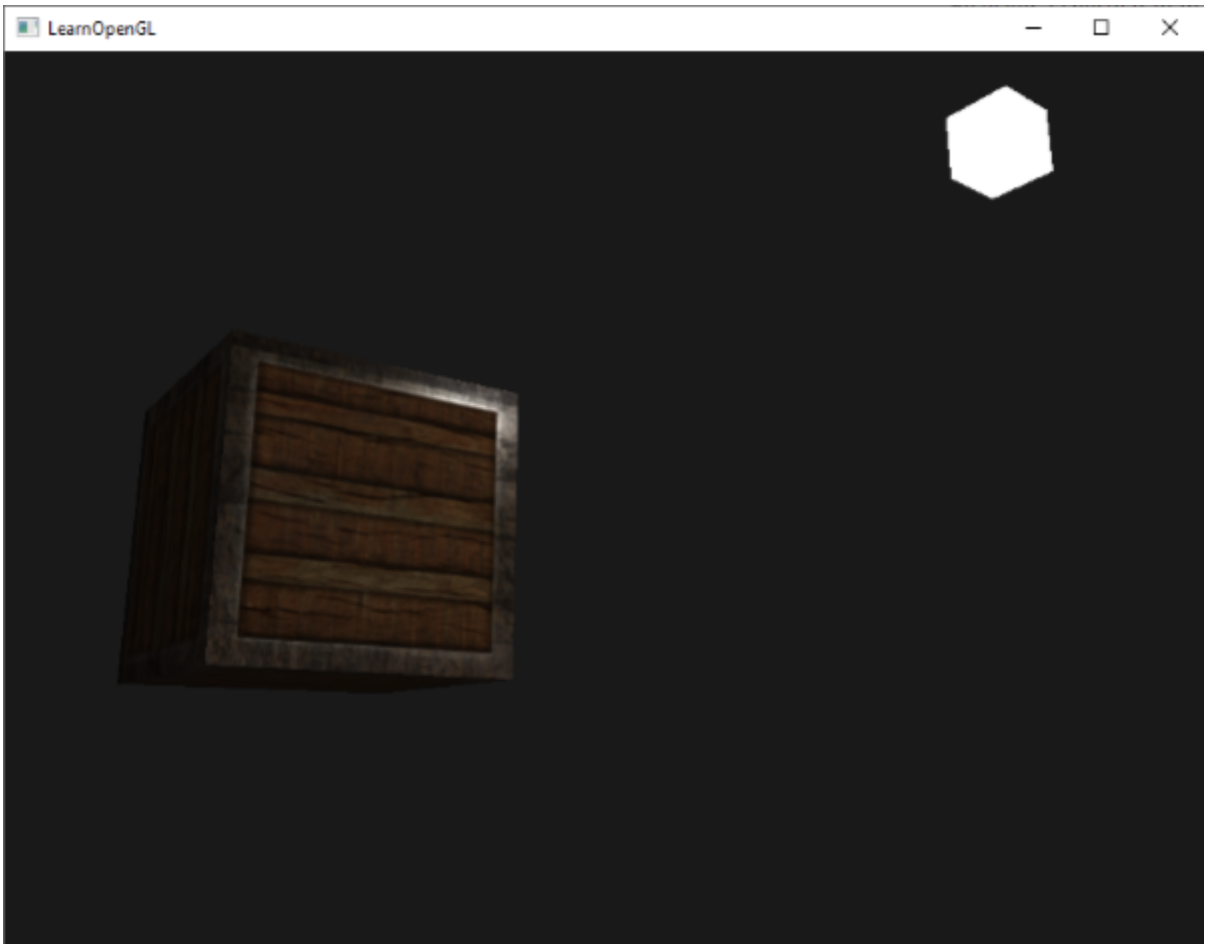
І, нарешті, ми хочемо взяти вибірку дзеркальної карти, щоб отримати відповідну дзеркальну інтенсивність фрагмента:

```
vec3 ambient  = light.ambient  * vec3(texture(material.diffuse, TexCoords));
vec3 diffuse  = light.diffuse  * diff * vec3(texture(material.diffuse, TexCoords));
vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
FragColor = vec4(ambient + diffuse + specular, 1.0);
```

Використовуючи дзеркальну карту, ми можемо вказати з величезною кількістю деталей, які частини об'єкта мають *блискучі* властивості, і ми навіть можемо контролювати відповідну інтенсивність. Відзеркальні карти дають нам додатковий рівень контролю над освітленням поверх дифузної карти.

> Якщо ви не хочете бути надто масовим, ви також можете використати фактичні кольори в дзеркальній карті, щоб не лише встановити інтенсивність дзеркального відображення кожного фрагмента, але й колір дзеркального відблиску. Реально, однак, колір дзеркального відблиску в основному визначається самим джерелом світла, тому він не створює реалістичні візуальні ефекти (ось чому зображення зазвичай чорно-білі: нас дбає лише про інтенсивність).

Якщо ви зараз запустите програму, ви зможете чітко побачити, що матеріал контейнера тепер дуже нагадує справжній дерев'яний контейнер зі сталевими рамами:



Ви можете знайти повний вихідний код програми тут.

За допомогою дифузних і дзеркальних карт ми дійсно можемо додати величезну кількість деталей до відносно простих об'єктів. Ми навіть можемо додати більше деталей до об'єктів, використовуючи інші текстурні карти, наприкладнормальні карти/карти рельєфута/абокарти відображення, але це те, що ми залишимо для наступних розділів. Покажіть свій контейнер усім своїм друзям і родині та будьте задоволені тим фактом, що одного дня наш контейнер може стати ще кращим, ніж він є!

вправи

- **Поекспериментуйте з навколишнім, дифузним і дзеркальним векторами джерела світла та подивіться, як вони впливають на візуальний вихід контейнера.**
- **Спробуйте інвертувати значення кольорів дзеркальної карти у фрагментному шейдері, щоб деревина відображала дзеркальні відблиски, а сталеві межі – ні (зауважте, що через тріщини в сталевій рамці межі все ще мають відблиски, хоча з меншою інтенсивністю): рішення.**
- **Спробуйте створити дзеркальну карту з дифузної текстури, яка використовує фактичні кольори замість чорно-білого, і побачите, що результат виглядає не надто реалістичним. Ви можете скористатися цією кольоровою дзеркальною картою, якщо ви не можете створити її самостійно: результат .**
- **Також додайте те, що вони називають ankарта викидів, яка є текстурою, яка зберігає значення емісії для кожного фрагмента. Значення випромінювання – це кольори, які об'єкт може *випромінювати* так, ніби він сам містить джерело світла; таким чином об'єкт може світитися незалежно від умов освітлення. Карти випромінювання – це часто те, що ви бачите, коли об'єкти в грі світяться (наприклад, очі робота або світлові смуги на контейнері). Додайте наведену нижче текстуру (від creativesam) як карту випромінювання до контейнера, наче літери випромінюють світло: рішення ; результат.**

Світлові ролики

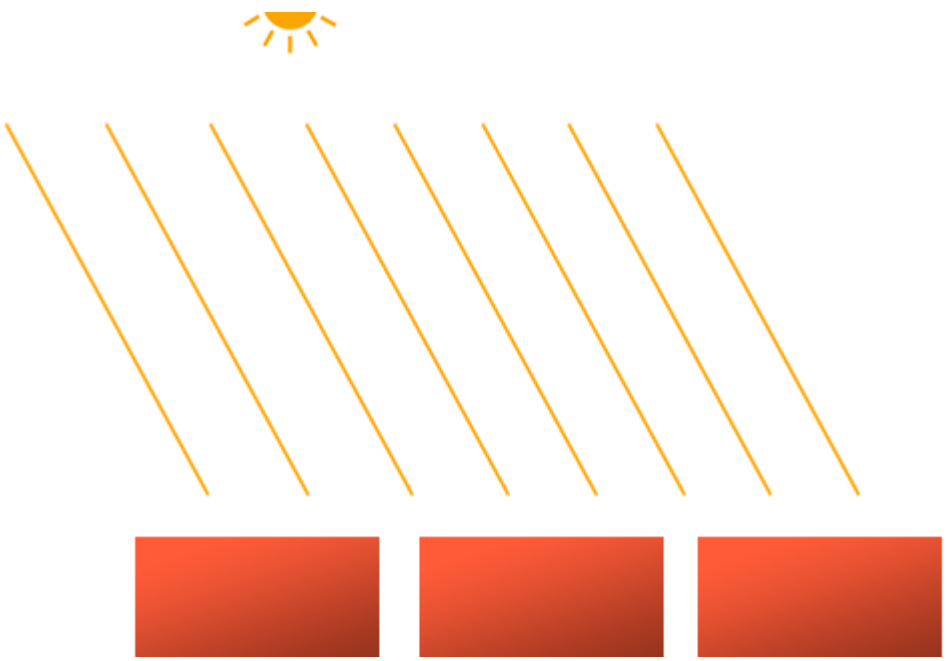
Усе освітлення, яке ми використовували досі, надходило від одного джерела, тобто однієї точки в просторі. Це дає хороші результати, але в реальному світі ми маємо кілька типів світла, кожен з яких діє по-різному. Джерело світла, яке *випромінює* світло на предмети, називається легкий ролик. У цьому розділі ми обговоримо кілька різних типів коліщаток світла. Навчання моделювати різні джерела світла – це ще один інструмент у вашому наборі інструментів для подальшого збагачення вашого середовища.

Спочатку ми обговоримо спрямоване світло, потім точкове світло, яке є продовженням того, що ми мали раніше, і нарешті ми обговоримо прожектори. У наступному розділі ми об'єднаємо декілька з цих різних типів світла в одну сцену.

Спрямоване світло

Коли джерело світла знаходиться далеко, світлові промені, що виходять від джерела світла, майже паралельні один одному. Схоже, що всі промені світла йдуть з одного напрямку, незалежно від того, де знаходиться об'єкт і/або глядач. Коли джерело світла моделюється як *нескінченно* далеко, це називається спрямоване світлооскільки всі його світлові промені мають однаковий напрямок; це не залежить від розташування джерела світла.

Прекрасним прикладом спрямованого джерела світла є сонце, яке ми знаємо. Сонце не нескінченно далеко від нас, але воно настільки далеко, що ми можемо сприймати його як нескінченно далеке в розрахунках освітлення. Потім усі світлові промені від сонця моделюються як паралельні світлові промені, як ми бачимо на наступному зображенні:



Оскільки всі промені світла паралельні, не має значення, як кожен об’єкт пов’язаний із положенням джерела світла, оскільки напрямок світла залишається незмінним для кожного об’єкта в сцені. Оскільки вектор напрямку світла залишається незмінним, обчислення освітлення будуть подібними для кожного об’єкта сцени.

Ми можемо змоделювати таке спрямоване світло, визначивши вектор напрямку світла замість вектора позиції. Розрахунки шейдерів здебільшого залишаються незмінними, за винятком того, що цього разу ми безпосередньо використовуємо вектор `напрямку` світла замість обчислення `lightDir` вектор із використанням вектора `position` світла:

```
struct Light {

    // vec3 position; // no longer necessary when using directional lights.

    vec3 direction;


    vec3 ambient;
```

```
    vec3 diffuse;

    vec3 specular;

};

[...]

void main()

{

  vec3 lightDir = normalize(-light.direction);

  [...]

}
```

Зауважте, що спочатку ми заперечуємо вектор `light.direction`. Розрахунки освітлення, які ми використовували досі, очікують, що напрямок світла буде напрямком від фрагмента до джерела світла, але люди зазвичай вважають за краще вказувати спрямоване світло як глобальний напрямок, що вказує від джерела світла. Тому ми повинні заперечити глобальний вектор напрямку світла, щоб змінити його напрямок; тепер це напрямний вектор, що вказує на джерело світла. Крім того, обов'язково нормалізуйте вектор, оскільки нерозумно вважати, що вхідний вектор є одиничним вектором.

Отриманий вектор `lightDir` потім використовується, як і раніше, у дифузних і дзеркальних обчисленнях.

Щоб чітко продемонструвати, що спрямоване світло однаково впливає на кілька об'єктів, ми повертаємося до сцени контейнерної вечірки з кінця розділу Системи координат. На випадок, якщо ви пропустили вечірку, ми визначили 10 різних позицій контейнерів та згенерували різні матриці моделей для кожного

контейнера, де кожна матриця моделей містила відповідну локальну до світової перетворення:

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

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

Крім того, не забудьте фактично вказати напрямок джерела світла (зверніть увагу, що ми визначаємо напрямок як напрямок від джерела світла; ви можете швидко побачити, що напрямок світла вказує вниз):

```
lightingShader.setVec3("light.direction", -0.2f, -1.0f, -0.3f);
```

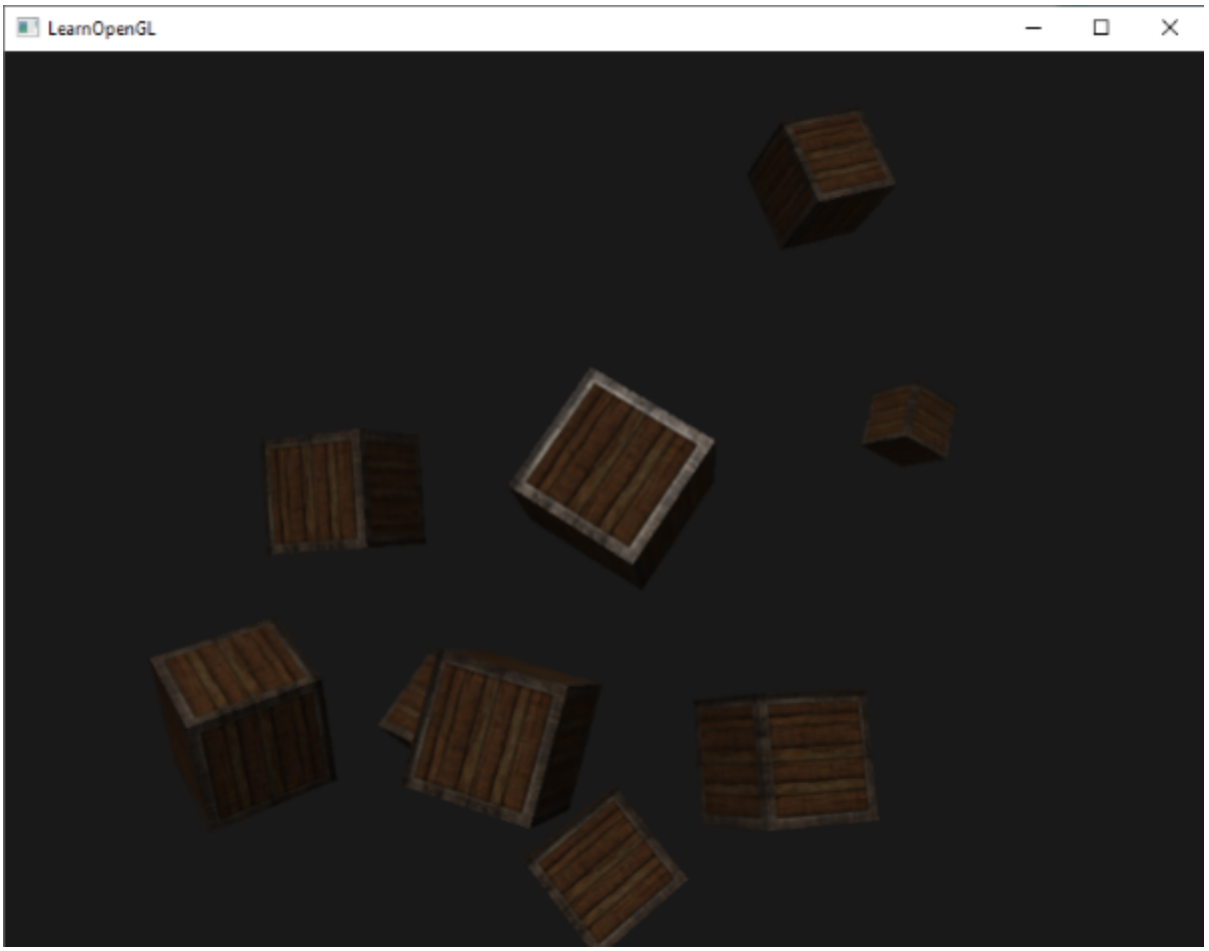
Ми деякий час передаємо вектори позиції та напрямку світла як vec3s, але деякі люди, як правило, вважають за краще зберігати всі вектори, визначені як vec4. Під час визначення векторів позиції як vec4 важливо встановити для компонента w значення 1.0, щоб трансляція та проекції застосовувалися належним чином. Однак, визначаючи вектор напрямку як vec4, ми не хочемо, щоб переклади мали ефект (оскільки вони представляють лише напрямки, нічого більше), тому ми визначаємо < компонент i=7> буде .w0.0

Тоді вектори напрямків можна представити так: vec4(-0.2f, -1.0f, -0.3f, 0.0f). Це також може функціонувати як проста перевірка типів світла: ви можете перевірити, чи дорівнює компонент w 1.0, щоб побачити, що тепер у нас є світло' s вектор позиції, і якщо w дорівнює 0.0, ми маємо вектор напрямку світла; тому скоригуйте обчислення на основі цього:

```
if(lightVector.w == 0.0) // note: be careful for floating point errors
  // do directional light calculations
else if(lightVector.w == 1.0)
  // do light calculations using the light's position (as in previous chapters)
```

Цікавий факт: насправді це те, як старий OpenGL (з фіксованою функціональністю) визначав, чи є джерело світла спрямованим чи позиційним джерелом світла, і регулював його освітлення на основі цього.

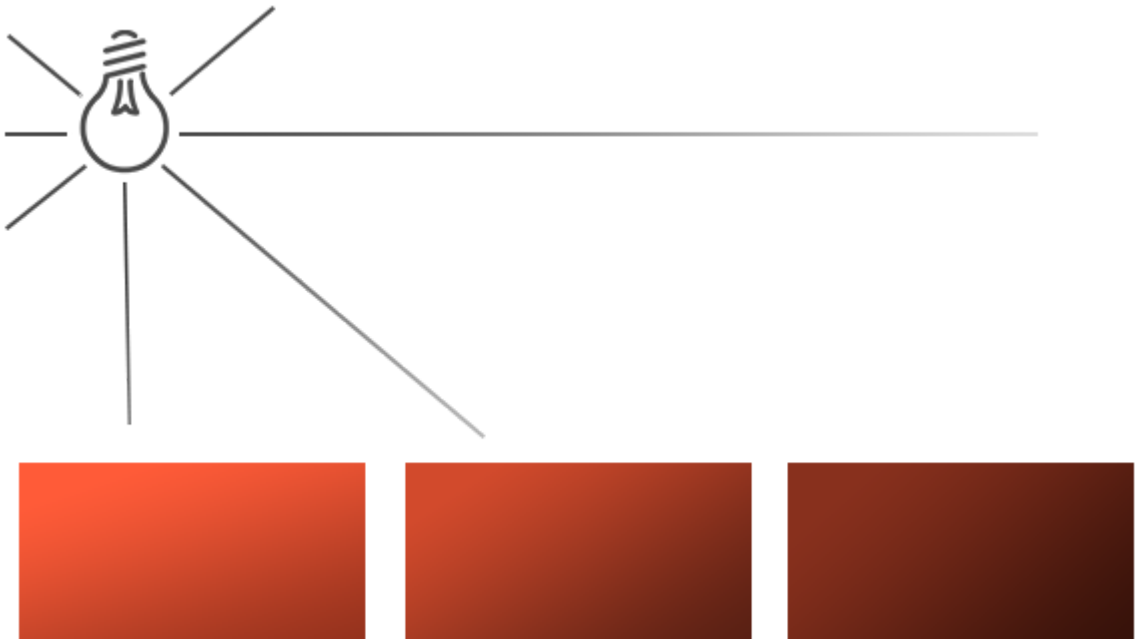
Якщо ви зараз скомпілюєте програму та пролетите через сцену, виявиться, що є джерело світла, схоже на сонце, яке проливає світло на всі об’єкти. Ви бачите, що всі дифузні та дзеркальні компоненти реагують так, ніби десь у небі є джерело світла? Це виглядатиме приблизно так:



Ви можете знайти повний вихідний код програми тут.

Точкові світильники

Спрямоване світло чудово підходить для глобального світла, яке освітлює всю сцену, але зазвичай нам також потрібно кількаточкові світильникирозкидані по сцені. Точкове світло — це джерело світла з заданим положенням десь у світі, яке освітлює в усіх напрямках, де світлові промені тьмяніють на відстані. Подумайте про лампочки та смолоскипи як про коліщатка, які діють як точкове світло.



У попередніх розділах ми працювали зі спрощеним точковим світлом. У нас було джерело світла в заданій позиції, яке розсіює світло в усіх напрямках від цієї заданої позиції світла. Однак джерело світла, яке ми визначили, моделювало світлові промені, які ніколи не зникали, що створювало враження, що джерело світла надзвичайно потужне. У більшості 3D-додатків ми хочемо імітувати джерело світла, яке освітлює лише область поблизу джерела світла, а не всю сцену.

Якщо ви додасте 10 контейнерів до сцени освітлення з попередніх розділів, ви помітите, що весь контейнер позаду освітлюється з тією ж інтенсивністю, що й контейнер перед світлом; поки що немає логіки, яка зменшує світло на відстані. Ми хочемо, щоб контейнер ззаду був лише трохи освітлений порівняно з контейнерами, розташованими поблизу джерела світла.

Затухання

Зазвичай називають зменшення інтенсивності світла на відстані, яку проходить світловий промінь**затухання**. Одним із способів зменшити інтенсивність світла на відстані є просто використання лінійного рівняння. Таке рівняння лінійно зменшувало б інтенсивність світла на відстані, таким чином гарантуючи, що об’єкти на відстані менш яскраві. Однак така лінійна функція має тенденцію

виглядати трохи фальшивою. У реальному світі світло, як правило, досить яскраве, якщо стояти поруч, але яскравість джерела світла швидко зменшується на відстані; решта інтенсивності світла потім повільно зменшується з відстанню. Таким чином, нам потрібне інше рівняння для зменшення інтенсивності світла.

На щастя, деякі розумні люди вже зрозуміли це для нас. Наступна формула обчислює значення ослаблення на основі відстані фрагмента до джерела світла, яке ми пізніше множимо на вектор інтенсивності світла:

(1)

$$(1)Ф\text{□□□}=1.0Кв+Кл*\text{□}+К\text{□}*\text{□}2$$

тут

d

�являє собою відстань від фрагмента до джерела світла. Потім, щоб обчислити значення затухання, ми визначаємо 3 (настроювані) умови: апостійнийтермін

К

в

Кв, алінійнийтермін

К

л

Клі аквадратичнийтермін

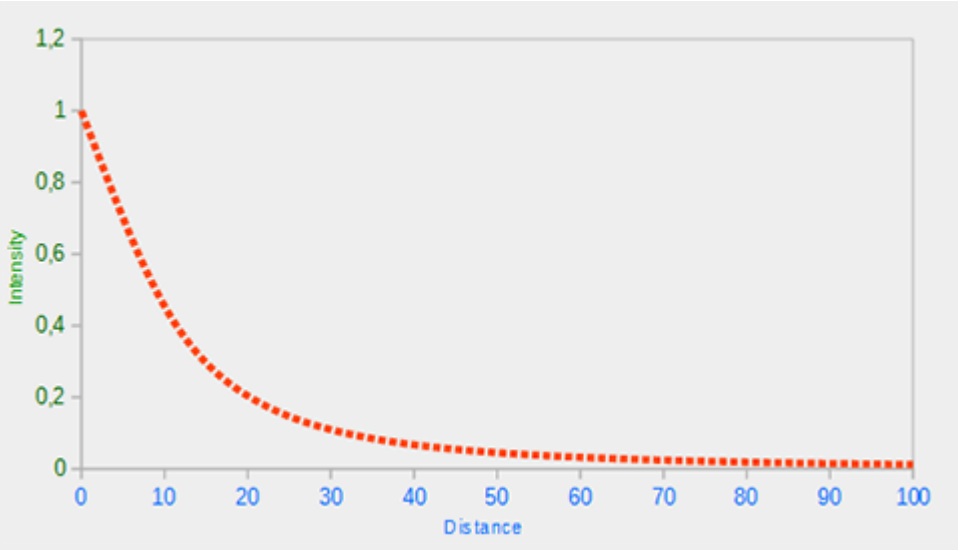
К

q

К�.

- **Постійний член зазвичай зберігається на рівні `1.0`, який в основному існує для того, щоб переконатися, що знаменник ніколи не стає меншим за 1, оскільки в іншому випадку це підвищить інтенсивність з певними відстанями, що не є тим ефектом, якого ми шукаємо.**
- **Лінійний член множиться на значення відстані, яке зменшує інтенсивність лінійним чином.**
- **Квадратичний член множиться на квадрант відстані та встановлює квадратичне зменшення інтенсивності для джерела світла. Квадратичний член буде менш значущим порівняно з лінійним, коли відстань невелика, але стає набагато більшим із збільшенням відстані.**

Завдяки квадратичному члену світло зменшуватиметься переважно лінійним способом, доки відстань не стане достатньо великою, щоб квадратичний член перевищив лінійний, і тоді інтенсивність світла зменшуватиметься набагато швидше. Отриманий ефект полягає в тому, що світло досить інтенсивне на близькій відстані, але швидко втрачає свою яскравість на відстані, поки врешті-решт не втратить свою яскравість у більш повільному темпі. Наступний графік показує ефект такого ослаблення на відстані `100`:



Можна побачити, що світло має найвищу інтенсивність, коли відстань невелика, але як тільки відстань збільшується, його інтенсивність значно зменшується й повільно досягає `0` інтенсивності приблизно на відстані `100`. Це саме те, чого ми хочемо.

Вибір правильних значень

Але при яких значеннях ми встановлюємо ці 3 доданки? Встановлення правильних значень залежить від багатьох факторів: навколишнього середовища, відстані, яку потрібно охопити світлом, типу світла тощо. У більшості випадків це просто питання досвіду та помірної кількості налаштувань. У наведеній нижче таблиці показано деякі значення, які ці терміни можуть прийняти для імітації реалістичного (свого роду) джерела світла, яке охоплює певний радіус (відстань). У першому стовпці вказано відстань, яку подолає світло за заданих термінів. Ці значення є хорошими відправними точками для більшості світильників, люб'язно наданих Ogre3D's wiki:

Відстань	Постійний	Лінійний	Квадратичний

7	1.0	0.7	1.8
13	1.0	0.35	0.44
20	1.0	0.22	0.20
32	1.0	0.14	0.07
50	1.0	0.09	0.032
65	1.0	0.07	0.017
100	1.0	0.045	0.0075
160	1.0	0.027	0.0028
200	1.0	0.022	0.0019
325	1.0	0.014	0.0007
600	1.0	0.007	0.0002

3250	1.0	0.0014	0.000007

Як бачите, постійний термін

К

в

Кв зберігається `1.0` в усіх випадках. Лінійний термін

К

л

Клзазвичай досить малий, щоб покрити більші відстані та квадратичний член

К

q

К навіть менший. Спробуйте трохи поекспериментувати з цими значеннями, щоб побачити їх вплив на вашу реалізацію. У нашому середовищі відстані 32 до `100` зазвичай достатньо для більшості світильників.

Реалізація ослаблення

Щоб застосувати затухання, нам знадобляться 3 додаткові значення у фрагментному шейдері: а саме постійний, лінійний і квадратичний члени рівняння. Найкраще їх зберігати в`світло`Структура , яку ми визначили раніше. Зауважте, що нам потрібно знову обчислити `lightDir` за допомогою `position`, оскільки це точкове світло (як ми робили в попередньому розділі), а не спрямоване світло.

```
struct Light {

    vec3 position;
```

```
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;

    float constant;
    float linear;
    float quadratic;
};
```

Потім ми встановлюємо терміни в нашій програмі: ми хочемо, щоб світло покривало відстань 50, тому ми будемо використовувати відповідні постійні, лінійні та квадратичні терміни з таблиці:

```
lightingShader.setFloat("light.constant",  1.0f);
lightingShader.setFloat("light.linear",    0.09f);
lightingShader.setFloat("light.quadratic", 0.032f);
```

Реалізація затухання у фрагментному шейдері є відносно простою: ми просто обчислюємо значення затухання на основі рівняння та множимо його на навколишній, дифузний і дзеркальний компоненти.

Нам потрібна відстань до джерела світла, щоб рівняння працювало. Пригадайте, як можна обчислити довжину вектора? Ми можемо отримати

відстань, обчисливши вектор різниці між фрагментом і джерелом світла та взявши довжину цього вектора. Ми можемо використовувати вбудований GLSL`довжина`функція для цієї мети:

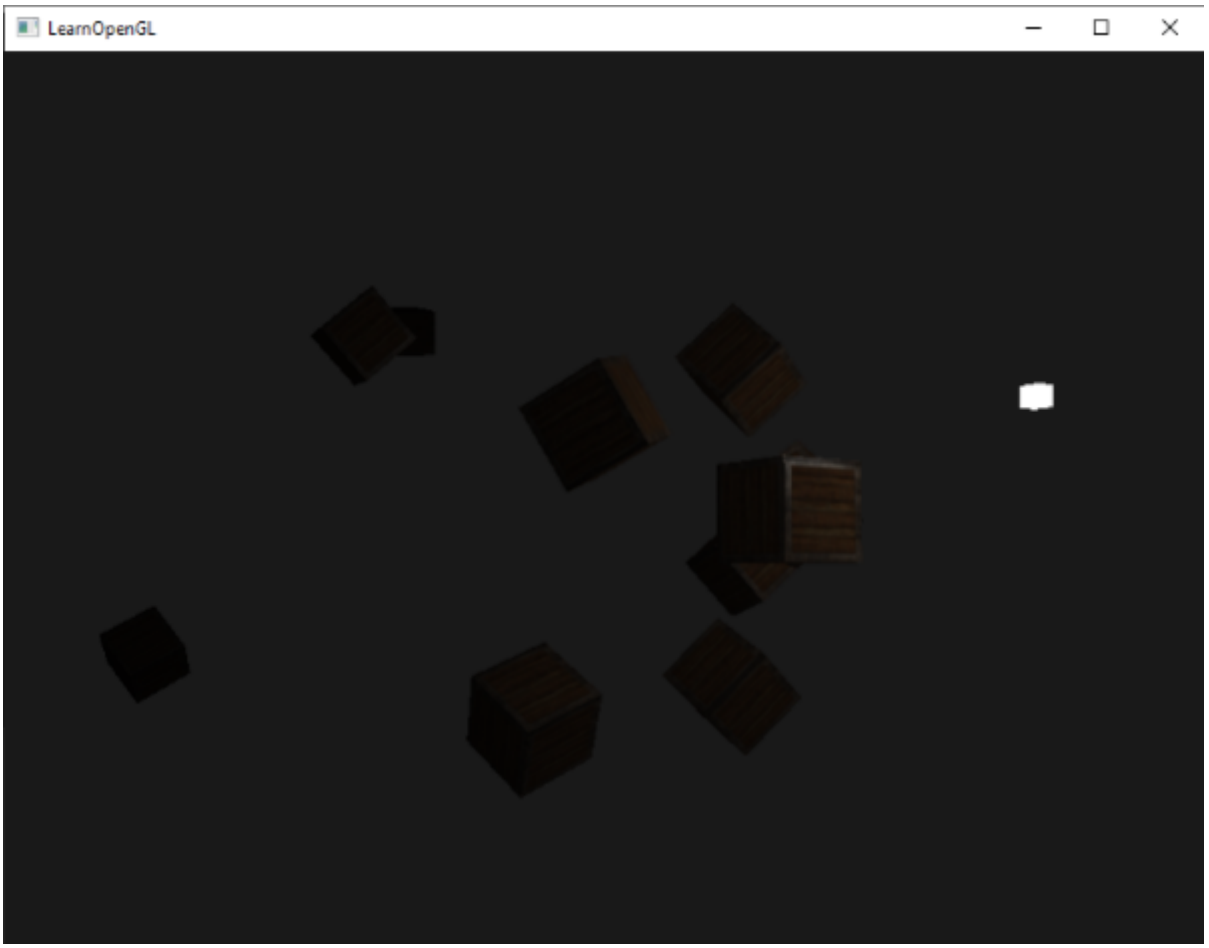
```
float distance    = length(light.position - FragPos);
float attenuation = 1.0 / (light.constant + light.linear * distance +
                           light.quadratic * (distance * distance));
```

Потім ми включаємо це значення ослаблення в розрахунки освітлення шляхом множення значення ослаблення на навколишні, дифузні та дзеркальні кольори.

> Ми могли б залишити компонент навколишнього середовища окремо, щоб навколишнє освітлення не зменшувалося з відстанню, але якщо використовувати більше ніж 1 джерело світла, усі компоненти навколишнього середовища почнуть накопичуватися. У цьому випадку ми хочемо також послабити навколишнє освітлення. Просто пограйте з тим, що найкраще підходить для вашого середовища.

```
ambient  *= attenuation;
diffuse  *= attenuation;
specular *= attenuation;
```

Якщо ви запустите програму, ви отримаєте щось на кшталт цього:



Ви бачите, що зараз світяться лише передні контейнери, а найближчий контейнер світиться найяскравіше. Контейнери в задній частині взагалі не освітлені, оскільки вони надто далеко від джерела світла. Ви можете знайти вихідний код програми тут.

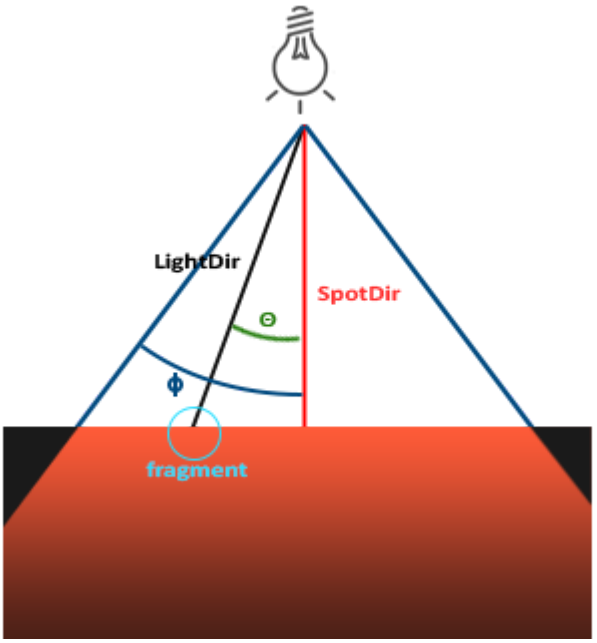
Таким чином, точкове світло — це джерело світла з настроюваним розташуванням і ослабленням, які застосовуються до розрахунків освітлення. Ще один вид світла для нашого світлотехнічного арсеналу.

Прожектор

Останній тип світла, який ми збираємося обговорити, це апрожектор. Прожектор — це джерело світла, розташоване десь у навколишньому середовищі, яке замість того, щоб стріляти світловими променями в усіх напрямках, спрямовує їх лише в певному напрямку. У результаті освітлюються лише об'єкти в певному радіусі від напрямку прожектора, а все інше

залишається темним. Хорошим прикладом прожектора може бути вуличний ліхтар або ліхтарик.

Центр уваги в OpenGL представлений позицією у світовому просторі, напрямком і авідрізаткут, що визначає радіус прожектора. Для кожного фрагмента ми обчислюємо, чи знаходиться фрагмент між напрямками відрізу прожектора (тобто в його конусі), і якщо так, ми освітлюємо фрагмент відповідно. Наступне зображення дає вам уявлення про те, як працює прожектор:



- **`LightDir`: вектор, що вказує від фрагмента до джерела світла.**
- **`SpotDir`: напрямок, куди спрямований прожектор.**
- **`Phi`**
- **ϕ**
- **: кут зрізу, який визначає радіус прожектора. Все поза цим кутом не освітлюється прожектором.**
- **`Theta`**
- **θ**
- **: кут між вектором `LightDir` та `SpotDir` вектор.**
- **θ**
- **значення має бути меншим за**
- **Φ**
- **Фбути в центрі уваги.**

Отже, що нам потрібно зробити, це обчислити скалярний добуток (повертає косинус кута між двома одиничними векторами) між вектором `LightDir` та < вектор /span> і порівняйте його з кутом обрізання `SpotDir`

ф

. Тепер, коли ви (начебто) розумієте, що таке прожектор, ми збираємося створити його у формі ліхтарика.

Ліхтарик

Ліхтарик — це прожектор, розташований у позиції глядача та зазвичай спрямований прямо перед собою з точки зору гравця. По суті, ліхтарик — це звичайний прожектор, але його положення та напрямок постійно оновлюються залежно від позиції та орієнтації гравця.

Отже, значення, які нам знадобляться для фрагментного шейдера, це вектор позиції прожектора (для обчислення вектора напрямку фрагмента до світла), вектор напрямку прожектора і кут зрізу. Ми можемо зберігати ці значення в`світло`структура:

```
struct Light {
    vec3  position;
    vec3  direction;
    float cutOff;
    ...
};
```

Далі ми передаємо відповідні значення шейдеру:

```
lightingShader.setVec3("light.position",  camera.Position);

lightingShader.setVec3("light.direction", camera.Front);

lightingShader.setFloat("light.cutOff",   glm::cos(glm::radians(12.5f)));
```

Як бачите, ми не встановлюємо кут для граничного значення, а обчислюємо значення косинуса на основі кута та передаємо результат косинуса фрагментному шейдеру. Причиною цього є те, що у фрагментному шейдері ми обчислюємо скалярний добуток між векторами `LightDir` та `SpotDir`, а скалярний добуток повертає значення косинуса а не кут; і ми не можемо безпосередньо порівняти кут із значенням косинуса. Щоб отримати кут у шейдері, нам потрібно обчислити арккосинус результату скалярного добутку, що є дорогою операцією. Тому, щоб трохи заощадити продуктивність, ми заздалегідь обчислюємо значення косинуса даного кута зрізу та передаємо цей результат фрагментному шейдеру. Оскільки обидва кути тепер представлені як косинуси, ми можемо безпосередньо порівнювати між ними без дорогих операцій.

Тепер, що залишилося зробити, це обчислити тета

θ

�і порівняйте його з межею

ϕ

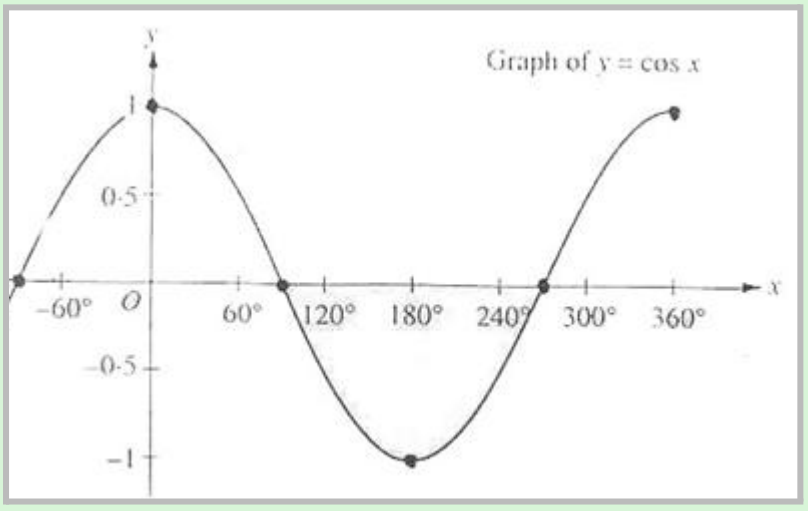
�значення, щоб визначити, чи перебуваємо ми в центрі уваги чи поза ним:

```
float theta = dot(lightDir, normalize(-light.direction));
```

```
if(theta > light.cutOff)

{

  // do lighting calculations

}

else  // else, use ambient light so scene isn't completely dark outside the spotlight.

  color = vec4(light.ambient * vec3(texture(material.diffuse, TexCoords)), 1.0);
```

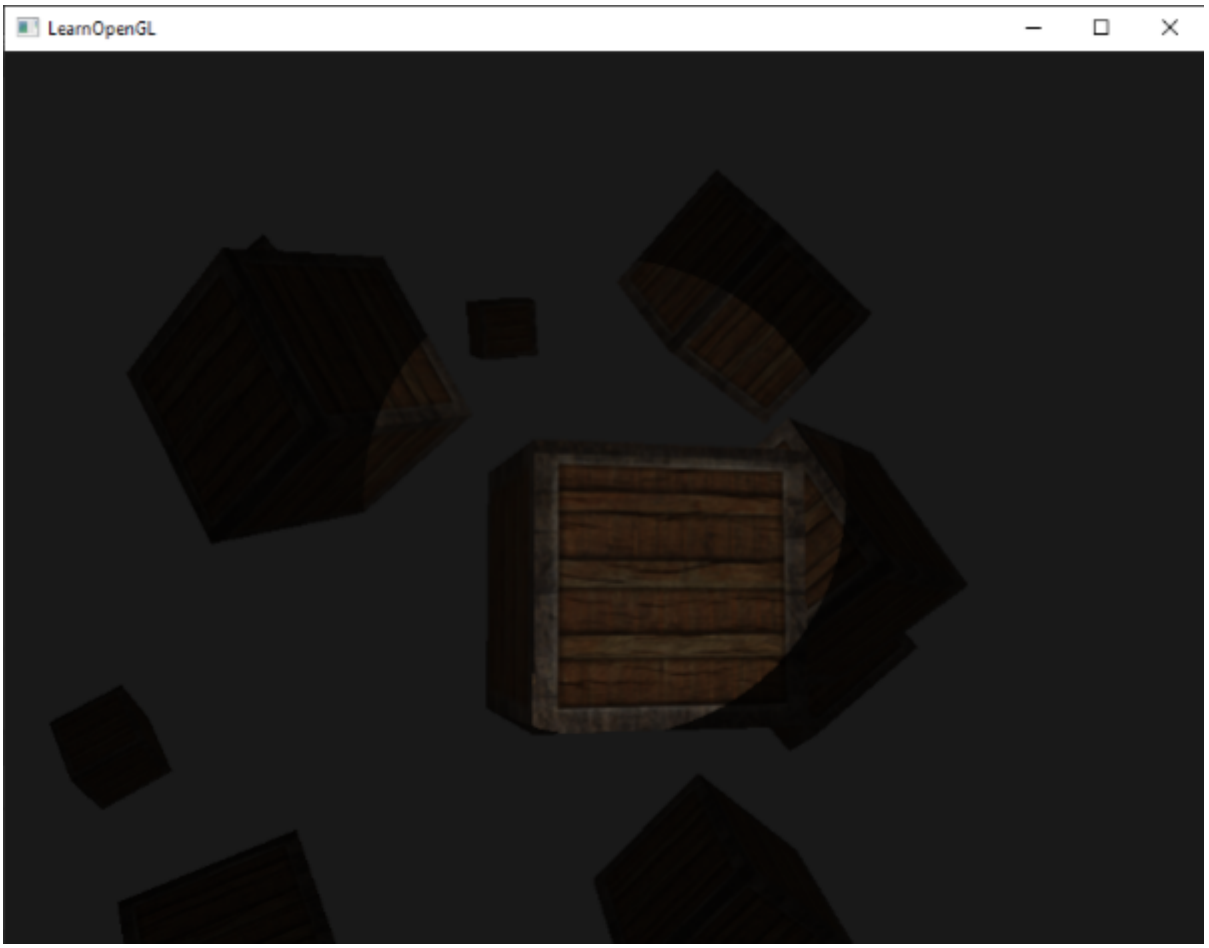
Спочатку ми обчислюємо скалярний добуток між вектором `lightDir` та об’єднаним `напрямком` вектор (заперечено, тому що ми хочемо, щоб вектори вказували на джерело світла, а не від). Обов’язково нормалізуйте всі відповідні вектори.

> **Вам може бути цікаво, чому на щитку стоїть знак > замість знака <. Чи не має `тета` бути меншим за граничне значення світла, щоб бути всередині прожектора? Це правильно, але не забувайте, що значення кутів представлено як значення косинуса, а кут градусів представлено як значення косинуса , тоді як кут градусів представлено як значення косинуса , як ви можете бачити тут: `if01.0900.0`**



Тепер ви бачите, що чим ближче значення косинуса до `1.0`, тим менший його кут. Тепер зрозуміло, чому `theta` має бути більшим за граничне значення. Граничне значення наразі встановлено на косинусі `12.5`, який дорівнює `0.976`, тому косинус `тета0.976< значення і=7>` між і `1.0` призведе до того, що фрагмент буде освітлено так, ніби всередині прожектора.

Запуск програми призводить до прожектора, який освітлює лише ті фрагменти, які знаходяться безпосередньо всередині конуса прожектора. Це виглядатиме приблизно так:



Ви можете знайти повний вихідний код тут.

Хоча він все ще виглядає трохи фальшивим, головним чином тому, що прожектор має жорсткі краї. Усюди, де фрагмент досягає краю конуса прожектора, він повністю вимикається, а не з гарним плавним згасанням. Реалістичний прожектор поступово зменшував би світло навколо своїх країв.

Гладкі/м'які краї

Щоб створити ефект прожектора з гладкими краями, ми хочемо імітувати прожектор, який маєвнутрішнійі анзовнішнійконус. Ми можемо встановити внутрішній конус як конус, визначений у попередньому розділі, але нам також потрібен зовнішній конус, який поступово приглушує світло від внутрішнього до країв зовнішнього конуса.

Щоб створити зовнішній конус, ми просто визначаємо інше значення косинуса, яке представляє кут між вектором напрямку прожектора та вектором

зовнішнього конуса (дорівнює його радіусу). Тоді, якщо фрагмент знаходиться між внутрішнім і зовнішнім конусом, він повинен обчислити значення інтенсивності між `0.0` і `1.0`. Якщо фрагмент знаходиться всередині внутрішнього конуса, його інтенсивність дорівнює `1.0` і `0.0`, якщо фрагмент знаходиться поза зовнішнім конусом.

Ми можемо розрахувати таке значення за допомогою наступного рівняння:

$$\text{я} = \frac{\theta - \gamma}{\epsilon}$$

тутϵ(епсилон) - це різниця косинусів між внутрішніми (ϕ) і зовнішній конус (γ) ($\epsilon = \phi - \gamma$). Отриманийя тоді значення дорівнює інтенсивності прожектора в поточному фрагменті.

Трохи важко уявити, як ця формула насправді працює, тому давайте спробуємо її з кількома зразками значень:

Трохи важко уявити, як ця формула насправді працює, тому давайте спробуємо її з кількома зразками значень:

θ	θв градусах	ϕ(внутрішнє відсічення)	ϕв градусах	γ(зовнішнє відсічення)	γв градусах	ϵ	я
0.87	30	0.91	25	0.82	35	0.91 - 0.82 = 0.09	0.87 - 0.82 / 0.09 = 0.56
0.9	26	0.91	25	0.82	35	0.91 - 0.82 = 0.09	0.9 - 0.82 / 0.09 = 0.89
0.97	14	0.91	25	0.82	35	0.91 - 0.82 = 0.09	0.97 - 0.82 / 0.09 = 1.67
0.83	34	0.91	25	0.82	35	0.91 - 0.82 = 0.09	0.83 - 0.82 / 0.09 = 0.11
0.64	50	0.91	25	0.82	35	0.91 - 0.82 = 0.09	0.64 - 0.82 / 0.09 = -2.0
0.966	15	0.9978	12.5	0.953	17.5	0.9978 - 0.953 = 0.0448	0.966 - 0.953 / 0.0448 = 0.29

Як ви бачите, ми в основному інтерполюємо зовнішній косинус і внутрішній косинус на основі

θ

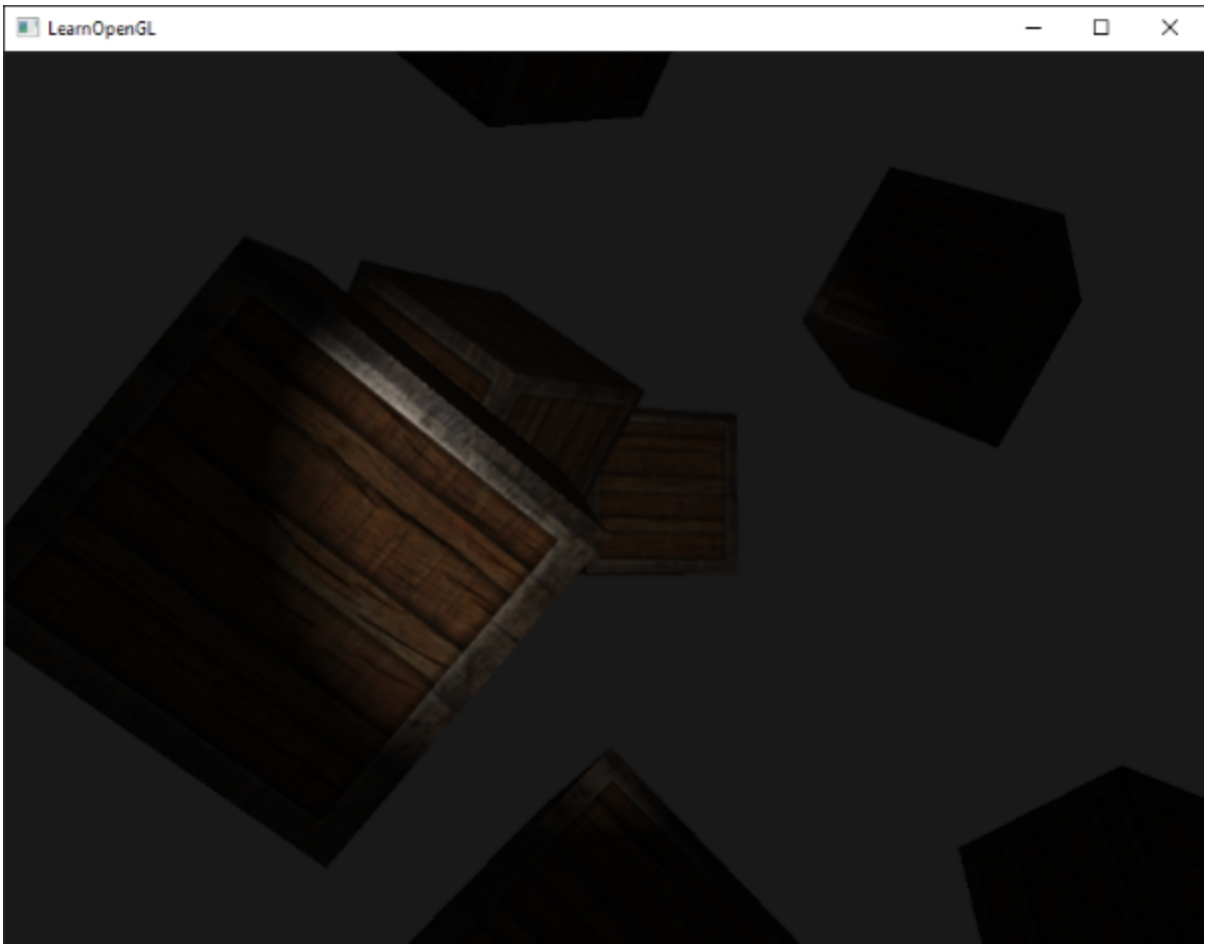
значення. Якщо ви все ще не розумієте, що насправді відбувається, не хвилюйтеся, ви можете просто прийняти формулу як належне та повернутися сюди, коли станете набагато старшими та мудрішими.

Тепер ми маємо значення інтенсивності, яке є або від'ємним, коли поза прожектором, або вищим за `1.0`, коли всередині внутрішнього конуса, і десь посередині навколо країв. Якщо ми правильно обмежимо значення, нам більше не потрібен `if-else` у фрагментному шейдері, і ми можемо просто помножити світлові компоненти на обчислене значення інтенсивності:

```
float theta     = dot(lightDir, normalize(-light.direction));

float epsilon   = light.cutOff - light.outerCutOff;

float intensity = clamp((theta - light.outerCutOff) / epsilon, 0.0, 1.0);

...

// we'll leave ambient unaffected so we always have a little light.

diffuse  *= intensity;

specular *= intensity;

...
```

Зверніть увагу, що ми використовуємо`затискач`функція що`затискачі` його перший аргумент між значеннями `0.0` та `1.0`. Це гарантує, що значення інтенсивності не вийдуть за межі діапазону [0, 1].

Переконайтеся, що ви додали значення `outerCutOff` до `світло` struct і встановіть його однорідне значення в програмі. Для наступного зображення використано внутрішній кут обрізання `12.5` та зовнішній кут обрізання `17.5`:



Аааа, це набагато краще. Пограйте з внутрішнім і зовнішнім кутами зрізу та спробуйте створити прожектор, який краще відповідає вашим потребам. Ви можете знайти вихідний код програми тут.

Такий тип лампи з ліхтариком/прожектором ідеально підходить для ігор жахів, і в поєднанні з спрямованим і точковим світлом навколишнє середовище дійсно почне освітлюватися.

вправи

- **Спробуйте поекспериментувати з усіма різними типами світла та їхніми фрагментними шейдерами. Спробуйте інвертувати деякі вектори та/або використати < замість >. Спробуйте пояснити різні візуальні результати.**

Кілька вогнів

У попередніх розділах ми дізналися багато нового про освітлення в OpenGL. Ми дізналися про затінення Phong, матеріали, карти освітлення та різні типи світильників. У цьому розділі ми збираємося об'єднати всі отримані раніше знання, створивши повністю освітлену сцену з 6 активними джерелами світла. Ми збираємося імітувати світло, схоже на сонце, як спрямоване джерело світла, 4 точкові світла, розсіяні по всій сцені, і ми також додамо ліхтарик.

Щоб використовувати більше одного джерела світла в сцені, ми хочемо інкапсулювати розрахунки освітлення в GLSLфункції. Причина цього полягає в тому, що код швидко стає неприємним, коли ми виконуємо обчислення освітлення з кількома типами світла, кожен з яких потребує різних обчислень. Якби ми зробили всі ці розрахунки восновнийлише функція, код швидко стає важко зрозуміти.

Функції в GLSL схожі на C-функції. У нас є назва функції, тип повернення, і нам потрібно оголосити прототип у верхній частині файлу коду, якщо функція ще не була оголошена перед основною функцією. Ми створимо різні функції для кожного типу світла: спрямоване світло, точкове світло та прожектори.

При використанні кількох джерел світла в сцені підхід зазвичай такий: ми маємо єдиний колірний вектор, який представляє вихідний колір фрагмента. Для кожного світла внесок світла у фрагмент додається до цього вихідного вектора кольорів. Таким чином, кожне джерело світла в сцені розраховуватиме свій індивідуальний вплив і вноситиме його в кінцевий вихідний колір. Загальна структура виглядатиме приблизно так:

```
out vec4 FragColor;


void main()
{
  // define an output color value
  vec3 output = vec3(0.0);
  // add the directional light's contribution to the output
  output += someFunctionToCalculateDirectionalLight();
  // do the same for all point lights
  for(int i = 0; i < nr_of_point_lights; i++)
          output += someFunctionToCalculatePointLight();
  // and add others lights as well (like spotlights)
  output += someFunctionToCalculateSpotLight();


  FragColor = vec4(output, 1.0);
}
```

Фактичний код, ймовірно, відрізнятиметься залежно від реалізації, але загальна структура залишається незмінною. Ми визначаємо кілька функцій, які обчислюють вплив на джерело світла та додають його результуючий колір до вихідного кольорового вектора. Якщо, наприклад, два джерела світла

знаходяться близько до фрагмента, їхній спільний внесок призведе до більш яскравого освітлення фрагмента порівняно з фрагментом, освітленим одним джерелом світла.

Направлене світло

Ми хочемо визначити функцію у фрагментному шейдері, яка обчислює вплив спрямованого світла на відповідний фрагмент: функцію, яка приймає кілька параметрів і повертає розрахований спрямований колір освітлення.

Спочатку нам потрібно встановити необхідні змінні, які нам мінімально потрібні для спрямованого джерела світла. Ми можемо зберігати змінні в структурі під назвою`DirLight` і визначте його як однорідний. Змінні структури повинні бути знайомі з [попереднього](#) розділу:

```
struct DirLight {
    vec3 direction;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
uniform DirLight dirLight;
```

Потім ми можемо передати форму `dirLight` функції з таким прототипом:

```
vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir);
```

Подібно до C і C++, коли ми хочемо викликати функцію (у цьому випадку всередині`основний`функція) функція має бути визначена десь перед номером рядка абонента. У цьому випадку ми б віддали перевагу визначати функції нижче`основний`тому ця вимога не виконується. Тому ми оголошуємо прототипи функції десь вище`основний`функція, як і в C.

Ви бачите, що функція вимагає а`DirLight`struct і два інші вектори, необхідні для його обчислення. Якщо ви успішно пройшли попередній розділ, то вміст цієї функції не повинен викликати подиву:

```
vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir)
{
    vec3 lightDir = normalize(-light.direction);
    // diffuse shading
    float diff = max(dot(normal, lightDir), 0.0);
    // specular shading
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    // combine results
    vec3 ambient  = light.ambient  * vec3(texture(material.diffuse, TexCoords));
```

```
    vec3 diffuse  = light.diffuse  * diff * vec3(texture(material.diffuse, TexCoords));
    vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
    return (ambient + diffuse + specular);
}
```

Ми фактично скопіювали код із попереднього розділу та використали вектори, подані як аргументи функції, щоб обчислити вектор вкладу спрямованого світла. Отримані навколишні, дифузні та дзеркальні внески потім повертаються як один колірний вектор.

Точкове світло

Подібно до спрямованого світла, ми також хочемо визначити функцію, яка обчислює внесок точкового світла у даний фрагмент, включаючи його затухання. Подібно до спрямованого світла, ми хочемо визначити структуру, яка вказує всі змінні, необхідні для точкового світла:

```
struct PointLight {
    vec3 position;

    float constant;
    float linear;
    float quadratic;
```

```
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
#define NR_POINT_LIGHTS 4
uniform PointLight pointLights[NR_POINT_LIGHTS];
```

Як бачите, ми використали директиву препроцесора в GLSL, щоб визначити кількість точкових джерел світла, які ми хочемо мати в нашій сцені. Потім ми використовуємо цю константу `NR_POINT_LIGHTS` для створення масиву `PointLight`структури. Масиви в GLSL схожі на масиви C і можуть бути створені за допомогою двох квадратних дужок. Зараз у нас 4`PointLight`структури для заповнення даними.

Прототип функції точкового світла такий:

```
vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir);
```

Функція приймає всі необхідні дані як свої аргументи та повертає vec3, який представляє внесок кольору, який це конкретне точкове світло має на фрагменті. Знову ж таки, інтелектуальне копіювання та вставлення призводить до такої функції:

```
vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
```

```
{
    vec3 lightDir = normalize(light.position - fragPos);
    // diffuse shading
    float diff = max(dot(normal, lightDir), 0.0);
    // specular shading
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    // attenuation
    float distance    = length(light.position - fragPos);
    float attenuation = 1.0 / (light.constant + light.linear * distance +
                               light.quadratic * (distance * distance));
    // combine results
    vec3 ambient  = light.ambient  * vec3(texture(material.diffuse, TexCoords));
    vec3 diffuse  = light.diffuse  * diff * vec3(texture(material.diffuse, TexCoords));
    vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
    ambient  *= attenuation;
    diffuse  *= attenuation;
    specular *= attenuation;
    return (ambient + diffuse + specular);
}
```

Абстрагування цієї функції в подібній функції має ту перевагу, що ми можемо легко обчислити освітлення для кількох точкових світильників без необхідності дублювати код. В`основний`ми просто створюємо цикл, який повторює масив точкового світла, який викликає`CalcPointLight`для кожного точкового світла.

Зібравши все разом

Тепер, коли ми визначили функцію для спрямованого світла та функцію для точкового світла, ми можемо об'єднати їх у`основний`функція.

```
void main()

{

    // properties

    vec3 norm = normalize(Normal);

    vec3 viewDir = normalize(viewPos - FragPos);



    // phase 1: Directional lighting

    vec3 result = CalcDirLight(dirLight, norm, viewDir);

    // phase 2: Point lights

    for(int i = 0; i < NR_POINT_LIGHTS; i++)

        result += CalcPointLight(pointLights[i], norm, FragPos, viewDir);
```

```
    // phase 3: Spot light

    //result += CalcSpotLight(spotLight, norm, FragPos, viewDir);


    FragColor = vec4(result, 1.0);

}
```

Кожен тип світла додає свій внесок у вихідний колір, доки не буде оброблено всі джерела світла. Отриманий колір містить колірний вплив усіх джерел світла в сцені разом. Ми залишаємо`CalcSpotLight`функціонувати як вправа для читача.

> У цьому підході є багато повторюваних обчислень, розподілених за функціями типу світла (наприклад, обчислення вектора відображення, дифузних і дзеркальних елементів, а також вибірка текстур матеріалу), тому тут є місце для оптимізації.

Встановлення уніформ для структури спрямованого світла не повинно бути надто незнайомим, але вам може бути цікаво, як встановити уніфіковані значення точкових вогнів, оскільки уніформа точкового світла насправді є масивом`PointLight`структури. Це не те, про що ми раніше говорили.

На щастя для нас, це не надто складно. Встановлення уніфікованих значень масиву структур працює так само, як встановлення уніфікованих значень окремої структури, хоча цього разу ми також маємо визначити відповідний індекс під час запиту розташування уніформи:

```
lightingShader.setFloat("pointLights[0].constant", 1.0f);
```

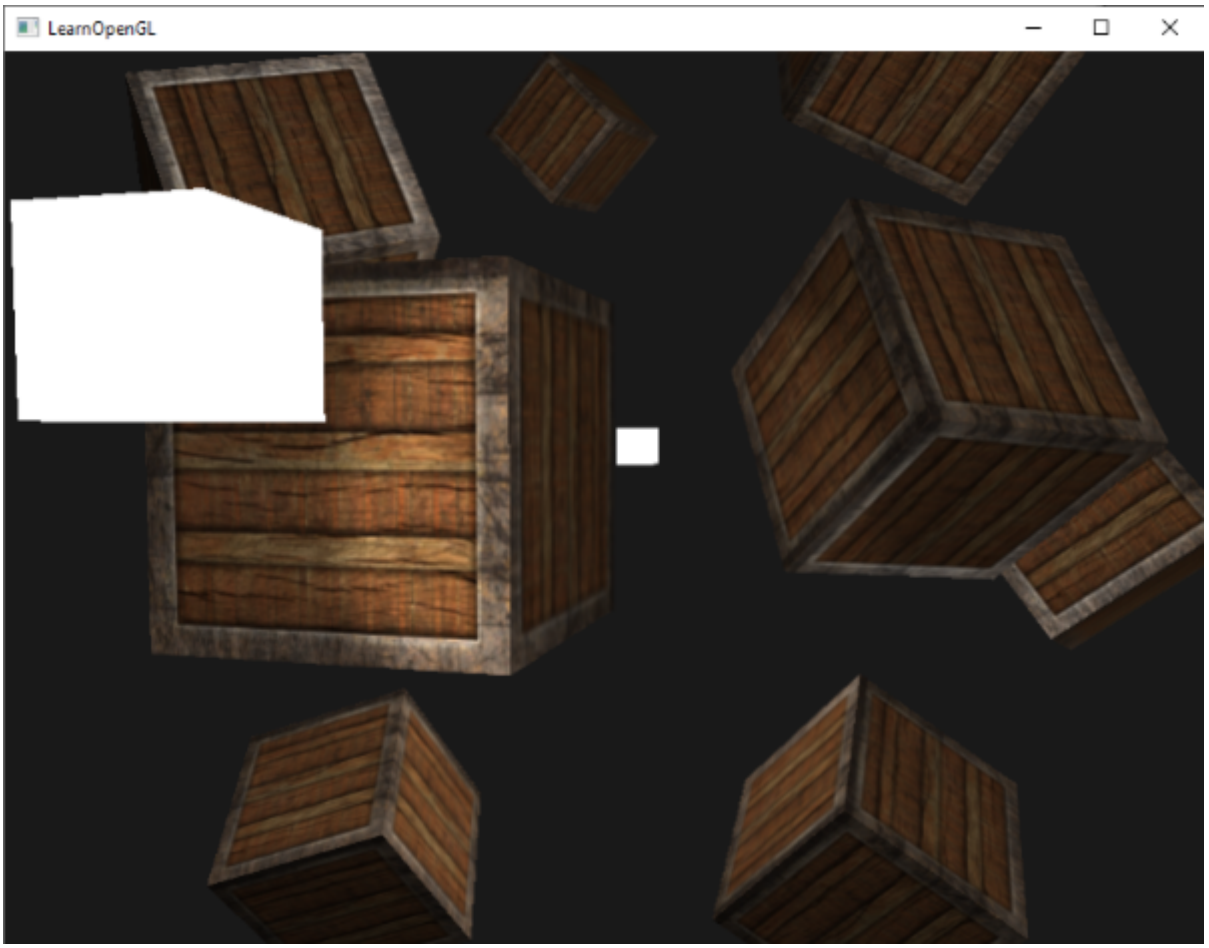
Тут ми індексуємо перший`PointLight` struct у масиві `pointLights` та внутрішньо отримувати розташування його `константи`1.0< змінна i=4>, якій ми встановили .

Не забуваймо, що нам також потрібно визначити вектор позиції для кожного з 4-х точкових джерел світла, тому давайте трохи розподілимо їх по сцені. Ми визначимо інший масив `glm::vec3`, який містить точкові світильники' позиції:

```
glm::vec3 pointLightPositions[] = {
        glm::vec3( 0.7f,  0.2f,  2.0f),
        glm::vec3( 2.3f, -3.3f, -4.0f),
        glm::vec3(-4.0f,  2.0f, -12.0f),
        glm::vec3( 0.0f,  0.0f, -3.0f)
};
```

Потім індексуємо відповідні`PointLight` struct з масиву `pointLights` та встановіть його `позицію` атрибут як одну з позицій, які ми щойно визначили. Також не забудьте намалювати 4 світлові куби замість 1. Просто створіть іншу матрицю моделі для кожного світлового об’єкта, як ми робили з контейнерами.

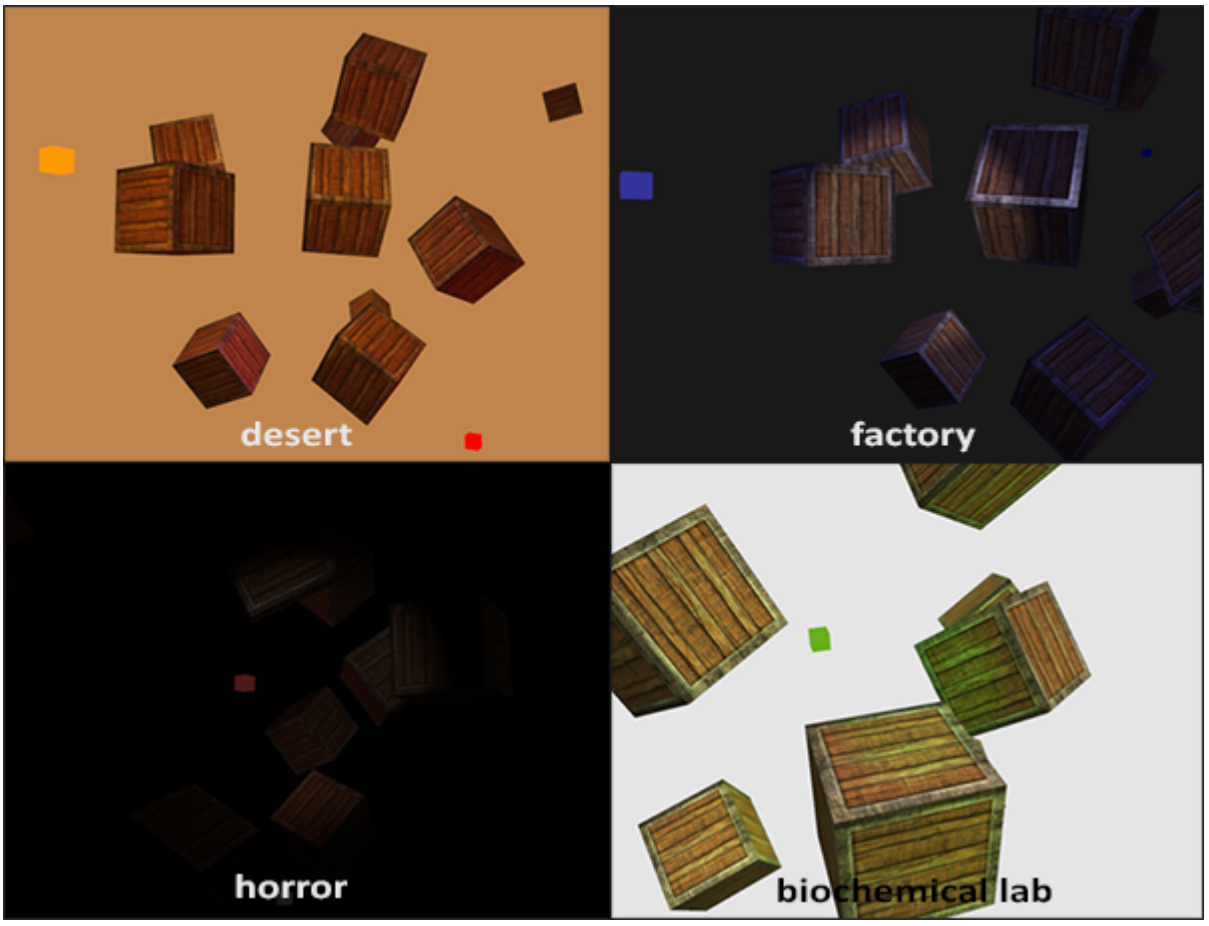
Якщо ви також використовуєте ліхтарик, результат усіх об’єднаних вогнів виглядатиме приблизно так:



Як ви бачите, здається, що десь на небі є якась форма глобального світла (наприклад, сонце), у нас є 4 вогні, розкидані по всій сцені, і ліхтарик видно з точки зору гравця. Виглядає досить акуратно, чи не так?

Ви можете знайти повний вихідний код кінцевої програми тут.

На зображенні показано всі джерела світла, встановлені за умовчанням, які ми використовували в попередніх розділах, але якщо ви пограєте з цими значеннями, ви можете отримати досить цікаві результати. Художники та дизайнери рівнів зазвичай налаштовують усі ці змінні освітлення у великому редакторі, щоб переконатися, що освітлення відповідає навколишньому середовищу. Використовуючи наше просте середовище, ви вже можете створювати досить цікаві візуальні ефекти, просто налаштовуючи світло' атрибути:



Ми також змінили чіткий колір, щоб краще відображати освітлення. Ви бачите, що просто регулюючи деякі параметри освітлення, ви можете створити зовсім іншу атмосферу.

Тепер ви маєте досить добре розуміти освітлення в OpenGL. З наявними знаннями ми вже можемо створювати цікаві та візуально багаті середовища й атмосфери. Спробуйте погратися з різними цінностями, щоб створити власну атмосферу.

вправи

- **Чи можете ви (начебто) відтворити різні атмосфери останнього зображення, налаштувавши значення атрибутів світла? рішення.**

ОГЛЯД

Вітаємо з успіхом! Я не впевнений, чи ви помітили, але протягом усіх розділів про освітлення ми не дізналися нічого нового про сам OpenGL, окрім кількох незначних елементів, як-от доступ до однорідних масивів. Усі розділи про освітлення досі стосувалися маніпулювання шейдерами за допомогою технік і рівнянь для досягнення реалістичних результатів освітлення. Це ще раз демонструє вам силу шейдерів. Шейдери надзвичайно гнучкі, і ви на власні очі переконалися, що за допомогою лише кількох 3D-векторів і деяких настроюваних змінних ми змогли створити дивовижну графіку!

Кілька останніх розділів ви дізналися про кольори, модель освітлення Фонга (яка включає навколишнє, дифузне та дзеркальне освітлення), матеріали об'єктів, властивості світла, які можна налаштувати, дифузні та дзеркальні карти, різні типи світла та те, як об'єднати всі знання в одна повністю освітлена сцена. Обов'язково поекспериментуйте з різними джерелами світла, кольорами матеріалів, властивостями світла та спробуйте створити власне середовище за допомогою трохи творчості.

У наступних розділах ми додамо до нашої сцени більш просунуті геометричні фігури, які дуже добре виглядатимуть у моделях освітлення, які ми обговорювали.

Глосарій

- **`Color vector`: вектор, що відображає більшість кольорів реального світу за допомогою комбінації червоного, зеленого та синього компонентів (скорочено RGB). Колір об'єкта — це відбиті компоненти кольору, які об'єкт не поглинув.**
- **`Phong lighting model`: модель для наближення освітлення реального світу шляхом обчислення навколишнього, дифузного та дзеркального компонентів.**

- **`Ambient lighting`: наближення глобального освітлення шляхом надання кожному об’єкту невеликої яскравості, щоб об’єкти не були повністю темними, якщо вони не освітлені прямо.**
- **`Diffuse shading`: освітлення, яке стає сильнішим, чим більше вершина/фрагмент вирівняно до джерела світла. Використовує нормальні вектори для обчислення кутів.**
- **`Normal vector`: одиничний вектор, перпендикулярний до поверхні.**
- **`Normal matrix`: матриця 3x3, яка є матрицею моделі (або модель-вид) без перекладу. Він також модифікований таким чином (зворотне транспонування), що зберігає вектори нормалей у правильному напрямку під час застосування нерівномірного масштабування. Інакше нормальні вектори спотворюються при використанні нерівномірного масштабування.**
- **`Specular lighting`: встановлює віддзеркалення, що ближче глядач дивиться на відображення джерела світла на поверхні. На основі напрямку глядача, напрямку світла та значення блиску, яке встановлює ступінь розсіювання відблиску.**
- **`Phong shading`: модель освітлення Phong, застосована у фрагментному шейдері.**
- **`Gouraud shading`: модель освітлення Phong, застосована у вершинному шейдері. Створює помітні артефакти при використанні невеликої кількості вершин. Підвищує ефективність при втраті якості зору.**
- **`GLSL struct`: C-подібна структура, яка діє як контейнер для змінних шейдера. Здебільшого використовується для організації входу, виходу та уніформи.**
- **`Material`: навколишній, дифузний і дзеркальний колір, який відображає об’єкт. Вони встановлюють кольори об’єкта.**
- **`Light (properties)`: навколишня, дифузна та дзеркальна інтенсивність світла. Вони можуть приймати будь-які значення кольору та визначати,**

яким кольором/інтенсивністю світить джерело світла для кожного конкретного компонента Phong.

- **`Diffuse map`: зображення текстури, яке встановлює дифузний колір для кожного фрагмента.**
- **`Specular map`: карта текстури, яка встановлює інтенсивність/колір дзеркального відображення для кожного фрагмента. Дозволяє створювати відблиски лише на певних ділянках об'єкта.**
- **`Directional light`: джерело світла, яке має лише напрямок. Він змодельований так, що він знаходиться на нескінченній відстані, що призводить до того, що всі його світлові промені здаються паралельними, а вектор його напрямку, таким чином, залишається незмінним на всій сцені.**
- **`Point light`: джерело світла в сцені зі світлом, яке зникає на відстані.**
- **`Attenuation`: процес зменшення інтенсивності світла на відстані, використовується в точкових і прожекторах.**
- **`Spotlight`: джерело світла, визначене конусом в одному конкретному напрямку.**
- **`Flashlight`: прожектор, розташований з точки зору глядача.**
- **`GLSL uniform array`: масив рівномірних значень. Працюють так само, як C-масив, за винятком того, що їх не можна динамічно розподіляти.**