4

Тени и отражения

Наш квест по реалистичному рендерингу сцены продолжается. В прошлой главе мы смоделировали способ взаимодействия лучей света с поверхностями. В этой главе мы смоделируем два аспекта процесса взаимодействия света со сценой: объекты, отбрасывающие тени и отражающиеся в других объектах.

Тени

Там, где присутствуют свет и объекты, есть и тени. Все это у нас есть, так где же тени?

Принцип формирования теней

Начнем с основного вопроса: откуда вообще берутся тени? Они *возникают*, когда лучи света не могут достичь объекта из-за встреченного на пути препятствия.

Ранее мы рассмотрели только локальные взаимодействия между источником света и поверхностью, проигнорировав все остальные процессы сцены. Чтобы возникли тени, нужно посмотреть глобальнее и проанализировать взаимодействие между источником света, желаемой поверхностью и другими объектами сцены.

Воплотить это несложно. Только следует помнить, что «если между точкой и источником света есть объект, то освещение, исходящее от этого источника, добавлять не нужно».

На рис. 4.1 есть два случая, которые нам необходимо будет различать.

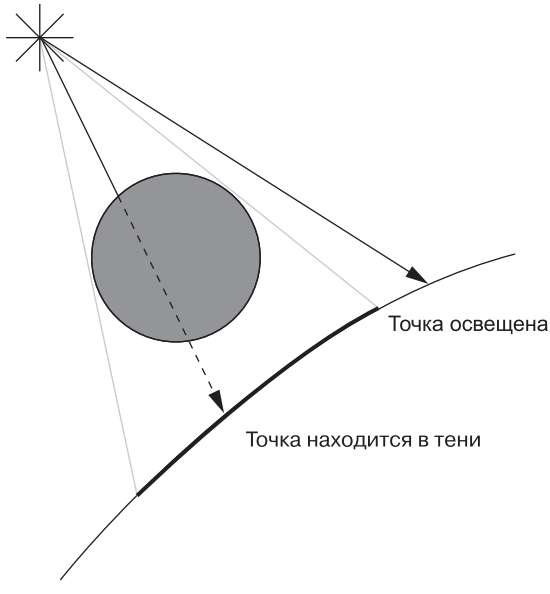

Рис. 4.1. Если между точкой и источником света есть объект, тень всегда будет отбрасываться

У нас уже есть все нужные инструменты. Начнем с направленного света. Нам известна P — интересующая нас точка. Нам известен $\vec{L}$ — часть определения света. Зная P и $\vec{L}$, можно определить луч $(O+t\vec{L})$, проходящий из этой точки поверхности к бесконечно удаленному источнику света. Этот луч с чем-то пересекается? Если ответ отрицательный, то между точкой и источником света ничего нет. Поэтому мы вычисляем освещение от этого источника, как и раньше. Если же пересекается, то точка будет в тени и освещение от этого источника нужно игнорировать.

Мы уже умеем вычислять ближайшее пересечение между лучом и сферой. Для этого у нас есть `TraceRay`, с помощью которой мы трассируем исходящие из камеры лучи. Можно повторно использовать большую ее часть для вычисления ближайшего пересечения луча света с остальной частью сцены.

Хотя при этом параметры функции будут немного другие.

- Теперь луч начинается не от камеры, а от точки P.
- Направление луча не $(V - O)$, а $\vec{L}$.

- Мы не хотим, чтобы объекты *позади P* отбрасывали тени на эту точку, значит, нам нужно $t_{min} = 0$.
- Мы имеем дело с бесконечно удаленными направленными источниками света, поэтому даже очень далекий объект должен все равно отбрасывать тень на P. Значит, $t_{max} = +\infty$.

На рис. 4.2 есть две точки: P_0 и P_1. При трассировке луча, исходящего из P_0 к источнику света, пересечений с объектами нет. Значит, свет может достичь P_0 и тени на ней не будет. Если же с P_1 между лучом и сферой мы находим два пересечения с $t > 0$ (пересечение находится между поверхностью и источником света), точка находится в тени.

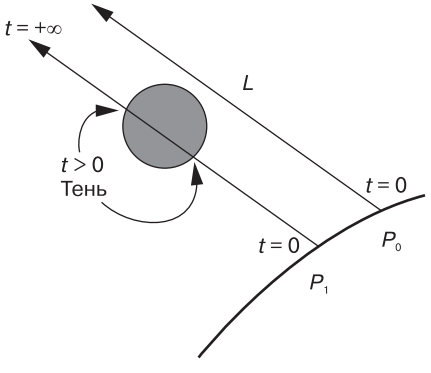

Рис. 4.2. Сфера отбрасывает тень на P_1, но не на P_0

Точечные источники можно рассматривать так же, но есть два исключения. Во-первых, $\vec{L}$ непостоянен, но мы уже умеем вычислять его из P и позиции света. Во-вторых, мы не хотим, чтобы объекты дальше источника света смогли отбрасывать тени на P. Значит, в этом случае нам нужно $t_{max} = 1$, чтобы при достижении источника света луч останавливался.

На рис. 4.3 есть все эти ситуации. Когда мы испускаем луч из P_0 к L_0, то находим пересечения с небольшой сферой. Но для них $t > 1$, то есть они не находятся между источником света и P_0. Значит, мы их игнорируем, поэтому P_0 находится не в тени. С другой стороны, луч из P_1 с направлением L_1 пересекает большую сферу с $0 < t < 1$, и в результате она отбрасывает тень на P_1.

При этом надо учесть буквальный граничный случай. Рассмотрим луч $P + t\vec{L}$. Если мы ищем пересечения начиная с $t_{min} = 0$, то найдем одно в самой точке P! Мы знаем, что P находится на сфере, значит, для $t = 0$ $P + 0\vec{L} = P$. То есть каждая точка будет отбрасывать тень на саму себя.

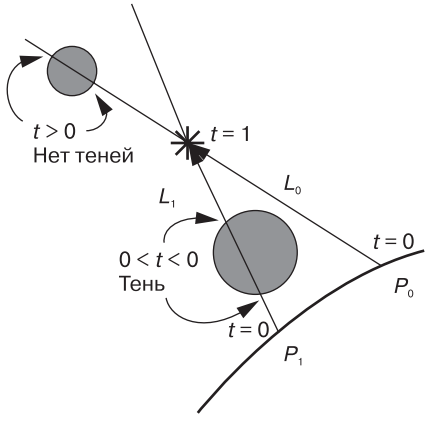

Рис. 4.3. Определяем, отбрасывается ли тень на точку, используя значение t

Простейшее решение — установить t_{min} на очень малое значение вместо 0. С точки зрения геометрии мы говорим, что луч должен начинаться немного над поверхностью, а не из самой *P*. Так диапазон будет [ε, +∞] для направленных источников и [ε, 1] для точечных.

Вы можете захотеть это исправить и не вычислять пересечения между лучом и сферой, которой принадлежит *P*. Для сфер вариант сработает, но не подойдет для объектов с формами посложнее. Например, когда вы прикрываетесь рукой от солнца, то она отбрасывает тень на ваше лицо, обе поверхности в этом случае — часть одного объекта — вашего тела.

Рендеринг с тенями

Теперь переведем все это в псевдокод.

Ранее `TraceRay` сначала вычислял ближайшее пересечение луча со сферой, а потом освещение в нем. Нам нужно извлечь код ближайшего пересечения — он нам понадобится для вычисления теней (листинг 4.1).

Листинг 4.1. Вычисление ближайшего пересечения

```
ClosestIntersection(O, D, t_min, t_max) {
    closest_t = inf
    closest_sphere = NULL
    for sphere in scene.Spheres {
        t1, t2 = IntersectRaySphere(O, D, sphere)
        if t1 in [t_min, t_max] and t1 < closest_t {
            closest_t = t1
            closest_sphere = sphere
        }
```

```
        if t2 in [t_min, t_max] and t2 < closest_t {
            closest_t = t2
            closest_sphere = sphere
        }
    }
    return closest_sphere, closest_t
}
```

Перепишем `TraceRay` для повторного использования этой функции и получим ее упрощенный вариант (листинг 4.2).

Листинг 4.2. Упрощенная версия TraceRay после вычленения ClosestIntersection

```
TraceRay(O, D, t_min, t_max) {
    closest_sphere, closest_t = ClosestIntersection(O, D, t_min, t_max)
    if closest_sphere == NULL {
        return BACKGROUND_COLOR
    }
    P = O + closest_t * D
    N = P - closest_sphere.center
    N = N / length(N)
    return closest_sphere.color * ComputeLighting(P, N, -D, closest_sphere.specular)
}
```

Теперь добавим в `ComputeLighting` проверку теней ❶ (листинг 4.3).

Листинг 4.3. ComputeLighting с поддержкой теней

```
ComputeLighting(P, N, V, s) {
    i = 0.0
    for light in scene.Lights {
        if light.type == ambient {
            i += light.intensity
        } else {
            if light.type == point {
                L = light.position - P
                t_max = 1
            } else {
                L = light.direction
                t_max = inf
            }

            // Проверка теней
         ❶ shadow_sphere, shadow_t = ClosestIntersection(P, L, 0.001, t_max)
           if shadow_sphere != NULL {
               continue
           }

            // Диффузное
            n_dot_l = dot(N, L) if n_dot_l > 0 {
                i += light.intensity * n_dot_l / (length(N) * length(L))
            }
```

```
            // Зеркальное
            if s != -1 {
                R = 2 * N * dot(N, L) - L r_dot_v = dot(R, V)
                if r_dot_v > 0 {
                    i += light.intensity * pow(r_dot_v / (length(R) * length(V)), s)
                }
            }
        }
    }
    return i
}
```

На рис. 4.4 показан итоговый вариант нашей сцены.

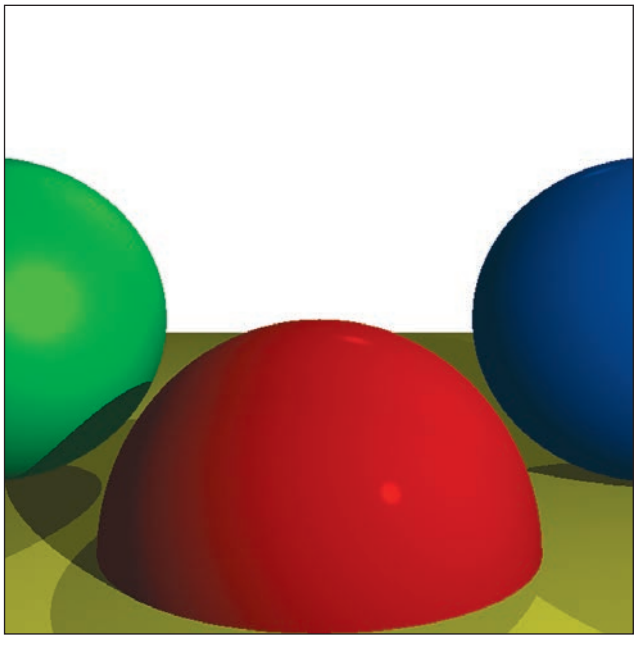

Рис. 4.4. Сцена на основе трассировки лучей, с тенями

Живую реализацию этого алгоритма можно найти по ссылке https://gabrielgambetta.com/cgfs/shadows-demo. В этом демо доступен выбор трассировки из $t = 0$ и из $t = \varepsilon$, чтобы вы смогли понять разницу между этими вариантами.

Вот теперь мы уже кое-чего добились. Объекты в сцене взаимодействуют реалистичнее, отбрасывая друг на друга тени. Дальше мы рассмотрим другие варианты взаимодействия между ними — отражение объектами других объектов.

Отражения

Мы уже говорили о зеркалоподобных поверхностях, но тогда речь шла только о придании им глянца. А можем ли мы использовать объекты, которые будут отражать на своей поверхности другие объекты? Можем, и в трассировщике сделать это будет просто, хотя сначала голова может пойти кругом.

Зеркала и отражения

Разберем принцип работы зеркала. Когда мы смотрим в него, то видим отражаемые им лучи света. Как показано на рис. 4.5, они отражаются симметрично относительно нормали поверхности.

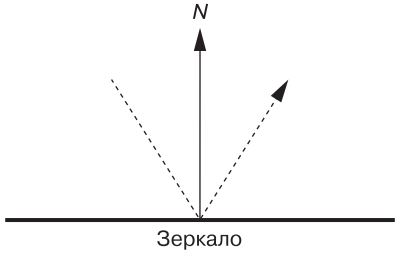

Рис. 4.5. Луч света отражается от зеркала в направлении, симметричном относительно нормали его поверхности

Допустим, мы трассируем луч, и ближайшее пересечение происходит с зеркалом. Какого цвета будет этот луч? Это не цвет самого зеркала, потому что мы наблюдаем отраженный свет. Значит, нужно выяснить, откуда он исходит и каким цветом обладает. Для этого вычислим направление отраженного луча и определим цвет света, идущего из этого направления.

Если бы только у нас была функция, помогающая нам это сделать... Подождите-ка! У нас она есть и имя ей `TraceRay`!

В основном цикле для каждого пикселя мы создаем луч из камеры в сцену и вызываем `TraceRay` для определения цвета, который камера видит в этом направлении. Если `TraceRay` определяет, что камера видит зеркало, то этой функции нужно лишь вычислить направление отраженного луча. Так мы сможем определить цвет света, идущего из этого направления. Причем вызывать `TraceRay` должна... *саму себя*.

Перечитывайте два последних абзаца, пока полностью не поймете их. Если это ваша первая встреча с рекурсивной трассировкой лучей, то прочесть придется

не единожды. Не спешите, я подожду — и как только эйфория от прекрасного момента *«Ага! Понял!»* начнет рассеиваться, мы сформулируем все более четко.

При создании рекурсивного алгоритма (вызывающего самого себя) нам нужно убедиться, что мы не порождаем бесконечный цикл (известный как «Программа не отвечает. Закрыть?»). У этого алгоритма есть два естественных условия выхода: когда луч сталкивается с неотражающим объектом и когда он ни с чем не сталкивается. Но есть простой случай, когда мы можем попасть в западню бесконечного цикла: эффект *бесконечного коридора*. Его можно увидеть, если поставить два зеркала напротив друг друга.

Есть много способов предотвратить это. Но мы просто введем в алгоритм понятие *ограничения рекурсии*, которое поможет нам контролировать, насколько «глубоко» она может уйти. Назовем это ограничение r. Когда $r = 0$, мы видим объекты без отражений. Когда $r = 1$, мы видим объекты и отражения некоторых объектов в них (рис. 4.6).

Рис. 4.6. Отражения некоторых объектов в других объектах ($r = 1$). Мы видим сферы, отраженные от других сфер, но сами они отражений не дают

При $r = 2$ мы видим объекты, отражения некоторых объектов и отражения их отражений. На рис. 4.7 виден результат для $r = 3$. Нет смысла уходить глубже трех уровней, ведь разницы уже почти не будет.

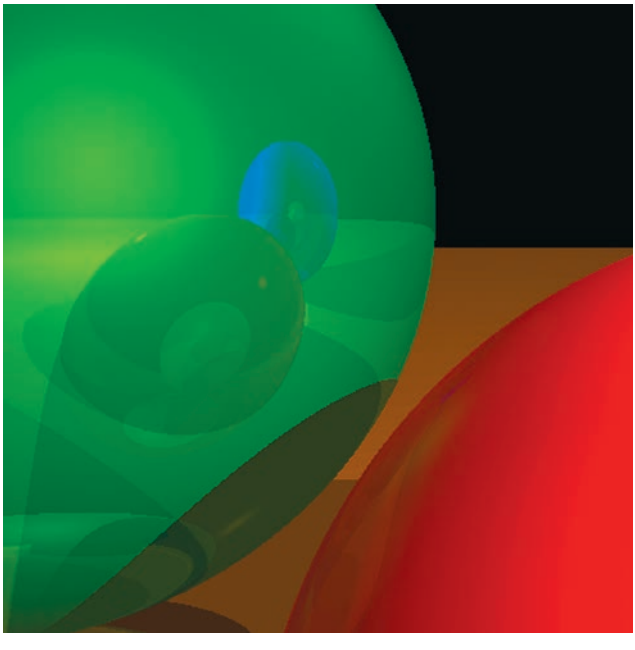

Рис. 4.7. Отражения, ограниченные тремя рекурсивными вызовами ($r = 3$). Теперь мы видим в них отражения отражений сфер

Внесем еще одно отличие: отражаемость не должна определяться как «есть» или «нет» и объекты могут быть только частично отражающими. Мы присвоим каждой поверхности число между 0 и 1 для указания степени ее отражаемости. Потом мы вычислим средневзвешенное локально освещенного и отраженного цвета, используя это число как вес.

Ну и в завершение уточним параметры для рекурсивного вызова `TraceRay`.

- Луч начинается от поверхности объекта в точке P.
- Направление отраженного луча — это направление падающего луча, отраженного от P. В `TraceRay` есть $\vec{D}$, направление падающего луча к P, значит, направление отраженного луча будет $-\vec{D}$, отраженным относительно $\vec{N}$.

- Нам не нужно, чтобы объекты отражали сами себя, поэтому $t_{min} = \varepsilon$.
- Мы хотим видеть отраженный объект независимо от его удаленности, поэтому $t_{max} = +\infty$.
- Ограничение рекурсии будет на 1 меньше его текущей границы (во избежание бесконечной рекурсии).

Теперь можно переводить все это в псевдокод.

Рендеринг с отражениями

Давайте добавим в наш трассировщик отражения. Для начала изменим определение сцены, добавив каждой поверхности свойство `reflective`. Оно будет описывать степень ее отражаемости в диапазоне от 0 (совсем не отражает) до 1 (идеальное зеркало).

```
sphere {
    center = (0, -1, 3)
    radius = 1
    color = (255, 0, 0) # Красная
    specular = 500 # Глянцевая
    reflective = 0.2 # Слегка отражающая
}
sphere {
    center = (-2, 1, 3)
    radius = 1
    color = (0, 0, 255) # Синяя
    specular = 500 # Глянцевая
    reflective = 0.3 # Чуть более отражающая
}
sphere {
    center = (2, 1, 3)
    radius = 1
    color = (0, 255, 0) # Зеленая
    specular = 10 # Немного глянцевая
    reflective = 0.4 # Еще более отражающая
}
sphere {
    color = (255, 255, 0) # Желтая
    center = (0, -5001, 0)
    radius = 5000
    specular = 1000 # Очень глянцевая
    reflective = 0.5 # Полуотражающая
}
```

Мы уже используем формулу отражения луча при вычислении зеркальных отражений, поэтому ее можно исключить. Она получает луч $\vec{R}$ и нормаль $\vec{N}$, возвращая $\vec{R}$, отраженный относительно $\vec{N}$.

```
ReflectRay(R, N) {
    return 2 * N * dot(N, R) - R;
}
```

Единственное, что следует поправить в `ComputeLighting`, — заменить уравнение отражения на вызов `ReflectRay`.

Есть небольшое изменение в основном методе — нужно передавать ограничение рекурсии в высокоуровневый вызов `TraceRay`:

```
color = TraceRay(O, D, 1, inf, recursion_depth)
```

Можно установить начальное значение `recursion_depth` на разумную величину, например 3.

Единственные заметные изменения происходят в конце `TraceRay`, где мы рекурсивно вычисляем отражения. В листинге 4.4 можно увидеть это наглядно.

Листинг 4.4. Псевдокод трассировщика лучей с отражениями

```
TraceRay(O, D, t_min, t_max, recursion_depth) {
    closest_sphere, closest_t = ClosestIntersection(O, D, t_min, t_max)

    if closest_sphere == NULL {
        return BACKGROUND_COLOR
    }

    // Вычисляем локальный цвет
    P = O + closest_t * D
    N = P - closest_sphere.center N = N / length(N)
    local_color = closest_sphere.color * ComputeLighting(P, N, -D,
    closest_sphere.specular)

    // Если достигается граница рекурсии либо объект
    // оказывается неотражающим, процесс завершается
  ❶ r = closest_sphere.reflective
    if recursion_depth <= 0 or r <= 0 {
        return local_color
    }

    // Вычисляем отраженный цвет
    R = ReflectRay(-D, N)
```

```
  ❷ reflected_color = TraceRay(P, R, 0.001, inf, recursion_depth - 1)

  ❸ return local_color * (1 - r) + reflected_color * r
}
```

Изменения в коде очень простые. Сначала выполняется проверка необходимости вычисления отражений ❶. Если сфера оказывается неотражающей или достигается граница рекурсии, мы заканчиваем и просто возвращаем собственный цвет сферы.

Самое интересное изменение в рекурсивном вызове ❷. `TraceRay` вызывает саму себя с нужными параметрами для отражения и уменьшает значение счетчика рекурсий. Вместе с проверкой ❶ это предотвращает появление бесконечного цикла.

После получения локального и отраженного цветов сферы мы их смешиваем ❸, определяя пропорции по «степени отражаемости сферы».

Результаты пусть говорят сами за себя. Смотрим на рис. 4.8.

Живую реализацию этого алгоритма можно найти по адресу: https://gabrielgambetta.com/cgfs/reflections-demo.

Рис. 4.8. Полученная на основе трассировки лучей сцена, теперь с отражениями