Table of Contents

Вступ в рейтрейсінг: простий спосіб створення 3D зображень

Дана стаття перекладена з англійської мови. Оригінальна стаття знаходиться за цим посиланням.

Keywords: рейтрейсинг (ray-tracing), перспективна проекція (perspective projection), conductor, діалектрик (dielectric), пряме трасування (forward tracing), зворотнє трасування (backward tracing), тіньовий промінь (shadow ray), головний промінь (primary ray), горовий промінь (eye ray), path tracing, віддзеркалення (reflection), заломлення (refraction), Appel, Whitted, коефіцієнт заломлення (index of refraction), рівняння Френеля (Fresnel equation), transmission, рекурсивний (recursive), глибина рекурсії (recursion depth), image file format.

В цьому розділі будуть розглядатись наступні питання:

1. **Як це працює?**
2. **В двох словах про рейтрейсінг алгоритм**
3. **Застовування рейтрейсінг алгоритму**
4. **Додаємо віддзеркалення і заломлення (Reflection and Refraction)**
5. **Пишемо простенький рейтрейсер**
6. **Source Code**

Як це працює?

> Ти також можеш перевірити урок An Overview of the Ray-Tracing Rendering Technique якщо ти зацікавлений у вивченні особливостей рейтрейсингу. Цей урок є більш загальним введенням до 3D рендерингу.

Для того, щоб почати цей урок, ми пояснимо, як тривимірна сцена перетворюється в двовимірне зображення. Після того, як ми зрозуміємо цей процес і те, що він включає в себе, ми будемо мати можливість використовувати комп'ютер для імітації «штучних» зображень аналогічними методами. Нам подобається думати про цей розділ як теорію, на якій будується більш просунута CG (комп'ютерна графіка). (We like to think of this section as the theory that more advanced CG is built upon.)

У другій частині цього уроку ми покажемо алгоритм трасування променів і пояснимо в двох словах, як він працює. Ми отримали листи від різних людей, які запитують, чому ми зосереджені на рейтрейсінгу, а не на інших алгоритмах. Чому ми вирішили зосередитися на трасуванні променів в цьому вступному уроці? Просто тому, що цей алгоритм є найбільш простим способом моделювання фізичних явищ, які змушують об'єкти бути видимими. З цієї причини ми вважаємо, трасування променів є кращим вибором, серед інших методів, при написанні програми, яка створює прості зображення.

Для початку, ми закладемо фундамент з алгоритмом трасування променів. Однак, як тільки ми осягнемо всю інформацію, яку ми повинні реалізувати для візуалізації, ми покажемо на прикладі як це зробити добре.

Як створити зображення?

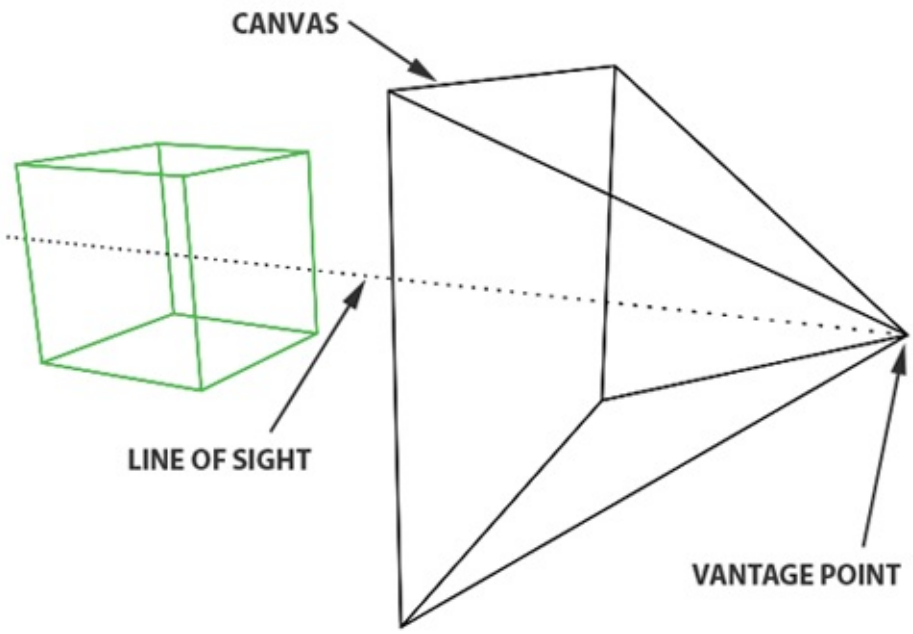


*Рисунок 1: ми можемо уявити собі таку картину, як надріз зроблений через піраміду, вершина якого знаходиться в центрі нашого ока і висота якого паралельна (**line of sight**) лінії зору.*

Здається незвичним починати з такого твердження, що перша річ, яка потрібна для отримання зображення, являє собою двовимірну поверхню (ця поверхня повинна бути на площині і не може бути точкою). Маючи це на увазі, ми можемо уявити собі таку картину, як надріз зроблений через піраміду, вершина якого знаходиться в центрі нашого ока і висота якого паралельна (**line of sight**) лінії зору (пам'ятаєте, для того, щоб побачити щось, ми повинні дивитися вздовж лінії, яка з'єднується з об'єктом). Ми будемо називати цей надріз, згаданий вище, площиною зображення (ви можете побачити цю (**image plane**) площину зображення в якості полотна, яке використовується художниками). Площина зображення є концепція комп'ютерної графіки, і ми будемо використовувати її в якості двомірної поверхні, на яку проектується наша тривимірна сцена. Те що ми тільки що описали може здатися очевидним, але це є одним з основних понять, які використовуються для створення зображень на безлічі різних апаратів. Наприклад, еквівалент в фотографії є поверхня плівки (або вже як згадувалося раніше, полотно, яке використовують художники).

Перспективна проекція (Perspective Projection)

Давайте уявимо, що ми хочемо намалювати куб на чистому полотні. Найпростіший спосіб опису процесу проектування - це почати малювати лінії від кожного кута тривимірного куба до ока. Для того, щоб намітити форму об'єкта на полотні, ми відзначаємо точку, де кожна лінія перетинається з поверхнею площини зображення. Наприклад, припустимо, що **c0** кут куба, і що він пов'язаний з трьома іншими точками:

c1, **c2** і **c3**. Після проектування цих чотирьох точок на полотно, ми отримаємо **c0'**, **c1'**, **c2'** і **c3'**. Якщо **c0-c1** визначає ребро, то ми малюємо лінію від **c0'** на **c1'**. Якщо **c0-c2** визначає ребро, то ми малюємо лінію від **c0'** до **c2'**.

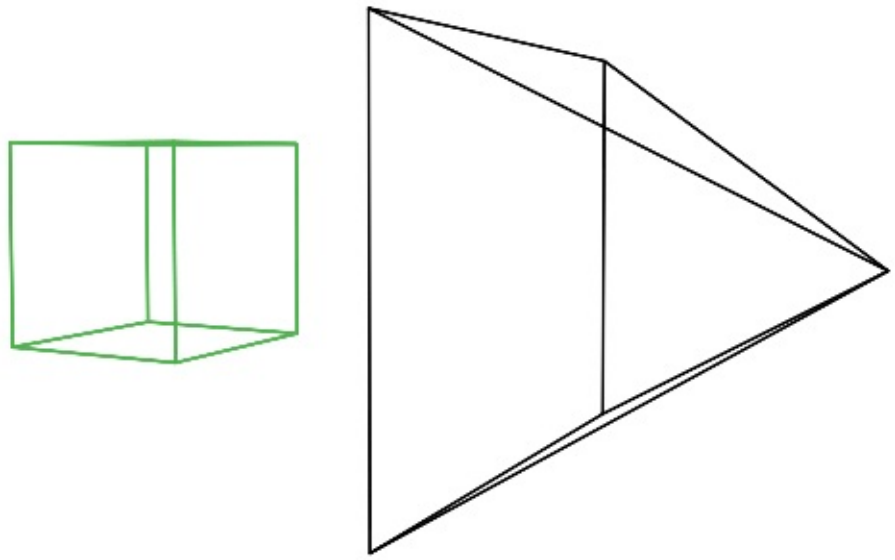


Рисунок *2: проектування чотирьох кутів передньої грані на полотні*

Якщо повторити цю операцію для інших ребер куба, ми в кінцевому підсумку матимемо двовимірне зображення куба на полотні. Так ми створили наше перше зображення за допомогою перспективної проекції (**perspective projection**). Якщо ми постійно повторюватимемо цей процес для кожного об'єкта на сцені, то ми отримуємо зображення сцени, яке виникає з певної **точки зору (vantage point)**. Тільки на початку 15-го століття художники почали розуміти правила перспективної проекції.

Світло і колір (Light and Color)

Після того, як ми знаємо, де намалювати контур тривимірних об'єктів на двовимірній поверхні, ми можемо додати кольору, щоб закінчити картину.

Для того, щоб швидко підсумувати те, що ми тільки що дізналися: ми можемо створити зображення з тривимірної сцени в два етапи. Перший крок складається з проектування форми тривимірних об'єктів на поверхню зображення (або площину зображення). Цей крок не вимагає нічого, крім з'єднувальних ліній від об'єктів до очей. Контур промальовується на полотні, де ці лінії проекції перетинають площину зображення. Як ви вже помітили, це геометричний процес. Другий крок полягає в додаванні кольору до каркасу картинки.

Колір і яскравість об'єкта на сцені, в основному є результатом взаємодії світла з матеріалами об'єкта. Світло складається з фотонів (**photons -** електромагнітні частки), які мають, іншими словами, електричний компонент і магнітний компонент. Вони переносять енергію і коливаються, як звукові хвилі, які подорожують по прямих лініях. Фотони випромінюються різними джерелами світла, найбільш яскравим прикладом є сонце. Якщо група фотонів вдарится в об'єкт, можуть статися три речі: вони можуть або поглинутись, або відбитись, або передатись. Відсоток фотонів, що відбивається, поглинається, і передається варіюється від одного матеріалу до іншого і в цілому впливає на те, як з'являється об'єкт на сцені. Проте, є одне правило, всі матеріали мають спільні характеристики, загальне число фотонів, що прибувають завжди дорівнює сумі відбитих, поглинених і переданих фотонів. Іншими словами, якщо ми маємо 100 фотонів, які висвітлюють точку на поверхні об'єкта, 60 можуть бути поглинені і 40 можуть бути відображені. Загальна кількість рівна - 100. У даному конкретному випадку, ми ніколи не матимемо 70 поглинутих і 60 відбитих, або 20 поглинутих і 50 відбитих, так як загальна кількість переданих, поглинутих і відбитих фотонів має бути 100.

У науці ми розрізняємо тільки два типи матеріалів, метали, які називаються провідниками (**conductors**) і діелектрики (**dielectrics**). Діалектрики включають в себе такі речі як скло, пластик, дерево, вода і т.д. Ці матеріали мають властивість бути електричними ізоляторами (чиста вода являє собою електричний ізолятор). Зверніть увагу, що діелектричний матеріал може бути або прозорим (провідним) або непрозорим (непровідним). Скляні кульки та пластмасові кульки на зображенні нижче є діелектричними матеріалами. Фактично, кожен матеріал має в деякій мірі прозорість (провідність) для деякого виду електромагнітного випромінювання. Рентгенівські промені (X-rays) наприклад можуть пройти через тіло.

Об'єкт також може бути виконаний з композиційного або багатошарового матеріалу. Наприклад, можна мати непрозорий об'єкт (скажімо, до прикладу дерево) з прозорим шаром лаку поверх нього (через що він виглядає в той же час як розсіяний і блискучий, як кольорові пластикові кульки на зображенні нижче).

Давайте зараз розглянемо випадок непрозорих і дифузних об'єктів. Для простоти, ми будемо припускати, що процес поглинання відповідає за колір об'єкта. Біле світло складається з «червоних», «синіх», і «зелених» фотонів. Якщо біле світло освітлює червоний об'єкт, процес поглинання відфільтровує (або поглинає) «зелений» і «сині» фотони. Оскільки об'єкт не поглинає «червоних» фотонів, вони відображаються. Це причина, чому цей об'єкт з'являється червоним. Тепер, причиною чому ми бачимо весь об'єкт, бо деякі з «червоних» фотонів відбивається від об'єкта, прямуючи до нас і потрапляють в очі. Кожна точка на освітленій поверхні чи об'єкт, випромінює (відображає) світлові промені в будь-якому напрямку. Тільки один промінь від кожної точки потрапляє на око перпендикулярно і, отже, може бути побаченим. Наші очі мають фоторецептори, які перетворять світло в нервові сигнали. Наш мозок потім може використовувати ці сигнали для інтерпретації різних затінень і відтінків (саме як, ми не зовсім впевнені). Це дуже спрощений підхід до опису згаданих явищ. Все більш детально пояснено в уроці про колір (який ви можете знайти в розділі Математики і фізики для комп'ютерної графіки - Mathematics and Physics for Computer Graphics).

Подібно концепції перспективної проекції, потрібно якийсь час для людей, щоб зрозуміти світ. Греки розробили теорію бачення, в яких об'єкти видимі завдяки променям світла, що виходять з очей. Арабський вчений Ібн аль-Хайтам (. З 965-1039), був першим, хто пояснив, що ми бачимо об'єкти завдяки сонячним променям світла; потоки крихітних частинок, що рухаються по прямих лініях відбиваються від предметів в наші очі, утворюючи зображення (малюнок 3). Тепер давайте подивимося, як ми можемо імітувати природу світла з комп'ютером!

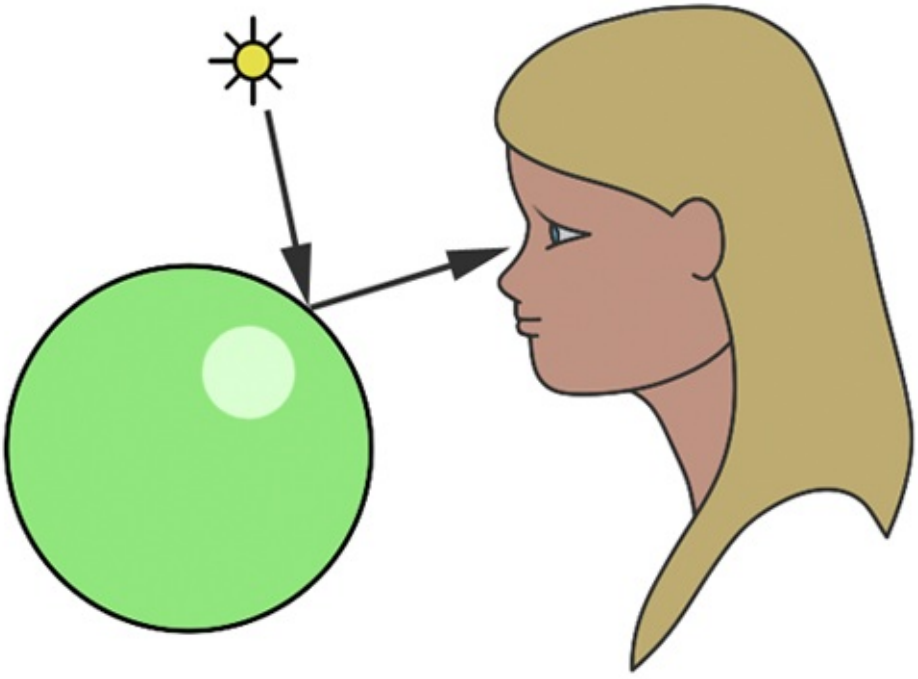


Figure 3: al-Haytham's model.

Алгоритм рейтрейсінгу в двох словах (Raytracing Algorithm in a Nutshell)

Явища описані Ібн аль-Хайтам пояснюють, чому ми бачимо об'єкти. Можна зробити два цікавих зауваження на основі його спостережень: по-перше, без світла ми не можемо бачити нічого, а по-друге, без об'єктів в нашому середовищі, ми не можемо бачити світло. Якби ми подорожували в міжгалактичному просторі, було б щось таке. Якщо немає матерії навколо нас, ми не можемо бачити нічого, крім темряви, навіть якщо фотони потенційно рухаються через цей простір.

Пряме трасування (Forward Tracing)

Якщо ми намагаємося змоделювати процес взаємодії світла та об'єкта в зображення згенероване комп'ютером, тоді ми повинні бути в курсі інших фізичних явищ. У порівнянні із загальним числом променів, відбитих від об'єкта, лише деякі з них досягають поверхні нашого ока. Ось приклад. Уявіть, що ми створили джерело світла, яке випромінює тільки один єдиний фотон одночасно. Тепер давайте подивимося, що відбувається з цим фотоном. Він випромінюється з джерела світла і рухається по прямій лінії, поки він не потрапляє на поверхню нашого об'єкта. Ігноруючи поглинання фотонів, ми можемо припустити, що фотон відбивається у випадковому напрямку. Якщо фотони потрапляють на поверхню нашого ока, ми «бачимо» точку, звідки фотон відбився (рисунок 1).

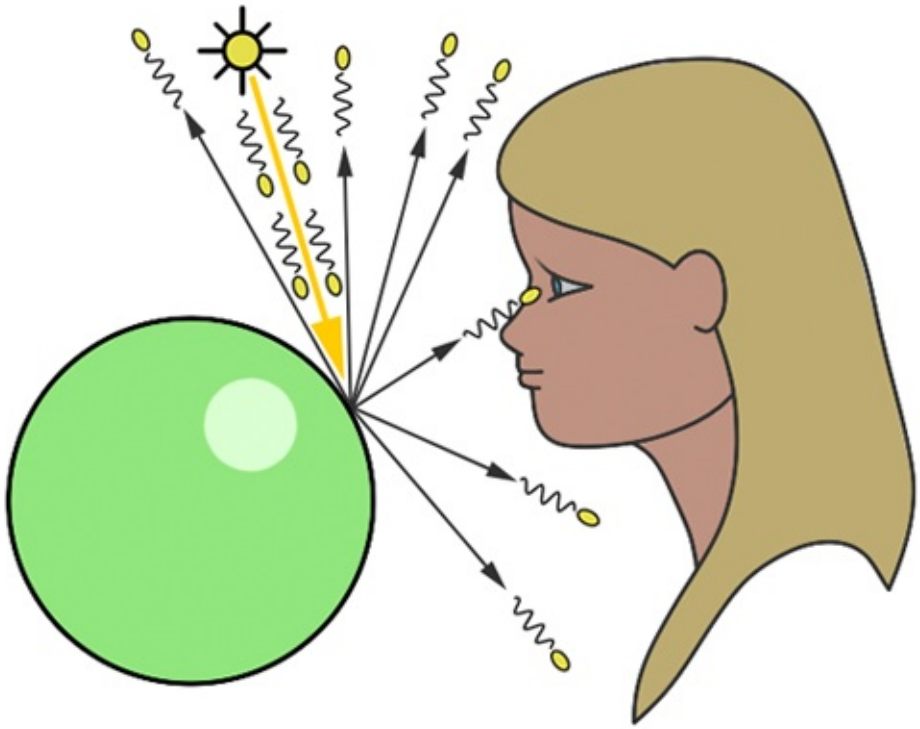
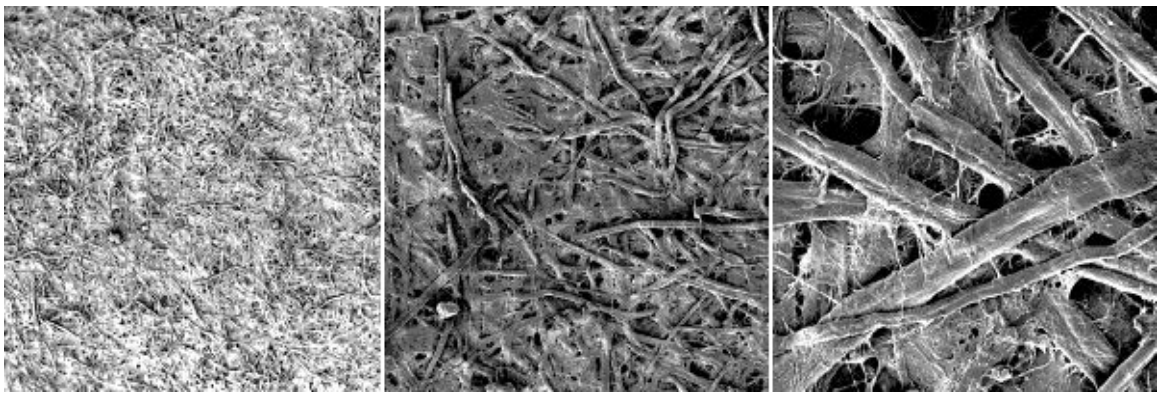


Figure 1: countless photons emitted by the light source hit the green sphere, but only one will reach the eye's surface.



> Питання від читача: вище, ви стверджуєте, що «кожна точка на освітленій поверхні або об'єкті випромінює (відбиває) світлові промені в будь-якому напрямку». Хіба це не суперечать "випадковості"?
>
> ВІДПОВІДЬ: пояснення, чому світло відбивається в усіх можливих напрямках знаходиться поза темою для даного уроку (можна послатися на урок із взаємодії світла і речовини для повного пояснення). Однак, відповісти на ваше запитання можна коротко: так і ні. Звичайно, в природі, справжній фотон відбивається від реальної поверхні в строго визначеному напрямку (і, отже, не є випадковим), який визначається геометрією топології і напрямком попадання фотона в точку перетину. Поверхня дифузного об'єкта здається гладкою, якщо ми подивимося на це нашими очима. Хоча, якщо дивитися на нього з допомогою мікроскопа, ми розуміємо, що мікро-структура дуже складна і зовсім не гладка. Зображення зверху є фотографію паперу з різним збільшенням масштабів. Фотони настільки малі, що вони відбиваються від мікро-особливостей і форм поверхні об'єкту. Якщо промінь світла потрапить на поверхню цього дифузного об'єкта, фотони, що містяться в цьому пучку світла потраплять на дуже різні частини мікро-структури поверхні і, отже, будуть відбиті у великій кількості різних напрямків. У такій великій кількості, що можна сказати, «у всіх можливих напрямках». Якщо ми хочемо змоделювати взаємодію між фотонами і мікроструктурою, тоді ми стріляємо променями у випадкових напрямках, які, статистично кажучи, приблизно те ж саме, якщо б вони були відбиті в усіх можливих напрямках.
>
> Іноді структура матеріалу на макрорівні організована таким чином, що поверня об'єкта може спричинити до того, що світло відбиватиметься в певних напрямках. Це описується як анізотропне відображення і буде пояснено детально в уроці про взаємодію світлових матеріалів. Макро структура матеріалу також може бути причиною незвичайних візуальних ефектів, до прикладу, таких як райдужні, які ми можемо спостерігати в крилах метеликів.

Тепер ми можемо почати дивитися на ситуацію з точки зору комп'ютерної графіки. По-перше, ми замінимо наші очі на площину зображення, яка складатимеься з пікселів. В цьому випадку, фотони, що випускаються потраплять в один з багатьох пікселів на площині зображення, збільшуючи яскравість в цій точці до значення більше нуля. Цей процес повторюється кілька разів, поки всі пікселі не коригуються, створюючи згенероване комп'ютером зображення. Ця техніка називається пряме трасування променів, тому що ми слідуємо шляхом фотона прямо від джерела світла до спостерігача.

Однак, ви бачите потенційну проблему з таким підходом?

Проблема полягає в наступному: в нашому прикладі ми припустили, що відбитий фотон завжди перетинав поверхню очей. Насправді, промені, по суті, відбиваються в кожному можливому напрямку, кожен з яких має дуже, дуже малу ймовірність фактичного потрапляння в очі. Ми потенційно повинні були б відкидати незліченну кількість фотонів від джерела світла, щоб знайти тільки один фотон, який буде потрапляти в очі. Це так працює у природі - незліченна кількість фотонів подорожують у всіх напрямках зі швидкістю світла. У комп'ютерному світі, моделювання взаємодії цих багатьох фотонів з об'єктами на сцені буде просто не практичним рішенням з причин, які ми зараз пояснимо.

Таким чином, ви можете думати: «нам дійсно потрібно випромінювати фотони у випадкових напрямках? Так як ми знаємо позицію очей, чому б просто не відправити фотон в цьому напрямку і бачити, через який піксель в зображенні він проходить, якщо звісно проходить?» Це, безсумнівно, буде одним з можливих способів оптимізації, однак ми можемо використовувати цей метод для деяких типів матеріалів. З причин, які ми пояснимо в наступному уроці про взаємодію світла з речовиною, напрямленість не є важливою для дифузних поверхонь. Бо якщо фотон потрапить на дифузну поверхню, він може бути відбитий в будь-якому напрямку в межах півсфери з центром навколо нормалі в точці контакту. Однак, якщо поверхня є дзеркалом, і не має дифузійні характеристики, промінь може бути відображено тільки в дуже точному напрямі; дзеркальний напрямок (те, що ми навчимося опрацьовувати пізніше). Для такого типу поверхні, ми не можемо штучно змінити напрямок фотона, якщо він насправді повинен слідувати віддзеркаленому напрямку. Це означає, що це рішення не є повністю задовільним.

> ПИТАННЯ ВІД ЧИТАЧІВ: «Чи око дійсно тільки точковий рецептор, або чи воно має поверхню? Навіть якщо приймаюча поверхня дуже дуже мала, воно все ж ще має площу і, отже, більше, ніж точка. Якщо в зона прийому більша, ніж точка, то, звичайно, поверхня буде отримувати більше, ніж просто один з незліченних променів?»
>
> ВІДПОВІДЬ: читач правий. Око не є точковим рецептором, а є поверхневим рецептором, як плівки або ПЗС-матриці в камері. Цей урок просто введення в алгоритм трасування променів і ця тема занадто складна, щоб пояснити в деталях. Камери і людське око мають лінзу, яка фокусує відбиті промені світла на поверхню позаду неї. Якщо лінза мала б дуже малий радіус (що технічно насправді не так) в теорії, ми могли б сказати, що світло, відбите від об'єкта може приходити тільки з одного напрямку. Так працюю точкові камери. Ми будемо говорити про них в уроці про камери.

Навіть, якщо ми все-таки вирішили використовувати цей метод для сцени з лише дифузних об'єктів, ми як і раніше стикнемося з однією серйозною проблемою. Ми можемо візуалізувати процес вистрілювання фотонів світла на сцену, як ніби ви обприскуєте світлові промені (або дрібні частиноки фарби) на поверхню об'єкта. Якщо спрей не досить щільний, в деяких місцях не буде рівномірного освітлення.

Уявіть собі, що ми намагаємося намалювати чайник, роблячи точки білим маркером на чорний аркуш паперу (розглянемо кожну точку як фотон). Як ми бачимо на зображенні нижче, якщо почати з лише кількох фотонів, які перетинатимуться з чайником, то в результаті матимемо багато непокритих ділянок. Якщо ми продовжуємо додавати точки, то щільність фотонів збільшиться доки чайник "майже" повністю покриється фотонами, зроблячи об'єкт легко пізнаваним.

Але вистрілювання 1000 фотонів, або навіть в X разів більше, дійсно ніколи не гарантує, що поверхня нашого об'єкта буде повністю покрита фотонами. Це головний недолік цього методу. Іншими словами, ми, ймовірно, повинні дозволити програмі працювати, поки ми не вирішимо, що вона розприскає досить фотонів на поверхні об'єкта, щоб отримати точне відображення нього. Це означає, що ми повинні були б спостерігати за візуалізованим зображенням для того, щоб вирішити, коли слід припинити програму. У виробничому середовищі, це просто не можливо. До того ж, як ми побачимо, найбільш витратна задача рейтрейсера полягає у знаходженні променево-геометричних перетинів. Створення багатьох фотонів від джерела світла не є проблемою, але, знаходження всіх їх перетинів в межах сцени було б занадто дорого.

Висновок: (Forward ray-tracing) пряме трасування променів (або **(light tracing)** світлове трасування, тому що ми вистрілюємо промені від світла) робить технічно можливим моделювати шлях подорожі світла в природі на комп'ютері. Проте, цей метод, як описано, не є ефективним або практичним. В основоположному документі, що має назву «Покращена модель освітлення для затемненого дисплея» і опублікованого в 1980 році, **(Turner Whitted)** Тернер Вітед (один з перших дослідників в галузі комп'ютерної графіки) написав:

> «Очевидним підхід до трасування променів є те, що промені світла йдуть від джерела через свої шляхи, поки вони не зіткнуться зі спостерігачем. Оскільки лише деякі дійдуть до глядача, цей підхід є марнотратним. У другому підході, запропонованому Аппель, промені проходять в зворотному напрямку, від глядача до об'єктів на сцені».

Тепер ми поглянемо на другий спосіб, запропонований Вітедом.

Зворотнє трасування (Backward Tracing)

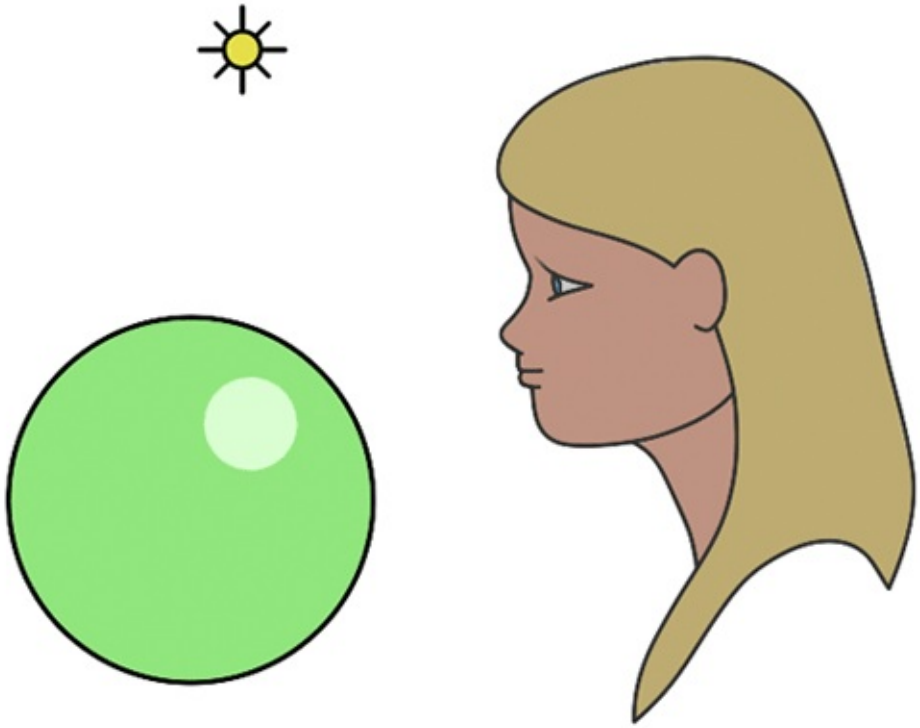


Figure 2: backward ray-tracing. We trace a ray from the eye to a point on the sphere, then a ray from that point to the light source.

Замість того, щоб трасувати промені від джерела світла до рецептора (наприклад, такого як наше око), ми простежимо промені в зворотному напрямку від рецептора до об'єктів. Оскільки цей напрямок протилежний тому, що відбувається в природі, він слушно називається зворотним трасуванням променів або очним трасуванням, тому що ми вистрілюємо промені від положення очей (рисунок 2). Цей метод забезпечує зручне рішення вади прямого трасування променів.

Оскільки наші симуляції не можуть бути настільки ж швидкими і такими досконалими, як природа, ми повинні піти на компроміс і простежити промінь з очей на сцену. Якщо промінь потрапляє на об'єкт, то ми дізнаємося, скільки світла він отримує, кидаючи ще один промінь (так званий промінь світла або тіні) від точки попадання до світла сцени. Іноді цей «промінь світла» перегороджується іншим об'єктом зі сцени, а це означає, що наша справжня точка попадання знаходиться в тіні; вона не отримає освітлення від світла. З цієї причини, ми не називаємо ці промені світловими променями, а натомість тіньові промені (**shadow rays**). У CG літературі, перший промінь, який ми стріляємо з ока на сцену називається первинний промінь (**primary ray**), видимий промінь (**visibility ray**) або промінь камери (**camera ray**).

> В цьому уроці ми використовували пряме трасування, щоб описати ситуацію, коли промені відходили від світла, на відміну від зворотного трасування променів, які виходять з камери. Однак деякі автори використовують ці терміни навпаки. Пряме трасування для них означає вихід променів з камери, тому що це найбільш поширений спосіб трасування, який використовується в CG. Щоб уникнути плутанини, ви також можете використовувати термін світлове і очне трасування, які є більш недвозначними. Ці терміни частіше використовується в контексті двонаправленого відстеження шляху (дивіться розділ **Light Transport**).

Висновок

У комп'ютерній графіці поняття випускання променів або від світла, або від ока, називається шляхом трасування. Термін трасування променів також може бути використаний, але концепція шляху трасування передбачає, що цей метод створює комп'ютерно генеровані зображення, які залежить від проходження по шляху від світла до камери (або навпаки). Роблячи це фізично реалістичним чином, ми можемо легко імітувати оптичні ефекти, такі як каустик або відбиття світла на іншій поверхні в сцені (непрямого освітлення). Ці теми будуть обговорюватися в інших уроках.

Рисунок 3: каустична поверхня (caustic surface).

Реалізація рейтрейсінгу (Implementing the Raytracing Algorithm)

Ми розглянули все, що мали сказати! Тепер ми готові написати наш перший рейтрейсер. Тепер ви повинні бути в змозі здогадатися, як працює алгоритм трасування променів.

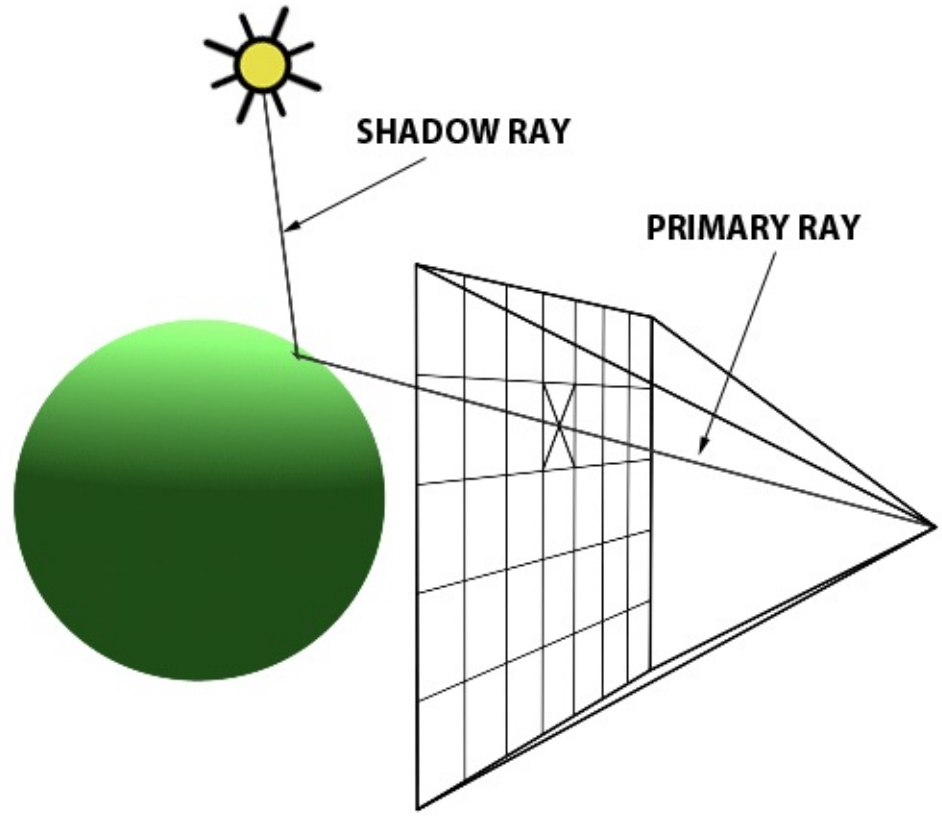


Figure 1: we shoot a primary ray through the center of the pixel to check for a possible object intersection. When we find one we then cast a shadow ray to find out if the point is illuminated or in shadow.

Перш за все, знайдіть час, щоб помітити, що поширення світла в природі - це просто незліченна кількість променів, що випускаються від джерел світла, які відбиваються навколо, поки вони не потрапилять на поверхню нашого ока. Трасування променів зрозуміле, бо воно засноване безпосередньо на тому, що насправді відбувається навколо нас. Крім того, що воно йде по шляху світла в зворотному порядку, це ніщо інше ніж досконалий природний симулятор.

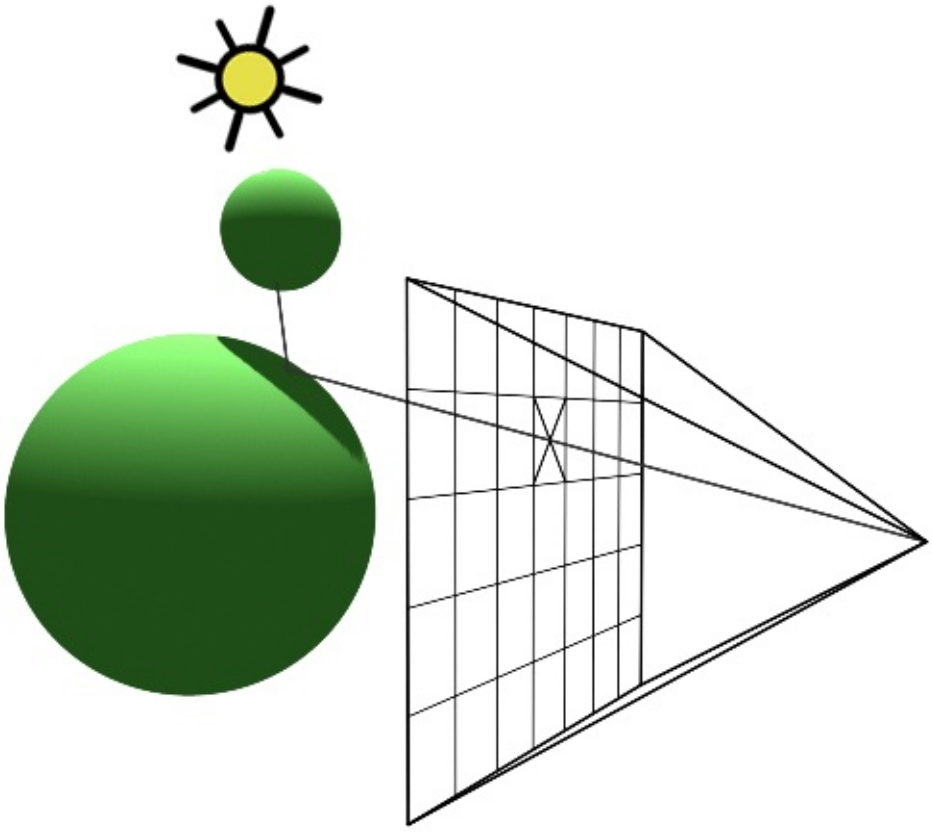


Figure 2: the small sphere cast a shadow on the large sphere. The shadow ray intersects the small sphere before it gets to the light.

Алгоритм трасування променів приймає зображення, складається з пікселів. Через кожен піксель в зображенні, він стріляє первинний промінь на сцену. Напрямок цього первинного променя отримують шляхом відстеження лінії від очей до центру цього пікселя. Після того, як ми маємо встановлений напрямок основного променя, ми перевіряємо кожен об'єкт на сцені, щоб побачити, чи він перетинається з будь-яким з них. У деяких випадках первинний промінь перетне більше одного об'єкта. Коли це станеться, ми вибираємо об'єкт, точка перетину якого знаходиться ближче до ока. Потім ми стріляємо тіньовим променем з точки перетину до світла (рисунок 6, зверху). Якщо цей конкретний промінь не перетинає об'єкт на своєму шляху до світла, точка попадання освітлюється. Якщо він перетинається з іншим об'єктом, тоді цей об'єкт відкидає тінь на нього (рисунок 2).

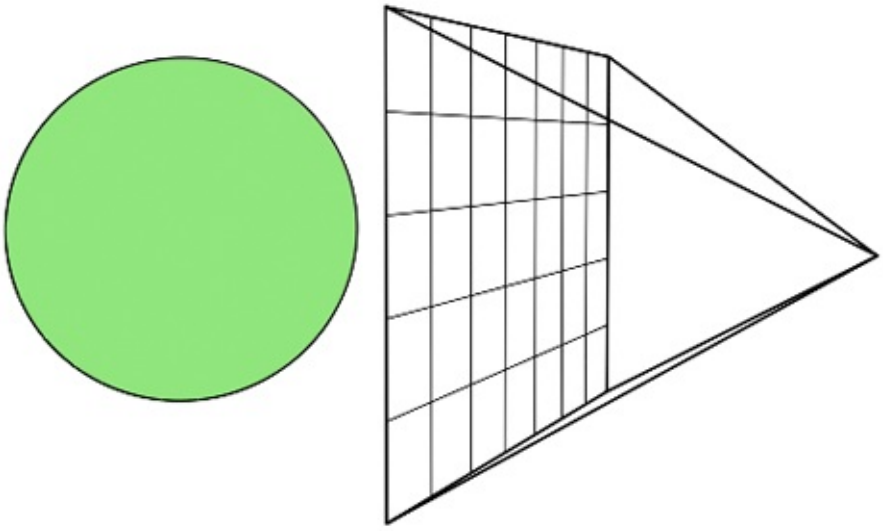


Figure 3: to render a frame, we shoot a primary ray for each pixel of the frame buffer

Якщо повторити цю операцію для кожного пікселя, ми отримуємо двовимірне зображення нашої тривимірної сцени (рисунок 3).

Ось реалізація алгоритму в псевдокоді:

```
for (int j = 0; j < imageHeight; ++j) {
    for (int i = 0; i < imageWidth; ++i) {
        // compute primary ray direction
        Ray primRay;
        computePrimRay(i, j, &primRay);
        // shoot prim ray in the scene and search for intersection
        Point pHit;
        Normal nHit;
        float minDist = INFINITY;
        Object object = NULL;
        for (int k = 0; k < objects.size(); ++k) {
            if (Intersect(objects[k], primRay, &pHit, &nHit)) {
                float distance = Distance(eyePosition, pHit);
                if (distance < minDistance) {
                    object = objects[k];
                    minDistance = distance; // update min distance
                }
            }
        }
        if (object != NULL) {
            // compute illumination
            Ray shadowRay;
            shadowRay.direction = lightPosition - pHit;
            bool isShadow = false;
            for (int k = 0; k < objects.size(); ++k) {
                if (Intersect(objects[k], shadowRay)) {
                    isInShadow = true;
                    break;
                }
            }
        }
        if (!isInShadow)
            pixels[i][j] = object->color * light.brightness;
        else
            pixels[i][j] = 0;
    }
}
```

Краса рейтрейсінгу, як можна побачити, що це займає всього кілька рядків коду; можна було б, звичайно, написати базовий рейтрейсер в 200 рядків. На відміну від інших алгоритмів, таких як сканлайн рендер, рейтрейсер забирає дуже мало зусиль для реалізації.

Ця методика була вперше описана Артуром Аппелем в 1969 році в документі, який називався «Деякі методи для затінення машинним рендерингом твердих тіл». Таким чином, якщо цей алгоритм такий чудовий, чому не він замінює всі інші алгоритми рендеринга? Основна причина, в той час (і навіть сьогодні в деякій мірі), була швидкість. Як Аппель згадує в своїй статті:

> «Цей метод (рейтрейсінг) споживає дуже багато часу, як правило, для корисних результатів потрібно кілька тисяч секунд, так багато розрахунків, як і в **wire frame drawing.** Близько половини цього часу присвячено визначенню відстані від точки до точки у відношенні до проекції і сцени».

Іншими словами, це повільно (але, як Kajiya - один з найвпливовіших дослідників комп'ютерної графіки в історії - колись сказав: «трасування променів не повільне - проблема в комп'ютерах»). Витрачається дуже багато часу, щоб знайти перетин між променями і геометрією. Протягом багатьох десятиліть, швидкість цього алгоритму є основним недоліком рейтрейсінгу. Однак, комп'ютери стають швидшими, а проблема все менша і менша. Хоча одна річ повинна бути ще сказана: порівняно з іншими методами, такими як алгоритм Z-буфера, трасування променів ще набагато повільніше. Проте, на сьогоднішній день, з швидкими комп'ютерами, ми можемо обчислити кадр, який обчислювався раніше одну годину, протягом декількох хвилин або менше. Насправді, рейтрейсери реального часу та інтерактивні є гарячою темою.

Підводячи підсумок, важливо пам'ятати (знову ж), що процедура рендерингу може бути розглянута як два окремих процеси. Один крок визначає, чи є точка видною на певному пікселі (видима частина), інший крок, чи точка затінюється (частина затінення). На жаль, обидва з двох етапів вимагать дорогої і часовитратної променевої геометрії для визначення перетинів. Алгоритм елегантний і потужний, але змушує нас міняти час рендеринга для точності і навпаки. З того часу як Appel опублікував статтю багато досліджень було зроблено для прискорення процедури перетину об'єкта променями. Комбінуючи ці схеми прискорення з новою технологією в комп'ютерах, стало простіше використовувати рейтрейсинг, де він використовувався майже в усіх випадках виготовлення рендерингового програмного забезпечення.

Додаємо відбиття і заломлення (Adding Reflection and Refraction)

Інша перевага трасування променів в тому, що, розширюючи ідею поширення променя, ми можемо дуже легко моделювати ефекти, як відбиття і заломлення, обидва з яких зручні при моделюванні матеріалів зі скла або дзеркальних поверхонь. У 1979 році, в документі під назвою «Удосконалена модель освітлення для затемненого дисплея», Тернер Вітед був першим, хто описав, як розширити рейтрейсинговий алгоритм Аппеля для більш просунутого рендерингу. Ідея Вітеда розширила модель Аппеля по вистрілюванні променів, щоб включати розрахунки також і для відбиття та заломлення.

В оптиці, віддзеркалення і заломлення добре відомі явища. Хоча пізніше цілий урок буде присвячено віддзеркаленню і заломленню, ми швидко поглянемо на те, що потрібно, щоб імітувати їх. Ми візьмемо приклад зі скляної кулі, об'єкт, який має як заломлювальні так і віддзеркалювальні властивості. Так, як ми знаємо напрямок променя, що перетинає м'яч, легко обчислити, що відбувається з ним. Напрямки віддзеркалення і заломлення засновані на нормалі в точці перетину і напрямоку входження променя (первинний промінь). Для того, щоб обчислити напрямок рефракції ми також повинні вказати показник заломлення (**index of refraction**) матеріалу. Хоча ми вже говорили раніше, що промені подорожують по прямій лінії, ми можемо візуалізувати заломлення променя по кривій. Коли фотон потрапляє на об'єкт з іншого

матеріалу (і, таким чином, інший показник заломлення), його напрямок змінюється. По науковому це буде обговорюватися більш докладно пізніше. Так як ми пам'ятаємо, що ці два ефекти залежать від нормалі і напрямку вхідного променя, і що заломлення залежать від показника заломлення матеріалу, то ми готові рухатися далі.

Крім того, ми також повинні розуміти те, що такий об'єкт як скляна куля віддзеркалює і заломлює в один і той же час. Нам потрібно обчислити і те, і інше для даної точки на поверхні, але як же ми змішаємо їх разом? Чи повинні ми взяти 50% результату віддзеркаленя і змішати його з 50% в результату заломлення? На жаль, все набагато складніше, ніж це. Змішування значень залежить від кута між первинним променем (або напрямком спостереження) і нормалі об'єкта та показника заломлення. Однак, на щастя для нас є рівняння, яке обчислює, як саме кожен з них повинен бути змішаний. Це рівняння знають як рівняння Френеля. Залишаючись стислим, все, що нам потрібно знати, на даний момент, є те, що воно існує, і це буде корисно в майбутньому при визначенні значення змішування.

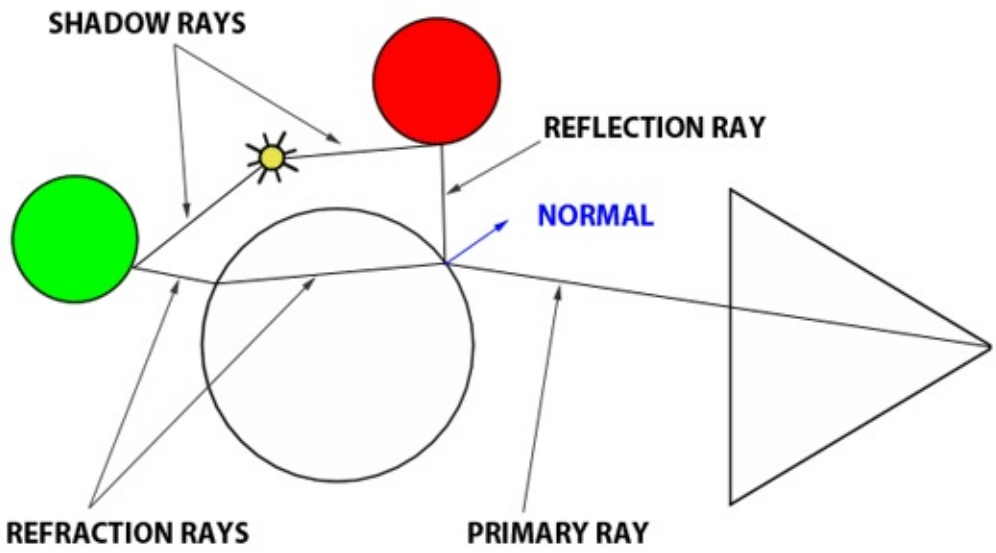
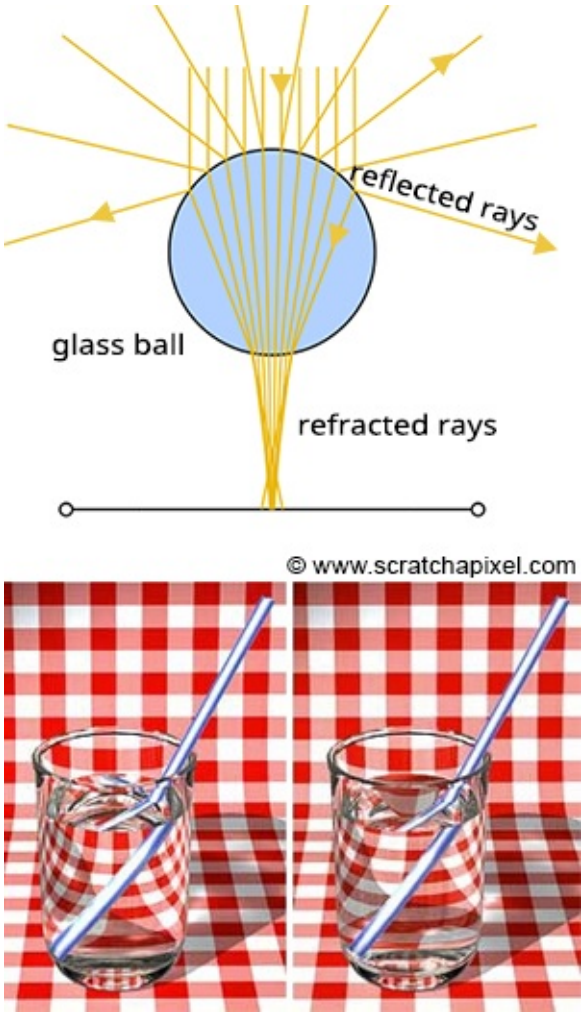


Рисунок *1: використовуємо оптичні закони для обрахування віддзеркалення і заломлення*

Отже, давайте резюмувати. Як працює алгоритм Вітеда? Випускаємо первинний промінь з очей і маємо найближче перетинання (якщо таке є) з об'єктами на сцені. Якщо промінь потрапить в об'єкт, який не є дифузним або непрозорим об'єктом, ми повинні зробити додаткову обчислювальну роботу. Для обчислення результуючого кольору в цій точці, скажімо для прикладу, на скляній кулі, вам потрібно обчислити колір віддзеркалення і колір заломлення і змішати їх разом. Пам'ятайте, що ми робимо це в три етапи. Обчислюємо колір віддзеркалення (**reflection**), обчислюємо колір заломлення (**refraction**), а потім застосовуємо рівняння Френеля.

Спочатку обчислимо напрямок віддзеркалення. Для цього нам потрібні дві речі: нормаль в точці перетину і напрямок первинного променя. Після того, як ми отримаємо напрям віддзеркалення, ми випускаємо новий промінь в цьому напрямку. Повертаючись до нашого старого наприкладу, можна сказати, що промінь віддзеркалення потрапляє в червону кулю. Використовуючи алгоритм Аппеля, ми знаходимо, скільки світла досягає цієї точки на червоній кулі, якщо вистрілити тіньовим променем до світла. Одержимо колір (чорний, якщо вона затінена), який потім множиться на інтенсивність світла і повернулися до поверхні скляної кулі.



Тепер ми робимо те ж саме для заломлення. Слід зазначити, що, так як промінь проходить через скляну кулю, він називається (**transmission ray**) промінь проходження (світло проходить від однієї сторони кулі до іншої; він був переданий - transmitted). Для того, щоб обчислити напрямок проходження нам потрібна нормаль в точці попадання, напрямок первинного променя, і показник заломлення матеріалу (в даному прикладі це може бути щось на зразок 1,5 для скляного матеріалу). З новим обчисленим напрямком, промінь заломлення продовжує свій курс на іншу сторону скляної кулі. Повторюється знову, тому що змінюється середовище, то промінь заломлюється ще раз. Як можна бачити в прилеглому зображенні, напрямок променя змінюється, коли промінь входить і виходить зі скляного об'єкта. Заломлення відбувається кожного разу,

коли є зміна середовища, у нас два середовища, з одиного промінь виходить і в інше він потрапить, маємо різний показник заломлення. Як ви, напевно, знаєте, показник заломлення повітря дуже близький до 1, а показник заломлення скла становить близько 1,5. Заломлення має для ефекту злегка відхилити промінь. Цей процес є те, що робить появу об'єктів зміщеними при перегляді через або на об'єкти з різними показниками заломлення. Давайте тепер уявімо, що, коли заломлений промінь залишає скляну кулю він потрапляє в зелену сферу. Там ми знову обчислимо локальне освітлення в точці перетину між зеленою сферою і заломленим променем (стріляючи тіньовий промінь). Колір (чорний колір, якщо вона затінена) потім множиться на інтенсивність світла і повернулися до поверхні скляної кулі.

Нарешті, ми обчислюємо рівняння Френеля. Нам потрібен показник заломлення скляної кулі, кут між первинним променем і нормаллю в точці попадання. Використовуючи скалярний добуток (ми пояснимо це пізніше), рівняння Френеля повертає два змішаних значення.

Ось трохи псевдо-коду, щоб показати, як це працює:

```
// compute reflection color
color reflectionCol = computeReflectionColor();
// compute refraction color
color refractionCol = computeRefractionColor();
float Kr; // reflection mix value
float Kt; // refraction mix value
fresnel(refractiveIndex, normalHit, primaryRayDirection, &Kr, &Kt);
// mix the two
color glassBallColorAtHit = Kr * reflectionColor + (1 - Kr) * refractionColor;
```

І останнє, прекрасна річ у цьому алгоритмі у тому, що він є (**recursive**) рекурсивний (але і прокляття в деякому сенсі, теж!). У випадку, що ми вже вивчали, промінь віддзеркалення потрапляє в червону, непрозору сферу і промінь заломлення потрапляє в зелену, непрозору, і дифузну сферу. Тим не менш, ми будемо уявляти собі, що червоні та зелені кулі є також скляними кульками. Це є серйозним недоліком алгоритму трасування променів і може бути просто жахливими в деяких випадках. Уявіть собі, що наша камера знаходиться в коробці, яка має тільки світловідбиваючі грані. Теоретично, промені вловлюються і будуть продовжувати відбиваючись від стін боксу нескінченно (або поки ви не зупините симуляцію). З цієї причини, ми повинні встановити довільну межу, яка запобігатиме взаємодії променів, і, таким чином, нескінченній рекурсії. Кожен раз, коли промінь або відбитий або заломлений його глибина (**depth**) збільшується. Ми просто зупинемо процес рекурсії, коли глибина променя стане більшою, ніж максимальна глибина рекурсії.роменів, і, таким чином, нескінченній рекурсії.

Пишемо простий Рейстрейсер (Writing a Basic Ray Tracer)

Ми отримали досить багато листів від читачів з проханням: «Ну, якщо це так легко зробити, ви не можете надати нам реальний приклад?» Це не по плану (оскільки ідея полягає в тому, щоб написати рендер крок за кроком), але ми написали мінімалістичний рейтрейсер, який займає близько 300 сот рядків приблизно через декілька годин. Незважаючи на те, що ми не обов'язково пишаємося цією реалізацією, ми просто хотіли показати, що, коли хтось знає ці методи добре, реалізувати їх не складно. Вихідний код доступний для завантаження. Ми не робили і не хотіли витрачати час на коментування цієї програми. Вона була написана досить швидко, так що є місце для вдосконалення. У цій версії рейтрейсера ми зробили світло видимим (це сфера), тому його відображення з'явиться на кулях. Коли скляні кульки повністю прозорі (білі), то їх інколи складно побачити, так що в нашому прикладі, ми їх трохи тонували (червоний). У реальному світі прозоре скло не обов'язково видиме. Це залежить від навколишнього середовища (ви коли-небудь ходили через двері, зроблені зі скла?). Зверніть увагу, що отримане зображення не зовсім точне. Тінь під прозорою червоною областю не повинна бути повністю непрозорою. Ми дізнаємося в майбутньому уроків, як можна легко виправити цю візуальну неточність. Ми також реалізовали інші функції, такі як підроблений Френель (використовуючи поверхневе співвідношення (**facing ratio**)) і заломлення. Всі ці речі будуть вивчатися пізніше, так що не хвилюйтеся, якщо ви не розумієте їх чітко в даний момент. В крайньому випадку, тепер у вас є невелика програма, щоб гратися.

Для компіляції програми завантажте вихідний код на жорсткий диск. Вам знадобиться компілятор c++ (ми використовуємо gcc під Linux). Ця програма нічого специфічного для компіляції не потребує. Візьміть shell Linux і введіть наступну команду:

```
c++ -O3 -o raytracer raytracer.cpp
```

Для того, щоб створити зображення, запустіть програму, набравши ./raytracer в shell. Зачекайте кілька секунд. Коли програма завершиться, ви повинні мати файл з ім'ям untitled.ppm на вашому диску. Ви можете відкрити цей файл за допомогою Photoshop, Preview (на Mac) або Gimp (на Linux). Крім того, перевірте урок, присвячений читанню і відображенню PPM зображення.

Ось одина з можливих реалізацій класичного рекурсивного алгоритму рейтрейсера в псевдокоді:

```
#define MAX_RAY_DEPTH 3
```

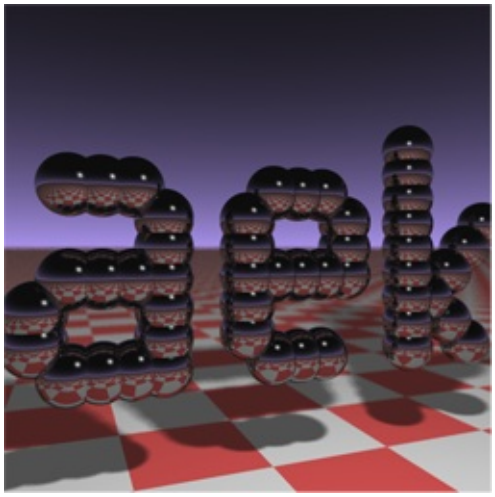
```
color Trace(const Ray &ray, int depth)
{
    Object *object = NULL;
    float minDist = INFINITY;
    Point pHit;
    Normal nHit;
    for (int k = 0; k < objects.size(); ++k) {
        if (Intersect(objects[k], ray, &pHit, &nHit)) {
            // ray origin = eye position of it's the prim ray
            float distance = Distance(ray.origin, pHit);
            if (distance < minDistance) {
                object = objects[i];
                minDistance = distance;
            }
        }
    }
    if (object == NULL)
        return 0;
    // if the object material is glass, split the ray into a reflection
    // and a refraction ray.
    if (object->isGlass && depth < MAX_RAY_DEPTH) {
        // compute reflection
        Ray reflectionRay;
        reflectionRay = computeReflectionRay(ray.direction, nHit);
        // recurse
        color reflectionColor = Trace(reflectionRay, depth + 1);
        Ray refractioRay;
        refractionRay = computeRefractionRay(
            object->indexOfRefraction,
            ray.direction,
            nHit);
        // recurse
        color refractionColor = Trace(refractionRay, depth + 1);
        float Kr, Kt;
        fresnel(
            object->indexOfRefraction,
            nHit,
            ray.direction,
            &Kr,
            &Kt);
        return reflectionColor * Kr + refractionColor * (1-Kr);
    }
    // object is a diffuse opaque object
    // compute illumination
    Ray shadowRay;
    shadowRay.direction = lightPosition - pHit;
    bool isShadow = false;
    for (int k = 0; k < objects.size(); ++k) {
        if (Intersect(objects[k], shadowRay)) {
            // hit point is in shadow so just return
            return 0;
        }
```

```
    }
    // point is illuminated
    return object->color * light.brightness;
}

// for each pixel of the image
for (int j = 0; j < imageHeight; ++j) {
    for (int i = 0; i < imageWidth; ++i) {
        // compute primary ray direction
        Ray primRay;
        computePrimRay(i, j, &primRay);
        pixels[i][j] = Trace(primRay, 0);
    }
}
```

Мінімальний рейтрейсер (A Minimal Ray Tracer)

Багато років тому, дослідник Пол Хекберт написав рейтрейсер, що може «поміститися на BUSSINESS карті». Ідея полягала в тому, щоб написати мінімальний рейтрейсер променів на C / C ++, настільки малий, щоб він міг би надрукувати його на задній частині його візитної картки (більш детальну інформацію про цю ідею можна знайти в статті, яку він написав у Graphics Gems IV). Багато програмістів з тих пір, вже пробували себе в кодуванні цієї вправи. Нижче ви можете знайти версію, написану Ендрю Кенслер. Зображення знизу, є результатом його програми. Зверніть увагу на глибину ефекту поля (об'єкти розмиваються на відстані). Це досить дивно, щоб було б спроможним створити досить складне зображення з декількома рядками коду.



```
#include <stdlib.h>   // card > aek.ppm
#include <stdio.h>
#include <math.h>
typedef int i;
typedef float f;
struct v {
f x,y,z;
v operator+(v r){return v(x+r.x,y+r.y,z+r.z);}
v operator*(f r){return v(x*r,y*r,z*r);}
f operator%(v r){return x*r.x+y*r.y+z*r.z;}
v(){}
v operator^(v r){return v(y*r.z-z*r.y,z*r.x-x*r.z,x*r.y-y*r.x);}
v(f a,f b,f c){x=a;y=b;z=c;}
v operator!(){return*this*(1/sqrt(*this%*this));}};
i G[]={247570,280596,280600,249748,18578,18577,231184,16,16};
f R(){return(f)rand()/RAND_MAX;}
i T(v o,v d,f&t,v&n){
t=1e9;
i m=0;
f p=-o.z/d.z;
if(.01<p)t=p, n=v(0,0,1), m=1;
for(i k=19;k--;)
for(i j=9;j--;)
if(G[j]&1<<k){v p=o+v(-k,0,-j-4);
f b=p%d, c=p%p-1, q=b*b-c;
if(q>0){f s=-b-sqrt(q);
if(s<t&&s>.01) t=s, n=!(p+d*t),m=2;}}return m;}
v S(v o,v d){f t;v n;i m=T(o,d,t,n);
if(!m)return v(.7,.6,1)*pow(1-d.z,4);
v h=o+d*t,l=!(v(9+R(),9+R(),16)+h*-1),r=d+n*(n%d*-2);
f b=l%n;
if(b<0||T(h,l,t,n))b=0;
f p=pow(l%r*(b>0),99);
if(m&1){h=h*.2; return((i)(ceil(h.x)+ceil(h.y))&1?v(3,1,1):v(3,3,3))*(b*.2+.1);}
return v(p,p,p)+S(h,r)*.5;}i main(){printf("P6 512 512 255 ");
v g=!v(-6,-16,0),a=!(v(0,0,1)^g)*.002,b=!(g^a)*.002,c=(a+b)*-256+g;
for(i y=512;y--;)
for(i x=512;x--;){v p(13,13,13);
for(i r=64;r--;){v t=a*(R()-.5)*99+b*(R()-.5)*99;
p=S(v(17,16,8)+t,!(t*-1+(a*(R()+x)+b*(y+R())+c)*16))*3.5+p;}
printf("%c%c%c",(i)p.x,(i)p.y,(i)p.z);}}
```

Для компіляції програми завантажте вихідний код на жорсткий диск. Вам знадобиться компілятор c++ (ми використовуємо gcc під Linux). Ця програма нічого специфічного для компіляції не потребує. Візьміть shell Linux і введіть наступну команду:

```
c++ -O3 -o min ray minray.cpp
// or
clang++ -O3 -o min ray minray.cpp
```

Щоб згенерувати зображення, запускаємо програму наступним чином у командному рядку:

```
min ray > minray.ppm
```

```
// A very basic raytracer example.

// Instructions to compile this program:
// c++ -o raytracer -O3 -Wall raytracer.cpp

#include <cstdlib>
#include <cstdio>
#include <cmath>
#include <fstream>
#include <vector>
#include <iostream>
#include <cassert>

#if defined __linux__ || defined __APPLE__
// "Compiled for Linux
#else
// Windows doesn't define these values by default, Linux does
#define M_PI 3.141592653589793
#define INFINITY 1e8
#endif

template<typename T>
class Vec3
{
public:
    T x, y, z;
    Vec3() : x(T(0)), y(T(0)), z(T(0)) {}
    Vec3(T xx) : x(xx), y(xx), z(xx) {}
    Vec3(T xx, T yy, T zz) : x(xx), y(yy), z(zz) {}
    Vec3& normalize()
    {
        T nor2 = length2();
        if (nor2 > 0) {
            T invNor = 1 / sqrt(nor2);
            x *= invNor, y *= invNor, z *= invNor;
        }
        return *this;
    }
    Vec3<T> operator * (const T &f) const { return Vec3<T>(x * f, y * f, z * f); }
    Vec3<T> operator * (const Vec3<T> &v) const { return Vec3<T>(x * v.x, y * v.y, z *
 v.z); }
    T dot(const Vec3<T> &v) const { return x * v.x + y * v.y + z * v.z; }
    Vec3<T> operator - (const Vec3<T> &v) const { return Vec3<T>(x - v.x, y - v.y, z -
 v.z); }
    Vec3<T> operator + (const Vec3<T> &v) const { return Vec3<T>(x + v.x, y + v.y, z +
 v.z); }
    Vec3<T>& operator += (const Vec3<T> &v) { x += v.x, y += v.y, z += v.z; return *th
is; }
    Vec3<T>& operator *= (const Vec3<T> &v) { x *= v.x, y *= v.y, z *= v.z; return *th
is; }
    Vec3<T> operator - () const { return Vec3<T>(-x, -y, -z); }
    T length2() const { return x * x + y * y + z * z; }
```

```
    T length() const { return sqrt(length2()); }
    friend std::ostream & operator << (std::ostream &os, const Vec3<T> &v)
    {
        os << "[" << v.x << " " << v.y << " " << v.z << "]";
        return os;
    }
};

typedef Vec3<float> Vec3f;

class Sphere
{
public:
    Vec3f center;                           /// position of the sphere
    float radius, radius2;                  /// sphere radius and radius^2
    Vec3f surfaceColor, emissionColor;      /// surface color and emission (light)
    float transparency, reflection;         /// surface transparency and reflectivity
    Sphere(
        const Vec3f &c,
        const float &r,
        const Vec3f &sc,
        const float &refl = 0,
        const float &transp = 0,
        const Vec3f &ec = 0) :
        center(c), radius(r), radius2(r * r), surfaceColor(sc), emissionColor(ec),
        transparency(transp), reflection(refl)
    { /* empty */ }

    // Compute a ray-sphere intersection using the geometric solution

    bool intersect(const Vec3f &rayorig, const Vec3f &raydir, float &t0, float &t1) co
nst
    {
        Vec3f l = center - rayorig;
        float tca = l.dot(raydir);
        if (tca < 0) return false;
        float d2 = l.dot(l) - tca * tca;
        if (d2 > radius2) return false;
        float thc = sqrt(radius2 - d2);
        t0 = tca - thc;
        t1 = tca + thc;

        return true;
    }
};

// This variable controls the maximum recursion depth

#define MAX_RAY_DEPTH 5

float mix(const float &a, const float &b, const float &mix)
{
    return b * mix + a * (1 - mix);
```

```
}

/* This is the main trace function. It takes a ray as argument (defined by its origin
and direction).
We test if this ray intersects any of the geometry in the scene. If the ray intersects
 an object,
we compute the intersection point, the normal at the intersection point, and shade thi
s point using this
information. Shading depends on the surface property (is it transparent, reflective, d
iffuse).
The function returns a color for the ray. If the ray intersects an object that is the
color of the
object at the intersection point, otherwise it returns the background color. */

Vec3f trace(
    const Vec3f &rayorig,
    const Vec3f &raydir,
    const std::vector<Sphere> &spheres,
    const int &depth)
{
    //if (raydir.length() != 1) std::cerr << "Error " << raydir << std::endl;
    float tnear = INFINITY;
    const Sphere* sphere = NULL;
    // find intersection of this ray with the sphere in the scene
    for (unsigned i = 0; i < spheres.size(); ++i) {
        float t0 = INFINITY, t1 = INFINITY;
        if (spheres[i].intersect(rayorig, raydir, t0, t1)) {
            if (t0 < 0) t0 = t1;
            if (t0 < tnear) {
                tnear = t0;
                sphere = &spheres[i];
            }
        }
    }
    // if there's no intersection return black or background color
    if (!sphere) return Vec3f(2);
    Vec3f surfaceColor = 0; // color of the ray/surfaceof the object intersected by th
e ray
    Vec3f phit = rayorig + raydir * tnear; // point of intersection
    Vec3f nhit = phit - sphere->center; // normal at the intersection point
    nhit.normalize(); // normalize normal direction
    // If the normal and the view direction are not opposite to each other
    // reverse the normal direction. That also means we are inside the sphere so set
    // the inside bool to true. Finally reverse the sign of IdotN which we want
    // positive.
    float bias = 1e-4; // add some bias to the point from which we will be tracing
    bool inside = false;
    if (raydir.dot(nhit) > 0) nhit = -nhit, inside = true;
    if ((sphere->transparency > 0 || sphere->reflection > 0) && depth < MAX_RAY_DEPTH)
 {
        float facingratio = -raydir.dot(nhit);
        // change the mix value to tweak the effect
        float fresneleffect = mix(pow(1 - facingratio, 3), 1, 0.1);
```

```
        // compute reflection direction (not need to normalize because all vectors
        // are already normalized)
        Vec3f refldir = raydir - nhit * 2 * raydir.dot(nhit);
        refldir.normalize();
        Vec3f reflection = trace(phit + nhit * bias, refldir, spheres, depth + 1);
        Vec3f refraction = 0;
        // if the sphere is also transparent compute refraction ray (transmission)
        if (sphere->transparency) {
            float ior = 1.1, eta = (inside) ? ior : 1 / ior; // are we inside or outsi
de the surface?
            float cosi = -nhit.dot(raydir);
            float k = 1 - eta * eta * (1 - cosi * cosi);
            Vec3f refrdir = raydir * eta + nhit * (eta *  cosi - sqrt(k));
            refrdir.normalize();
            refraction = trace(phit - nhit * bias, refrdir, spheres, depth + 1);
        }
        // the result is a mix of reflection and refraction (if the sphere is transpar
ent)
        surfaceColor = (
            reflection * fresneleffect +
            refraction * (1 - fresneleffect) * sphere->transparency) * sphere->surface
Color;
    }
    else {
        // it's a diffuse object, no need to raytrace any further
        for (unsigned i = 0; i < spheres.size(); ++i) {
            if (spheres[i].emissionColor.x > 0) {
                // this is a light
                Vec3f transmission = 1;
                Vec3f lightDirection = spheres[i].center - phit;
                lightDirection.normalize();
                for (unsigned j = 0; j < spheres.size(); ++j) {
                    if (i != j) {
                        float t0, t1;
                        if (spheres[j].intersect(phit + nhit * bias, lightDirection, t
0, t1)) {
                            transmission = 0;
                            break;
                        }
                    }
                }
                surfaceColor += sphere->surfaceColor * transmission *
                std::max(float(0), nhit.dot(lightDirection)) * spheres[i].emissionColo
r;
            }
        }
    }

    return surfaceColor + sphere->emissionColor;
}

/* Main rendering function. We compute a camera ray for each pixel of the image trace
it and return a color.
```

```
If the ray hits a sphere, we return the color of the sphere at the intersection point,

else we return the background color. */

void render(const std::vector<Sphere> &spheres)
{
    unsigned width = 640, height = 480;
    Vec3f *image = new Vec3f[width * height], *pixel = image;
    float invWidth = 1 / float(width), invHeight = 1 / float(height);
    float fov = 30, aspectratio = width / float(height);
    float angle = tan(M_PI * 0.5 * fov / 180.);
    // Trace rays
    for (unsigned y = 0; y < height; ++y) {
        for (unsigned x = 0; x < width; ++x, ++pixel) {
            float xx = (2 * ((x + 0.5) * invWidth) - 1) * angle * aspectratio;
            float yy = (1 - 2 * ((y + 0.5) * invHeight)) * angle;
            Vec3f raydir(xx, yy, -1);
            raydir.normalize();
            *pixel = trace(Vec3f(0), raydir, spheres, 0);
        }
    }
    // Save result to a PPM image (keep these flags if you compile under Windows)
    std::ofstream ofs("./untitled.ppm", std::ios::out | std::ios::binary);
    ofs << "P6\n" << width << " " << height << "\n255\n";
    for (unsigned i = 0; i < width * height; ++i) {
        ofs << (unsigned char)(std::min(float(1), image[i].x) * 255) <<
               (unsigned char)(std::min(float(1), image[i].y) * 255) <<
               (unsigned char)(std::min(float(1), image[i].z) * 255);
    }
    ofs.close();
    delete [] image;
}

 /* In the main function, we will create the scene which is composed of 5 spheres and
1 light
 (which is also a sphere). Then, once the scene description is complete we render that
 scene,
 by calling the render() function. */

int main(int argc, char **argv)
{
    srand48(13);
    std::vector<Sphere> spheres;
    // position, radius, surface color, reflectivity, transparency, emission color
    spheres.push_back(Sphere(Vec3f( 0.0, -10004, -20), 10000, Vec3f(0.20, 0.20, 0.20),
 0, 0.0));
    spheres.push_back(Sphere(Vec3f( 0.0,      0, -20),     4, Vec3f(1.00, 0.32, 0.36),
 1, 0.5));
    spheres.push_back(Sphere(Vec3f( 5.0,     -1, -15),     2, Vec3f(0.90, 0.76, 0.46),
 1, 0.0));
    spheres.push_back(Sphere(Vec3f( 5.0,      0, -25),     3, Vec3f(0.65, 0.77, 0.97),
 1, 0.0));
    spheres.push_back(Sphere(Vec3f(-5.5,      0, -15),     3, Vec3f(0.90, 0.90, 0.90),
```

```
 1, 0.0));
    // light
    spheres.push_back(Sphere(Vec3f( 0.0,     20, -30),     3, Vec3f(0.00, 0.00, 0.00),
 0, 0.0, Vec3f(3)));
    render(spheres);

    return 0;
}
```

Дуже легке введення в програмування комп'ютерної графіки

> Дана стаття перекладена з англійської мови. Оригінальна стаття знаходиться за цим посиланням.
>
> Ключові слова: 3D, ракурс, стереоскопічний зір, походження, координати, координатна система, 3D-сцена, топологія, модель, сітка, полігон, вершини, ребро, перспективна проекція, viewing frustum, перспектива розподіл, подібні трикутники, екранний простір, нормалізація.

Зрозумій як це працює!

Якщо ви тут, це, ймовірно, тому, що ви хочете навчитися комп'ютерної графіки. Кожен читач може мати різні причини бути тут, але ми всі рухомі тим же бажанням: зрозуміти, як це працює! Scratchapixel був створений, щоб відповісти на це конкретне питання. Тут ви дізнаєтеся, як це працює, і дізнатеся методи, які використовуються для створення CGI (computer generated imaginary), від найпростіших і важливих методів, до більш складних і менш поширених. Можливо, ви любите відеоігри, і ви хотіли б знати, як це працює, як вони зроблені. Може бути, що ви бачили фільм Pixar і цікаво, яка магія за ним. Якщо ви в школі, університеті, вже працюєте в цій галузі (або у відставці), ніколи не пізно, щоб бути зацікавленим в цих темах, щоб вивчити або поліпшити свої знання, і ми завжди потрібуємо такого ресурсу, як Scratchapixel, щоб знайти відповіді на ці питання. Ось чому ми тут. Scratchapixel доступний для всіх. Це уроки для всіх рівнів. Звичайно, це вимагає мінімуму знань в галузі програмування. У той час як ми плануємо написати короткий вступний урок з програмування в найближчому майбутньому, місія Scratchapixel є не про навчання програмування. Проте, в той час як ви будете дізнаватися про реалізацію різних методів, використовуваних для створення CGI, ви також, ймовірно, поліпшите свої навички програмування в процесі і вивчите кілька трюків програмування. Чи вважаєте ви себе новачком або експертом в галузі програмування, ви знайдете тут всі види уроків, адаптованих до вашого рівня. Почніть з простого, з базових програм і прогресуйте звідти. Щодо складності, якщо ви подивитеся на список уроків для кожної категорії, ви побачите математичну етикетку, за якою слідує ряд плюсів (знак «+»). Чим більше плюсів, тим важчі уроки з точки зору математики (три плюси максимум).

Легке введення в програмування комп'ютерної графіки

Ви хочете дізнатися про комп'ютерну графіку. Перше, що вам потрібно знати, що це таке? У другому уроці цього розділу, ви можете знайти визначення комп'ютерної графіки, а також дізнатися про те, як це взагалі працює. Можливо, ви чули про такі терміни, як моделювання, геометрія, анімація, 3D, 2D, цифрові зображення, 3D-перегляд, рендеринг в режимі реального часу, композитінг, але ви не впевнені в тому, що вони означають, і що більш важливо, як вони співвідносяться один з одним. Другий урок цього розділу відповість на ці питання. Звідси, ви повинні дізнатись трохи про програмування CG та отримаєте загальне уявлення про CG і різні інструменти і процеси, що беруть участь в створенні CGI.

Що далі? Наш світ принципово тривимірний. В крайньому випадку, наскільки ми можемо відчути його з нашими органами чуття. Деякі люди хотіли б додати вимір часу. Час грає важливу роль в CGI, але ми повернемося до цього пізніше. Об'єкти з реального світу, тобто тривимірні (three-dimensional). Це факт, ми думаємо, що ми всі можемо погодитися, без ніяких доведень. Однак, що цікаво, це бачення (vision), яким є один з органів чуття, за допомогою якого цей тривимірний світ може бути побачено, принципово двовимірний процес. Ми могли б сказати, що, можливо, образ, створений в нашій свідомості є безрозмірним (ми ще не дуже добре розуміємо як зображення «з'являються» в нашому мозку), але коли ми говоримо про зображення, як правило, це означає для нас плоска поверхня, на якої розмірність об'єктів було скорочено з трьох до двох вимірів (поверхні полотна або поверхні екрану). Єдина причина, чому це зображення на полотні насправді виглядає таким точним для нашого мозку, тому що об'єкти стаю все меншими, чим далі від вас вони знаходяться, ефект, що називається ракурсом (foreshortening). Якщо ви не впевнені в цьому, думайте про зображення як про дзеркальне відображення. Поверхня дзеркала ідеально рівна, і тим не менш, ми не можемо знайти різницю дивлячись на зображення сцени, відбитого від дзеркала і споглядання прямо на сцені: ти не вловлюєш відображення, тільки об'єкт. Це тому, що у нас є два ока, за допомогою яких ми дійсно можемо отримати почуття бачити речі в 3D, те, що ми називаємо стереоскопічний зір (stereoscopic vision). Кожне око дивиться на одну і тій же сцену з трохи іншим кутом, і мозок може використовувати ці два зображення однієї і тієї ж сцени, щоб наблизити відстань і положення об'єктів в 3D-просторі по відношенню один до одного. Однак стереоскопічний зір таким чином обмежений, що ми не можемо виміряти відстань до об'єктів або їх розмір дуже точно (що комп'ютери можуть зробити). Людський зір досить складний і вражаючий результат еволюції, але тим не менш, трюк, і його можна легко обдурити (багато фокусів засновані на цьому). В якійсь мірі, комп'ютерна графіка є засобом, за допомогою якого ми можемо створити образи штучних світів і представити їх в мозок (через бачення), як переживання реальності (те, що ми називаємо фото-реалізм), так само, як дзеркальне відображення. Ця тема є досить поширеним явищем в науковій фантастиці, але технологія не далекі від того, щоб зробити це можливим.

> Це насправді важливо, щоб сказати тут, що в той час як ми, здається, більше зосереджені на процесі створення цих зображень, процес, який ми називаємо рендеринг, комп'ютерна графіка не тільки про створення зображення, а й про розробку методів для моделювання таких речей, як рух рідини, рух м'яких і твердих тіл, знаходження способів анімації об'єктів і аватарів таким чином, щоб їх рух і всі ефекти, що виникають в результаті цього руху точно змоделювались (наприклад, коли ви гуляєте, форма ваших м'язів змінюються і в загальному форма вашого тіла є результатом цих м'язових деформацій), щоб створити реалістичний аватар вам потрібно знайти способи моделювання цих ефектів. Ми також дізнаємося про ці методи на Scratchapixel.

Що ми дізналися досі? Те, що світ є тривимірним, що ми дивимося на нього як на двовимірний, і що, якщо ви можете повторити форму і зовнішній вигляд об'єктів, мозок не може відрізнити різницю дивлячись на ці об'єкти безпосередньо, і дивлячись на зображення цих об'єктів. Комп'ютерна графіка не обмежується створенням фотореалістичних зображень, але в той час простіше створити не фотореалістичні зображення, ніж створення абсолютно фотореалістичних, мета комп'ютерної графіки справжній реалізм (як багато в шляху речі рухатися, ніж вони з'являються).

Все, що нам потрібно зробити зараз, це дізнатися, які правила для створення такого фотореалістичного зображення є, і це те, що ви також дізнаєтеся тут на Scratchapixel.

Опис предметів, що становлять віртуальний світ

Різниця між художником, який насправді малює реальну картину (якщо предмет картини не виходить за межі його / її уяви), і нами при створенні зображення за допомогою комп'ютера, в тому, що ми насправді повинні спочатку якось описати форму (і зовнішній вигляд) об'єктів, що становлять сцену, якщо ми хочемо зарендерити зображення на комп'ютері.

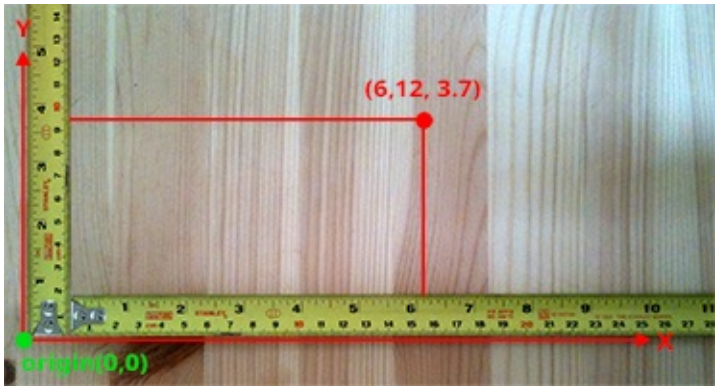


Figure 1: a 2D Cartesian coordinative systems defined by its two axis (x and y) and the origin. This coordinate system can be used as a reference to define the position or coordinates of points within the plane.

Одне з найпростішого і найбільш важливого поняття, що ми вивчили в школі є ідея простору, в якому можуть бути визначені точки. Положення точки, як правило, визначається по відношенню до початку координат. На лінійці, це, як правило, місце позначене номером нуль. Якщо ми використовуємо дві лінійки, одна перпендикулярна до іншої, ми можемо визначити положення точок в двох вимірах. Додайте третю лінійку, перпендикулярно до перших двох, і ви можете визначити положення точок в трьох вимірах. Фактичні числа, що представляють положення точки по відношенню до однієї з трьох лінійок називаються координатами точок.

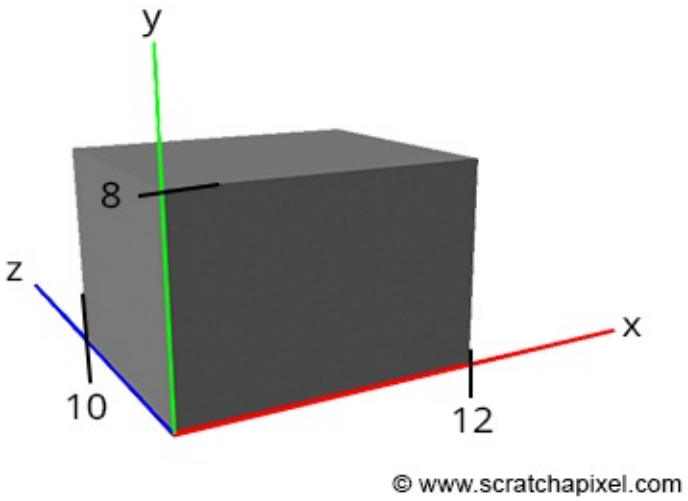


Figure 2: a box can be described by specifying the coordinates of its eight corners in a Cartesian coordinate system.

Ми всі знайомі з концепцією координат, щоб відзначити де ми були відносно деякої опорної точки або лінії (наприклад, Грінвічський меридіан). Тепер ми можемо визначити точки в трьох вимірах. Давайте уявимо, що ви тільки що купили комп'ютер. Цей комп'ютер, ймовірно, приїхав в коробці, і ця коробка має вісім кутів (вибачте, що констатую очевидне). Одним із способів опису цієї коробки, є вимірювання відстані від цих 8 кутів по відношенню до одного з кутів. Цей кут діє як початок нашої системи координат і, очевидно, відстань цього опорного кута по відношенню до самого себе буде у всіх вимірах 0. Однак відстань від опорного кута до семи інших кутів, буде відрізнятися від 0. Давайте зобразимо, що наша коробка має наступні розміри:

```
corner 1: ( 0, 0,  0)
corner 2: (12, 0,  0)
corner 3: (12, 8,  0)
corner 4: ( 0, 8,  0)
corner 5: ( 0, 0, 10)
corner 6: (12, 0, 10)
corner 7: (12, 8, 10)
corner 8: ( 0, 8, 10)
```

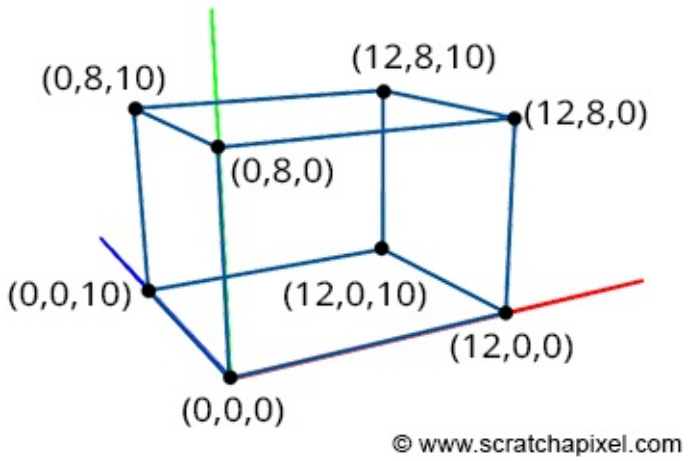


Figure 3: a box can be described by specifying the coordinates of its eight corners in a Cartesian coordinate system.

Перше число представляє ширину, друге число - висоту, і третє число - глибину. Кут 1, як ви можете бачити, є початком, з якого всі кути були виміряні. Все, що вам потрібно зробити тут, це якось написати програму, в якій буде визначено концепцію тривимірної точки, і використовувати її для зберігання координат восьми точок, які ви тільки що виміряли. На C / C++, така програма могла б виглядати наступним чином:

```
typedef float Point[3];
int main()
{
    Point corners[8] = {
        { 0, 0,  0},
        {12, 0,  0},
        {12, 8,  0},
        { 0, 8,  0},
        { 0, 0, 10},
        {12, 0, 10},
        {12, 8, 10},
        { 0, 8, 10},
    };

    return 0;
}
```

Як і в будь-якій мові, завжди є різні способи зробити те ж саме. Ця програма показує один можливий шлях в C / C ++ для визначення поняття точки (лінія 1) і зберігання кутів коробки в пам'яті (в даному прикладі, як масив з восьми точок).

Ви якимось чином створили свою першу програму 3D. Вона ще не виробляє зображення, але ви вже можете зберігати опис 3D-об'єкта в пам'яті. У CG колекція з цих об'єктів називається сцена (сцена також включає в себе поняття камери і світла, але ми будемо говорити про це іншим разом). Як було зазначено раніше, нам не

вистачає двох дуже важливих речей, щоб зробити процес дійсно повним і цікавим. По-перше, щоб насправді відобразити коробку в пам'яті комп'ютера, в ідеалі, ми також потребуємо системи, яка визначає, як ці вісім точок з'єднані одна з одною, щоб відобразити грані (faces) коробки. У CG, це називається топологія (topology) об'єкта (об'єкт також називається моделлю (model)). Ми будемо говорити про це в розділі Геометрія і основи 3D рендера (в уроці по візуалізації трикутників і полігональних сіток). Топологія відноситься до того, як точки, які ми зазвичай називаємо вершини (vertices) з'єднані одна з одною з утворенням граней (або плоских поверхонь). Ці грані також називаються багатокутники (полігони). Коробка буде зроблена з шести граней або шести полігонів і набір багатокутників утворює те, що ми називаємо полігональною сіткою (polygonal mesh) або просто сітка. Друге, що нам не вистачає, це система для створення образу цього блоку. Це вимагає, щоб фактично проектувати кути коробки на уявному полотні, процес який ми називаємо перспективна проекція.

Створення зображення цього віртуального світу

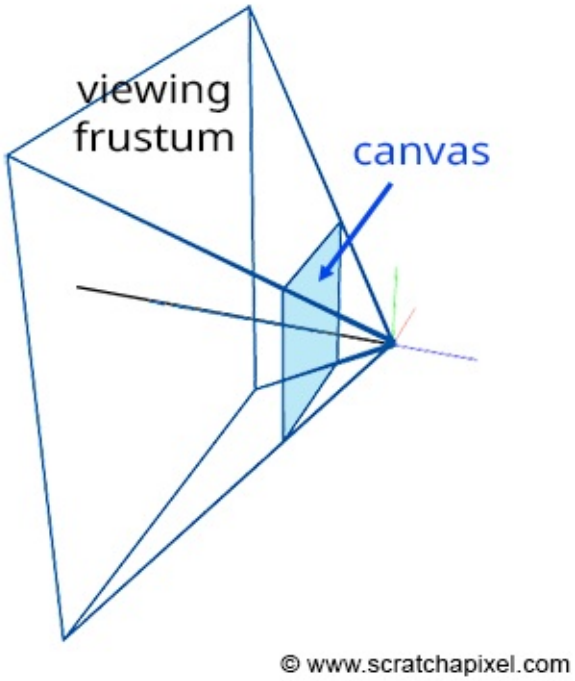


Figure 4: if you connect the corners of the canvas to the eye which by default is aligned with our Cartesian coordinate system, and extend the lines further away into the scene, you get some sort of pyramid which we call a viewing frustum. Any object contained within the frustum (or overlapping it) is visible and will show up on the image.

Процес проектування 3D точки поверхні полотна, насправді включає в себе спеціальну матрицю, яку ще називають перспективна матриця (не хвилюйтеся, якщо ви не знаєте, що таке матриця). Використання матриці для проектування точки не є абсолютно необхідним, але робить речі набагато простішими. Тим не менш, вам насправді не

потрібні математика і матриці, щоб з'ясувати, як це працює. Ви можете побачити зображення, або полотно, як якиусь плоску поверхню, яка розташована на деякій відстані від очей. Пропускаючи чотири лінії, починаючи від очей до кожного з чотирьох кутів полотна і продовжуючи ці лінії далі в світ (наскільки ви можете бачити). Ви отримуєте піраміду, яку ми називаємо viewing frustum (але не усічену (frustrum)).

Viewing frustum визначає якийсь об'єм в 3D просторі і полотно є простою площиною, яка перерізає цей простір перпендикулярно до лінії очей. Помістіть коробку перед полотном. Далі, проведіть лінію від кожного кута коробки до ока і відзначте точку, де лінія перетинає полотно. Знайдіть на полотні точки, що відповідають кожному з дванадцяти ребер коробки, і проведіть лінію між цими точками. Що ти бачиш? Зображення коробки.

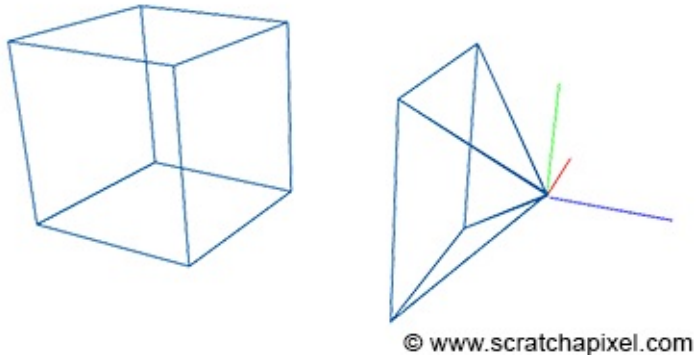


Figure 5: the box is move in front of our camera setup. The coordinates of the box corners are expressed with respect to this Cartesian coordinate system.

Три лінійки, які використовувались для вимірювання координат кутів коробки, ми називаємо системою координат. Це система, в якій точки можуть бути виміряні. Координати всіх точок належать до цієї системи координат. Зверніть увагу, що координата може бути або позитивною або негативною (або нульовою) в залежності від того, чи вона розташована праворуч або ліворуч від початку лінійки (значення 0). У CG, цю систему координат часто називають світовою системою координат, і точка (0,0,0), початок системи координат.

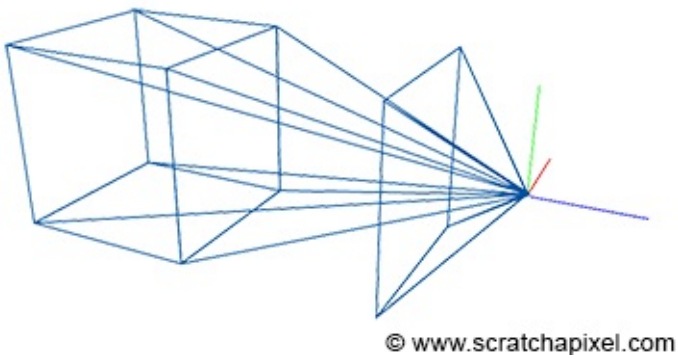


Figure 6: connecting the box corners to the eye.

Давайте перейдемо до вершини піраміди на початку координат і лінії погляду (напрямок погляду) уздовж негативної осі (рисунок 3). Багато графічних додатків використовують цю конфігурацію, як за замовчуванням їх «систему перегляду». Майте на увазі, що вершина піраміди насправді точка, з якої ми будемо дивитися на сцену. Давайте також перемістимо полотно на одну одиницю від нуля. Нарешті, давайте перемістимо вікно на деяку відстань від початку координат, так що вона повністю знаходитиметься в межах об'єму піраміди. Оскільки коробка знаходиться в новому положенні (ми перемістили її), координати восьми кутів змінилися, і ми повинні виміряти їх знову.

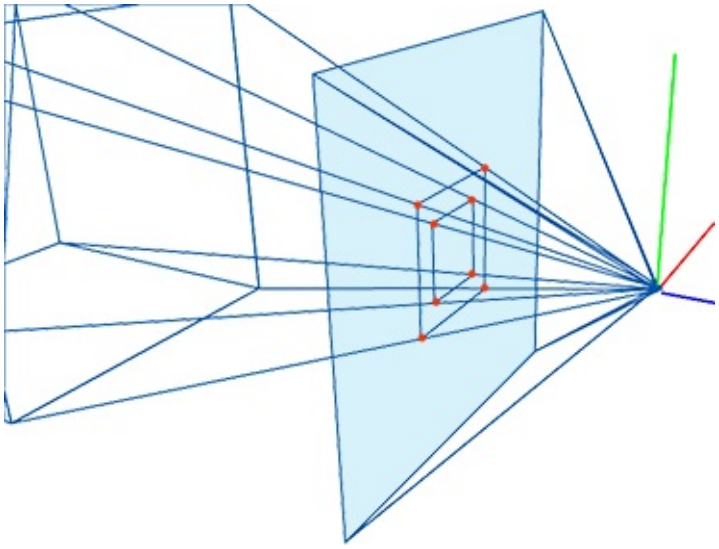


Figure 7: the intersection points between these lines and the canvas are the projection of the box corners onto the canvas. By connecting these points to each other, an wireframe image of the box is created.

Зверніть увагу, так як коробка на лівій стороні від початку лінійки, від якого ми вимірюємо глибину об'єкта, всі координати глибини, які також називаються Z-координати будуть негативними. Чотири з кутів також нижче точки відліку, що використовується для вимірювання висоти об'єкта, і буде мати негативну висоту або y-координату. Нарешті, чотири з кутів будуть зліва від початку лінійки вимірюваної ширини об'єкта: їх ширина, або x-координата також буде негативною. Нові координати кутів короба такі:

- corner 1: (1, -1, -5)
- corner 2: (1, -1, -3)
- corner 3: (1, 1, -5)
- corner 4: (1, 1, -3)
- corner 5: (-1, -1, -5)
- corner 6: (-1, -1, -3)
- corner 7: (-1, 1, -5)
- corner 8: (-1, 1, -3)

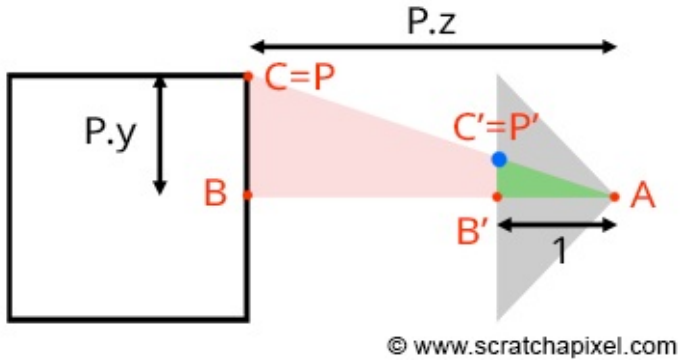


Figure 8: the coordinates of the point P', the projection of P on the canvas can be computed using simple geometry. The rectangle ABC and AB'C' are said to be similar.

Давайте подивимося на нашу інсталяцію з боку і проведемо лінію від одного з кутів до координат (точки огляду). Ми можемо визначити два трикутника: ABC і AB'C. Як ви можете бачити, що ці два трикутника мають однаковий початок (A). Вони є також такими собі копіями один одного, в тому сенсі, що кут, утворений ребрами AB і AC є таким же, як кут, утворений ребрами AB' і AC'. Такі трикутники називаються подібними. Подібні трикутники мають цікаву властивість: співвідношення між їх суміжними і протилежними сторонами однакові. Іншими словами:

```
BC / AB = B'C' / AB'.
```

Оскільки полотно на 1 одиницю від початку координат, ми знаємо, що AB дорівнює 1. Ми також знаємо позицію B і C, які є z (глибина) і y координати (висота) по відношенню до кута. Якщо підставити ці числа в наведеному вище рівнянні, отримаємо:

```
P.y / P.z = P'.y / 1.
```

Як ви можете бачити, проекція з кута в y-координаті на полотні, це не більше, ніж y-координата, розділена на глибину (Z-координата). Це, ймовірно, одне з найпростіших і найбільш фундаментальних відношень в галузі комп'ютерної графіки, відомої як z або перспективне ділення. Точно такий же принцип застосуємо і до x координат. Спроектована точка x координати (x ') є x координатою поділеною на z координату.

Зауважу, проте, що, оскільки z-координата P негативна в нашому прикладі (ми пояснимо, чому це завжди буває на уроці з основ 3D рендерингу, присвяченого перспективній проекції матриці), коли координата x позитивна , x-координата проектованої точки стане негативною (аналогічно, якщо Px негативна, P'.x стане позитивною. Та ж ситуація відбувається з y-координатою). В результаті зображення 3D об'єкта відбивається як по вертикалі, так і по горизонталі, що не є ефектом, який ми хочемо. Таким чином, щоб уникнути цієї проблеми, ми розділимо P.x і P.y координати на P.z зі знаком мінус; це зберігає знак в координатах x і y. Ми нарешті отримаємо:

```
P'.x = P.x / -P.z.
```

```
P'.y = P.y / -P.z.
```

Тепер у нас є метод, щоб обчислити фактичні позиції кутів, як вони з'являються на поверхні полотна. Це двовимірні координати точок, які проектується на полотні. Давайте оновимо нашу програму для обчислення цих координат:

```
typedef float Point[3];
int main()
{
    Point corners[8] = {
         { 1, -1, -5},
         { 1, -1, -3},
         { 1,  1, -5},
         { 1,  1, -3},
         {-1, -1, -5},
         {-1, -1, -3},
         {-1,  1, -5},
         {-1,  1, -3}
    };

    for (int i = 0; i < 8; ++i) {
        // divide the x and y coordinates by the z coordinate to
        // project the point on the canvas
        float x_proj = corners[i][0] / -corners[i][2];
        float y_proj = corners[i][1] / -corners[i][2];
        printf("projected corner: %d x:%f y:%f\n", i, x_proj, y_proj);
    }

    return 0;
}
```

Розмір самого полотна також є довільним. Воно також може бути квадратної або прямокутної форми. У нашому прикладі, ми зробили його в дві одиниці шириною в обох напрямках, що означає, що x і y координати будь-яких точок, що лежать на полотні, містяться в діапазоні від -1 до 1 (зображення 9).

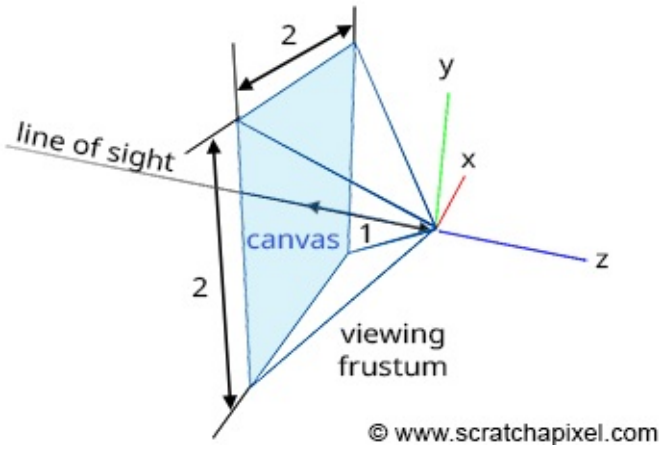


Figure 9: in this example, the canvas is 2 units along the x-axis and 2 units along the y-axis. You can change the dimension of the canvas if you wish. By making it bigger or smaller you will see more or less of the scene.

Питання: що станеться, якщо яка-небудь з проектованих координат точки не в цьому діапазоні, якщо, наприклад, x дорівнює -1.1? Точка буде просто не видною, вона лежить поза межами полотна.

На даний момент ми говоримо, що прогнозована координата точки в просторі екрану (screen space)(простір екрану, де екран і полотно в цьому контексті синоніми). Але ними не так легко маніпулювати, тому що вони можуть бути або негативними або позитивними, і ми не знаємо, як вони насправді відносяться по відношенню до, наприклад, розміру екрану вашого комп'ютера (якщо ми хочемо відобразити ці точки на екрані). З цієї причини ми спочатку нормалізувати їх, що просто означає, що ми перетворюємо їх з будь-якого діапазону в якому вони спочатку були, в діапазон [0,1]. У нашому випадку, так як нам потрібно відобразити координати від -1,1 до 0,1, ми можемо просто написати:

```
float x_proj_remap = (1 + x_proj) / 2;
float y_proj_remap = (1 + y_proj) / 2;
```

Координати, спроектовані в точках, не в діапазоні 0,1. Такі координати називаються визначені в NDC просторі, що означає нормовані реальні координати (Normalized Device Coordinates). Це зручно, тому що незалежно від початкового розміру полотна (або екрана), який може відрізнятися в залежності від налаштувань, які ви використовували, тепер ми маємо всі точки координат, визначені в загальному просторі. Термін нормалізація є дуже поширеним явищем. Це означає, що ви якимось чином перепризначаєте значення з будь-якого діапазону в якому вони були спочатку, в діапазон [0,1]. Нарешті, ми зазвичай вважаємо за краще, щоб визначити координати точки відносно розмірів кінцевого зображення, які, як ви знаєте чи ні, визначаються в пікселях. Цифрове зображення є нічим іншим, як двовимірний масив пікселів (як ваш екран комп'ютера).

Ти знаєш яким є розширення твого екрану в пікселях?

512x512 зображення являє собою цифрове зображення, що має 512 рядків по 512 пікселів, ви вважаєте за краще бачити його іншим чином, 512 стовпців по 512 вертикально орієнтованих пікселів. Так як наші координати вже нормалізовані, все, що нам потрібно зробити, щоб виразити їх в пікселях, це помножити ці NDC координати на розмір зображення (512). Так як наше полотно є квадратом, ми будемо також використовувати квадратне зображення:

```
#include <cstdlib>
#include <cstdio>

typedef float Point[3];

int main()
{
    Point corners[8] = {
         { 1, -1, -5},
         { 1, -1, -3},
         { 1,  1, -5},
         { 1,  1, -3},
         {-1, -1, -5},
         {-1, -1, -3},
         {-1,  1, -5},
         {-1,  1, -3}
    };
const unsigned int image_width = 512, image_height = 512;

    for (int i = 0; i < 8; ++i) {
        // divide the x and y coordinates by the z coordinate to
        // project the point on the canvas
        float x_proj = corners[i][0] / -corners[i][2];
        float y_proj = corners[i][1] / -corners[i][2];
        float x_proj_remap = (1 + x_proj) / 2;
        float y_proj_remap = (1 + y_proj) / 2;
        float x_proj_pix = x_proj_remap * image_width;
        float y_proj_pix = y_proj_remap * image_height;
        printf("corner: %d x:%f y:%f\n", i, x_proj_pix, y_proj_pix);
    }

    return 0;
}
```

Отримані координати визначаються в растровому просторі (XX століття, що це растр значить, поясніть, будь ласка). Наша програма як і раніше обмежена, оскільки вона не створює зображення коробки, але якщо ви скомпілювати і запустити її за допомогою наступних команд (копіювати / вставити код в файл і зберегти його як box.cpp):

```
c++ box.cpp
./a.out
corner: 0 x:307.200012 y:204.800003
corner: 1 x:341.333344 y:170.666656
corner: 2 x:307.200012 y:307.200012
corner: 3 x:341.333344 y:341.333344
corner: 4 x:204.800003 y:204.800003
corner: 5 x:170.666656 y:170.666656
corner: 6 x:204.800003 y:307.200012
corner: 7 x:170.666656 y:341.333344
```

Ви можете використовувати програму для малювання для створення зображення (встановіть його розмір до 512x512), а також додайте точки в піксельних координатах, які обчислені вашою програмою. Потім з'єднайте точки, щоб сформувати краЇ коробки, і ви отримаєте реальне зображення коробки (як показано у відео нижче). Піксельні координати є цілими числами, так що вам потрібно буде округляти числа, задані програмою.

Video

Що ми вже вивчили?

1. В першу чергу нам необхідно описати тривимірні об'єкти за допомогою таких речей, як вершини і топології (інформація про те, як ці вершини з'єднані одна з одною з утворенням багатокутника або особи), перш ніж ми можемо отримати зображення 3D сцени (сцена - зібрання об'єктів).
2. Візуалізація (рендеринг) - це процес, за допомогою якого створюється зображення 3D сцени. Незалежно від того, який метод використовується для створення 3D-моделі (їх досить багато), рендеринг є необхідним кроком на шляху щоб «побачити» будь-який віртуальний 3D світ.
3. З цієї простої вправи повинно бути абсолютно очевидно, що математика (більш ніж програмування) відіграють важливу роль в процесі створення зображення за допомогою комп'ютера. Насправді комп'ютер є лише інструментом, який використовується для прискорення обчислень, але правила, що використовуються для створення цього зображення - це чиста математика. Геометрія грає особливо важливу роль в цьому процесі, зокрема, для роботи з об'єктами перетворення (масштаб, поворот, переміщення), і також для забезпечення вирішення таких проблем, як обчислення кутів між лініями, або з'ясування перетин між лінією і іншими простими об'єктами (площина, сфера і т.д.).
4. Як висновок, комп'ютерна графіка це в основному математика застосована в комп'ютерній програмі, метою якої є для створення зображення (фото-реалістичні чи ні) з найшвидшою можливою швидкістю (і з точностю, на яку комп'ютери здатні).
5. Моделювання включає в себе всі техніки, використовувані для створення 3D-моделей. Методи моделювання будуть розглянуті в розділах Геометрія / Моделювання.
6. У той час як статичні моделі гарні, але їх також можна оживити протягом певного часу. Це означає, що зображення моделі на кожному часовому кроці може бути відрендерене (можна, наприклад, перемістити, обертати або масштабувати коробку потрохи на кожному наступному зображенні чи анімувати координати кутів або застосовувати матрицю перетворення моделі). Більш просунуті технології анімації можуть бути використані для імітації деформації шкіри на кістках і м'язах. Але всі ці методи мають в загальному ту ж геометрію (грані стають моделями), що

деформувалася з часом. Тому час, як це було запропоновано у вступі важливий в CGI також. Перевірте розділ анімації, щоб дізнатися про цю тему.

7. Одне конкретне поле перекриває як анімацію так і моделювання. Воно включає в себе всі методи, використовувані для моделювання руху об'єктів в реалістичній манері. Дуже велика галузь комп'ютерної графіки присвячена моделюванню руху рідини (вода, вогонь, дим), тканини, волосся і т.д. Закони фізики застосовуються до 3D-моделі, щоб змусити їх рухатися, деформуватися або ламатися, як це є в реальному світі. Фізика моделювання, як правило, потребує дуже великих обчислювальних витрат, але вони також можуть працювати в режимі реального часу (все залежить від складності сцени, яку ви симулюєте).
8. Рендеринг також обчислювально затратне завдання. Як затратно залежить від того, наскільки багато геометрії вкладено у вашу сцену і наскільки фото-реалістичне ви хочете остаточне зображення. У рендерингу, ми розрізняємо два режими, офлайн режим і режим реального часу рендерингу. У режимі реального часу використовується (насправді це вимога) для відеоігор, в яких контент 3D-сцен повинен бути відрендерений (візуалізований) щонайменше 30 кадрів в секунду (як правило, 60 кадрів в секунду, вважається стандарт). Більшість рендерингу в реальному часі виконується на GPU - процесор, який спеціально розроблений для візуалізації 3D сцен на максимально можливій швидкості. Методи візуалізації в реальному часі будуть обговорюватися в розділі Realtime. Offline рендеринг зазвичай використовується у виробництві CGI для фільмів, де в режим реального часу не є обов'язковою вимогою (зображення попередньо обчислюється і зберігається перед відображенням на 24 або 30 або 60 кадрів в секунду). Це може зайняти від декількох секунд до декількох годин, перш ніж одине єдине зображення буде завершене, але воно зазвичай обробляє набагато більше геометрії і створює зображення більш високої якості, ніж рендеринг в реальному часі. Проте, режим реального часу та офлайн режим рендеринга мають тенденцію перекривати один одного все більше і більше в ці дні, у відео-іграх збільшується кількість геометрії, яку вони можуть обробляти в хорошій якості, а тому офлайн двигуни рендерингу намагаються скористатися останніми досягненнями в області технологій CPU, щоб значно поліпшити свої характеристки. Офф-лайн рендеринг є головною темою кількох розділів на Scratchapixel: Основи 3D-рендеринга, методи, специфічні для трасування променів, легкі транспортні алгоритмів, затінення і процедурне текстурування. Ми радимо вам прочитати уроки першого розділу в хронологічному порядку.

Де я повинен починати?

Ми сподіваємося, що простий приклад з коробкою зачепив тебе, але головна мета цього введення є, щоб підкреслити ту роль, яку відіграє геометрія в комп'ютерній графіці. Звичайно, це не тільки про геометрію, але багато проблем може бути вирішено за допомогою геометрії. Більшість книг з комп'ютерної графіки починаються з главою про геометрію, яка завжди трохи лякає, тому що здається, що вам потрібно багато вчитися, перш ніж ви зможете робити прикольні речі. Тим не менш, ми дійсно рекомендуємо вам прочитати урок по геометрії, перш ніж що-небудь починати. Ми будемо говорити і дізнаємося про точки, і про поняття вектора і нормаль. Ми дізнаємося про системи координат і, що більш важливо про поняття матриці. Матриці широко використовуються для обробки перетворень, таких як обертання, масштабування або зміщення. Ці поняття використовуються повсюдно у всій літературі про комп'ютерну графіку, тому вам необхідно вивчити їх першими.

Багато книг про CG не забезпечують гарне введення до геометрії, можливо, тому що автори припускають, що читачі вже знають про це або, що краще читати книги, присвячені цій конкретній темі. Наш урок по геометрії дуже різноманітний. Він дуже ретельний і пояснює все в дуже простих словах (в тому числі і те, про що вам розкажуть тільки люди прошарені в цій темі). Почни з читання цього уроку.

Що я повинен читати далі?

Це зазвичай простіше і веселіше, щоб почати навчатись програмування комп'ютерної графіки з рендерингу. Один з можливих способів для вас, щоб пройти через зміст цього сайту, щоб почати читати уроки з розділу Основи 3D Рендерингу в хронологічному порядку. Для того, щоб зрозуміти зміст уроку вам можуть знадобитися попередні знання про деякі інші методи. На початку кожного уроку ви знайдете список інших уроків, який містить все, що потрібно знати для того, щоб зрозуміти зміст уроку, який ви збираєтеся почати (ми називаємо їх передумова). Використовуйте цей список, щоб направляти вас через вміст веб-сайту і що ще більш важливо отримати основи, необхідні для того, щоб прогресувати в навчанні.

Візуалізація зображення 3D сцени: огляд

Даний розділ є перекладом статті - Rendering an Image of a 3D Scene: an Overview.

Keywords: ray tracing, rasterization, rasterisation, pixel, discretisation, raster, aliasing, vector graphics, triangle, triangulation, smooth shading, sampling, tessellation, polygonal mesh, NURBS, subdivision surface, voxel, implicit surface, meatballs, bloodies, rendering primitive, perspective projection, foreshortening effect, orthographic projection, perspective divide, hidden surface elimination, hidden surface determination, hidden surface removal, occlusion culling, visible surface determination, shading, absorption, reflection, normal, scattering, shader, light transport, forward tracing, backward tracing, indirect diffuse, wireframe, frustum, screen space, raster space, depth sorting algorithm, acceleration structure, law of reflection, reflected direction, incident direction, Snell law, Fresnel equation, roughness, specular reflection, subsurface scattering, indirect diffuse, indirect specular, caustics, soft shadows, global illumination, radiosity, Monte Carlo ray tracing, primary ray, secondary ray, shadow ray, unidirectional path tracing, Appel, Whitted, photon mapping, albedo, energy conservation, physically plausible rendering, cosine law.

В цьому розділі будуть розглядатись наступні питання:

1. **Все починається з комп'ютера і комп'ютерного екрана (It All Starts with a Computer and a Computer Screen)**
2. **А тоді йде 3D сцена (And It Follows with a 3D Scene)**
3. **Огляд процесу рендеринга: видимість і затінення (An Overview of the Rendering Process: Visibility and Shading)**
4. **Перспективна проекція (Perspective Projection)**
5. **Проблема видимості (The Visibility Problem)**
6. **Симулятор світла (A Light Simulator)**
7. **Передача світла (Light Transport)**
8. **Затінення (Shading)**
9. **Підсумок і інші міркування про рендеринг (Summary and Other Considerations About Rendering)**

Все починається з комп'ютера і екрана комп'ютера

Рейтрейсинг: рендеринг трикутника (Ray Tracing: Rendering a Triangle)

Даний розділ є перекладом статті - Ray Tracing: Rendering a Triangle

Зміст

Ключові слова: перетин променя і трикутника, трасування променів, барицентричні координати, відбір поверхонь, змішаний добуток, визначник, алгоритм Moller-Trumbore.

Keywords: ray-triangle intersection test, ray-tracing, barycentric coordinates, back-face culling, scalar triple product, determinant, Möller-Trumbore algorithm.

Тлумачення

back-face culling - відбір поверхонь

Відбір поверхонь - процес ігнорування або видалення полігонів, межі яких розташовані в стороні від точки огляду. Це дозволяє збільшити швидкість обрахунку тривимірних моделей, оскільки програма звертає увагу тільки на багатокутники, розміщені на передній "стороні" об'єктів (тобто безпосередньо перед камерою).

Чому трикутники корисні?

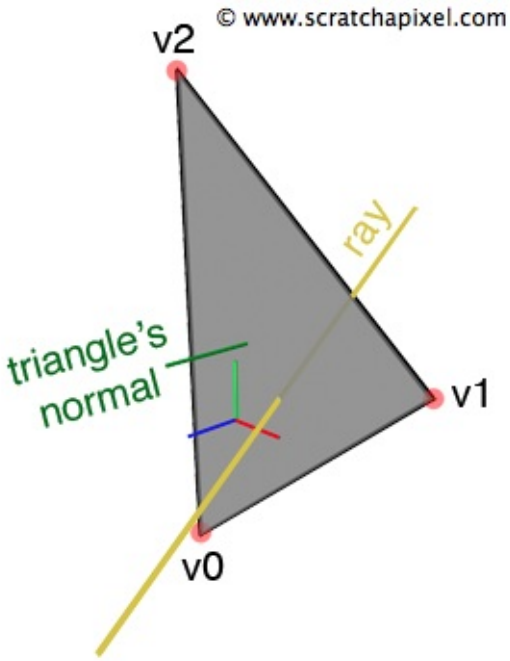
У теорії, знаходження перетинів променів і трикутників не так вже й складно. Те, що робить перетин променя і трикутника складним є безліч різних випадків, які ми повинні враховувати. Для того, щоб написати надійну програму, яка обробляє всі ці випадки швидко і ефективно може бути досить складно. З цієї причини ця тема була предметом багатьох досліджень і дискусій в товаристві CG. Крім того, це один з найважливіших і критичних процесів в рейтрейсері. У цьому уроці ми будемо дотримуватися основних методів; основи, на яких будуються більш складні методи.

Геометричні примітиви

Геометричний примітив є представлення 3D-об'єкта в програмі візуалізації. Наприклад, якщо ми хочемо показати сферу, ми можемо зробити це за допомогою положення її центру та її радіусу, як описано в попередньому уроці. Складні об'єкти, подібним чином, можна змоделювати за допомогою більш складних геометричних примітивів, таких як багатокутники, NURBS або Безьє пластирі, підрозділи поверхонь і т.д. Кожен з них є корисним для зображення певних типів 3D об'єктів. Наприклад, NURBS хороша для об'єктів з гладкими поверхнями. Полігони, з іншого боку, можуть бути використані для геометричних фігур, таких як будівлі. **Трикутники не зовсім тип геометрії їх власної (Triangles are not really a geometry type of their own).** Швидше, вони є підмножиною багатокутника примітивного типу. Ми побачимо в наступному уроці, що полігональний об'єкт, однак, легко конвертовується в трикутники.

Чому ми любимо трикутники так сильно?

Обчислення перетину променя з примітивом, таким як сфера, не є складним. Однак, так як важко моделювати більшість 3D-об'єктів з одними сферами, необхідно використовувати деякі інші типи примітивів для представлення більш складних об'єктів (об'єктів довільної форми). Ніщо не заважає нам використовувати полігональні сітки або NURBS поверхні з нашим візуалізатором, але обчислення перетину променя з цим примітивом може бути важкими і повільними. З іншого боку, обчислення перетину променя з трикутником є досить простою операцією, яка також може бути добре оптимізована. Це може бути не так швидко, як обчислення перетину між променем і сферами, але, в крайньому випадку, це буде краще, ніж перетин променя з поверхнею NURBS. Це компроміс, який ми готові зробити.



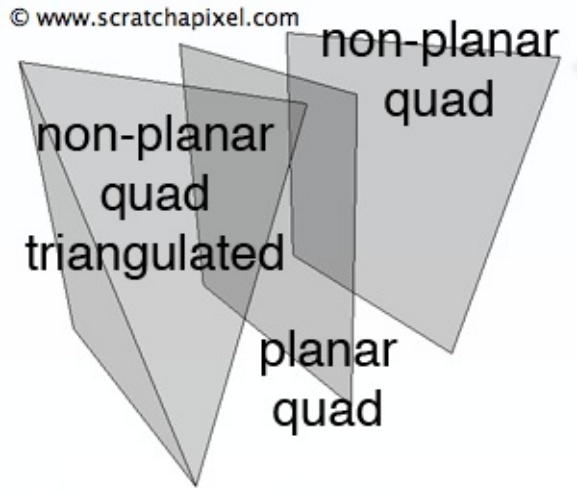
Малюнок 1: Геометрія трикутника. Трикутник визначається трьома вершинами, які визначають площину. Нормаль перпендикулярна до цієї площини. Промінь (жовтий колір) перетинає цей трикутник.

Замість того щоб працювати зі складними примітивами, такими як NURBS або згладжуванням Безьє, ми можемо перетворити будь-який об'єкт у трикутну сітку і обчислити перетин променя з кожним трикутником в цій сітці. Роблячи таким чином, ми перетворюємо перетин об'єктів променем до простої, але переважно швидкої рутини. Перетворення патча Безьє в трикутну сітку набагато простіше, ніж завдання обчислення декількох перетинів променів з патчами Безьє. У нашому коді, як наслідок цієї стратегії є те, що кожен раз, коли ми реалізуємо новий геометричний тип примітиву, ми повинні написати програму, яка перетворює цей тип в трикутну сітку, а не написати підпрограму для обчислення перетину променя з цим примітивном. Всі об'єкти сцени буде внутрішньо представлені у вигляді трикутних сіток. Більшість рейтрейсерів (принаймні ті, що виробляються) використовують цей підхід.

Що таке трикутник?

Трикутник визначається трьома вершинами (або точками), положення яких визначено в тривимірному просторі (рисунок 1). Одна точка може представляти тільки розташування в просторі. З двома точками, ми можемо визначити лінію. З трьома крапками, ми можемо визначити поверхню (і площину). За побудовою, трикутник компланарний; три вершини визначають площину, і всі три вершини лежать в одній площині. Це не обов'язково так для багатокутників з більш ніж трьома вершинами; чотири точки визначають чотирикутник навіть тоді, коли ці чотири точки знаходяться не в одній площині (що цілком можливо). Проте, ми можемо перетворити чотирикутник на

два планарних трикутники, як показано на малюнку 2. Цей метод може бути застосований до полігону, який має довільне число вершин: цей процес називається тріангуляція (ми будемо вивчати про рендеринг полігональних сіток в наступному уроці).



Малюнок 2: чотирикутник не обов'язково копланарний, чотири його вершини не лежать в одній площині (як показано на малюнку). Для того, щоб вирішити цю проблему, ми можемо тріангулювати чотирикутник, як ми знаємо, що трикутники обов'язково копланарні (зліва, правий чотирикутник перетворюється в два трикутника).

Як ми обчислюємо перетин променя з трикутником?

За кілька десятиліть, кілька алгоритмів, які були розроблені, щоб обчислити перетин між променем і трикутником (дослідження на цю тему ще тривають і документи продовжують публікуватися на регулярній основі). У цьому уроці ми будемо вивчати два з цих методів. Перший метод вимагає для реалізації тільки базової логіки й елементарної лінійної алгебри. Другий метод, розглядається як один з найшвидших алгоритмів перетину променем трикутника, який був запропонований Moller-Trumbore в 1997 році в статті під назвою «Швидке знаходження перетинання променя з трикутником». Незважаючи на те, що це вимагає більш поглибленого розуміння лінійної алгебри, ми зробимо все можливе, щоб пройти через це. Ми можемо розкласти тест на перетин променя трикутником через два кроки:

1. Чи перетинає промінь площину, визначену трикутником?
2. Якщо так, то точка перетину лежить всередині трикутника?

Давайте подивимося, як ми можемо відповісти на ці два питання.

Додаткові матеріали

Переклад статті Möller-Trumbore - "Fast, minimum storage ray-triangle intersection" російською мовою. Посилання на статтю тут.

Геометрія трикутника (Geometry of a Triangle)

Базові математичні іструменти (Basic Maths Tools)

Перш ніж ми дізнаємося про методи, які знаходять перетин променя і трикутника, давайте спочатку подивимося на деякі з інструментів, які ми будемо використовувати, щоб вирішити цю проблему. Як ми вже говорили раніше, вершини трикутника визначають площину. Кожна площина має нормаль, яка може розглядатись як пряма, перпендикулярна до поверхні площини. З огляду на те, що ми знаємо координати вершин трикутника (A, B і C), ми можемо обчислити вектори AB і AC (які просто лініями, що йдуть від A до B і від A до C). Щоб зробити це, ми просто віднімаємо B від A і C від A.

Для того, щоб обчислити нормаль площини (що, за логікою речей, теж саме, що і нормаль трикутника, так як трикутник лежить в площині), ми просто обчислимо векторний добуток (cross product) AB і AC. Пам'ятайте, що ці два вектори лежать в одній площині, так як вони з'єднують вершини трикутника. Вектор в результаті цього векторного добутку (позначається N) є нормаллю, яку ми шукаємо (рисунок 1). Ці операції наведені в даному прикладі псевдокоду:

```
Vec3f v0(-1, -1, 0), v1(1, -1, 0), v2(0, 1, 0);
Vec3f A = v1 - v0; // edge 0
Vec3f B = v2 - v0; // edge 1
Vec3f C = cross(A, B); // this is the triangle's normal
normalize(C);
```

```
A=v1−v0=(1−−1,−1−−1,0)=(2,0,0)
B=v2−v0=(0−−1,1−−1,0)=(1,2,0)
C=A×B=
Cx=Ay∗Bz−Az∗By=0
Cy=Az∗Bx−Ax∗Bz=0
Cz=Ax∗By−Ay∗Bx=2∗2−0∗1=4
```

Якщо нормалізувати C ми отримуємо вектор (0, 0, 1), який, як ви можете бачити, паралельний позитивній z-осі, що є тим, що ми очікуємо, так як вершини v0, v1 і v2 лежать в площині xy.

Направленість системи координат (Coordinate System Handedness)

Про порядок, в якому вершини трикутника впливають на орієнтацію нормалі поверхні. V0, V1, і V2, створені в порядку проти годинникової стрілки (CCW - counter-clockwise order), створюють нормаль, яку позначаємо - N. Якщо вершини створюються в порядку за годинниковою стрілкою (CW - clockwise order), нормаль, яка обчислюється з векторного добутку двох сторін (A = V1-V0 і B = V2-V0) буде потім вказувати в протилежному напрямку по відношенню до першої обчисленої N.

При створенні трикутника в Maya (яка використовує правосторонню систему координат), якщо ви повертаєте проти годинникової стрілки, відповідно до створених вами вершин трикутника, то вісь - x (V0V1) стає A, вісь - y (V0V2) стає B, а C, яка є векторниим добутоком AxB, паралельна осі - z (рисунок 2). Але якщо створити вершини за годинниковою стрілкою, то C буде вказувати уздовж негативної осі - z (створюючи вершини за годинниковою стрілкою і обчислюючи N від векторного добутку AxB є тим же, що і обчислення векторного добутку BxA, коли вершини створюються проти годинникової стрілки з A = V1-V0 і B = V2-V0).

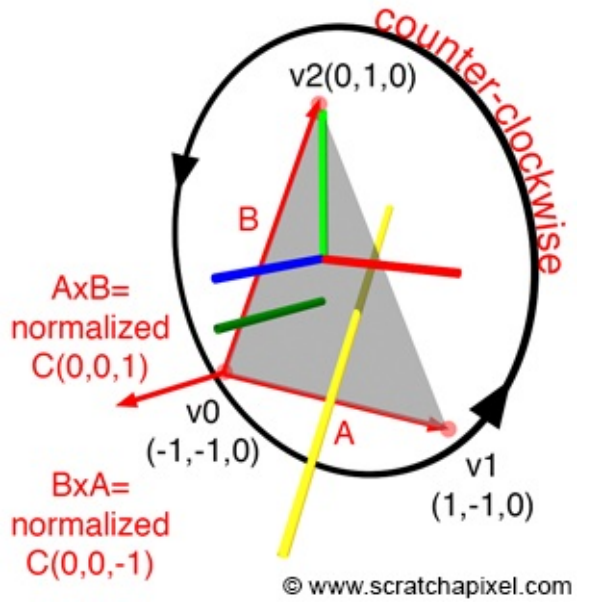


Рисунок 1: вектори A і B можуть бути обчислені з V1 - V0 і V2 - V0 відповідно. Нормаль трикутника, або вектор C є векторним добутком A і B. Слід зазначити, що порядок, в якому були створені вершини визначає напрямок нормалі (в цьому прикладі вершина була створена проти годинникової стрілки).

Тепер уявіть, що ви створюєте цей трикутник в лівосторонній системі координат від трьох вершин, які мають одні і ті ж координати, що й і в правій системі координат, до прикладу (V0 = (0,0,0), V1 = (1,0, 0) і V2 = (0,1,0)). Слідом за тими ж кроками, ми можемо створити вектори A (V1-V0) і B (V2-V0) і обчислити C (від AxB). Результат також дорівнює (0,0,1), так як із прикладом правосторонньої системи координат (координати вершини збігаються, отже, вектори A і B є такими ж, як і результаті векторного добутку AxB). Пам'ятайте, що за угодою, вісь - z в лівобічній системі координат вказує в

напрямку екрана (коли вісь x спрямована вправо). Знову ж таки, все це узгоджується з поясненнями, які ми дали досі про направленість координатних систем. Результат векторного добутку, як ми згадували в уроці з Геометрії, є анти-комутативною операцією.

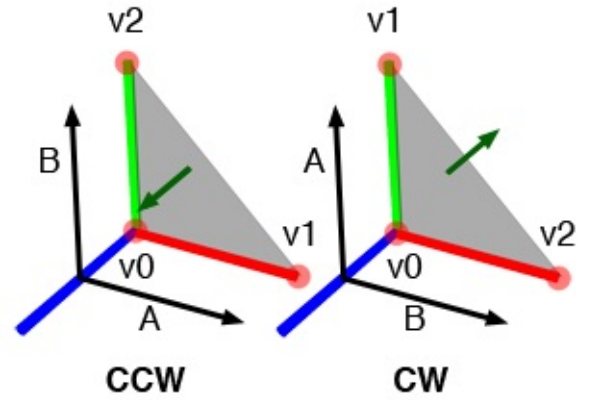


Рисунок 2: створення вершини в CCW або CW змінює напрямок нормалі.

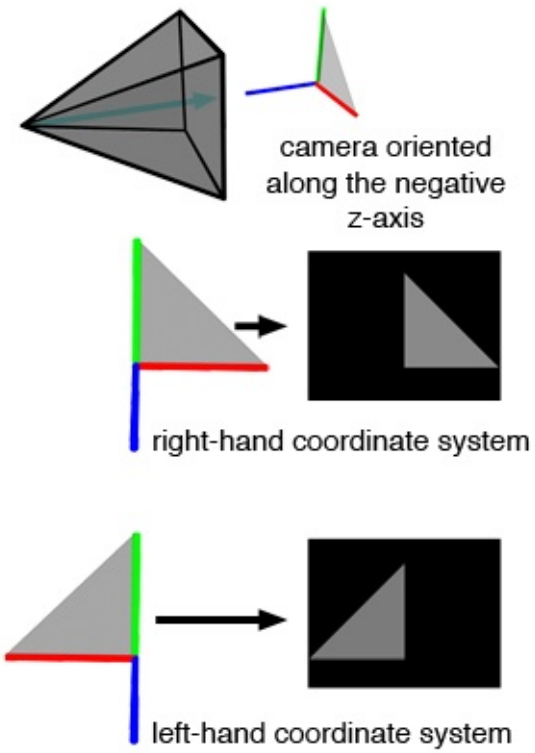


Рисунок 3: трикутники, створені в лівобічній або правобічній системі координат не виглядають однаково, коли рендеряться з однієї і тієї ж камери (вказуючи по негативній вісі - z).

Тепер наступний крок, щоб прорендерити кожен з цих трикутників з камери, яка вказує на негативну вісь - z. Результати цих випробувань показані на рисунку 4. Отримані зображення не виглядають однаково, навіть якщо вершини трикутника мають одні і ті ж координати, що не те, що ми хочемо. Вибір направленості системи координат, не

повинен змінити шляху геометрії, який бачимо з точки зору камери. Людина, яка робить зображення, ймовірно, хоче побачити те ж зображення одного і того ж трикутника, незалежно від використовуваної направленості. Навіть, якщо це не пов'язано безпосередньо з темою перетину променем трикутника, це важлива проблема, яку ви повинні знати. Як правило, спосіб, яким ми вирішимо це через дзеркальне відображення камери уздовж негативної осі (як через її положення, так і через орієнтацію), як показано на рисунку 4.

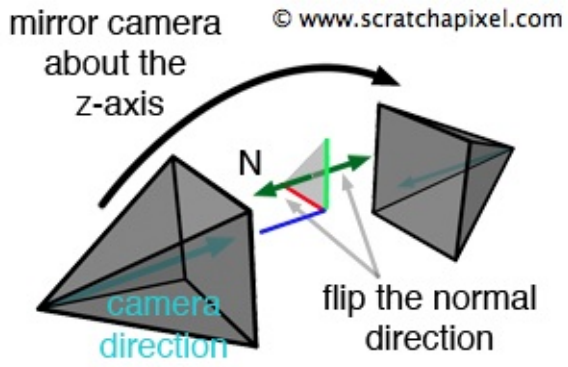


Рисунок 4: щоб отримати те ж саме зображення трикутника, ми повинні віддзеркалювати камеру щодо осі - z і перевертати нормаль трикутника.

Це працює, однак ви можете побачити, що тепер напрямок камери і напрямок нормалі змінилися в порівнянні з тим, що було до того як камера віддзеркалювалась. На рисунку 3, напрямок камери і нормалі трикутника вказують в протилежних напрямках. Але на рисунку 4, тепер вони вказують в одному напрямку. Ці два вектора (нормаль трикутника і напрямок камери) відіграють важливу роль в затіненні, і тому важливо, щоб ми не змінювали їх напрям по відношенню один до одного. Тому вирішення проблеми перевернутого трикутника не тільки в віддзеркалюванні камери, але і в переверненні напрямоку нормалі. Пам'ятайте, що це необхідно, тільки якщо ви змоделювали трикутник в додатку моделювання за допомогою лівобічної системи координат, але зробіть це в рендері за допомогою правобічної системи (або навпаки, якщо ви створюєте геометрію координат Maya і зробити цю геометрію в RenderMan, який використовує право- і лівосторонню системи координат, відповідно). ~~*(Remember though that this is only necessary if you modelled the triangle in a modeling application using a left-hand coordinate system but render it in a renderer using a right-hand coordinate system (or vice-versa which is the case if you generate geometry in Maya and render this geometry in RenderMan which is using a right- and left-hand coordinate system respectively).)*~~

> Поєднання порядку і напрямку, в яких вказані вершини, називається поворотом (winding)

Тепер у нас є всі інструменти під рукою, щоб вирішити тест перетину променем трикутника за допомогою простої геометрії.