Реалистичная симуляции каустики

Направление: Естественно-научное

Автор исследования: Гинзбург Григорий Ильич

Ученик 11 класса МОУ СОШ №82

Научный руководитель: Аветисян Лариса Феликсовна

Черноголовка,
2021г.

ОГЛАВЛЕНИЕ

Введение

Рендеринг – это процесс получения изображения по 3D модели с помощью компьютерной программы. Частной задачей при рендеринге является построение каустики, возникающей при освещении различных предметов и сред.

Каустика – это световые узоры на поверхностях, возникающие при отражении или преломлении света. Так каустикой являются искажения света на дне бассейна, свет, сфокусированный стаканом, или даже радуга и свет, отраженный от окон на зданиях. На самом деле, мы можем наблюдать каустику каждый день, и поэтому встаёт вопрос: как мы можем симулировать каустику, чтобы генерировать реалистичные изображения?

Алгоритм для генерации фотореалистичных изображений – трассировка лучей – был изобретен еще в 1963 году, однако до сих пор не во всех современных программах для рендеринга существует возможность симуляции каустики.

Актуальность: в настоящее время технология трассировки лучей применяется в рекламе, в архитектурных визуализациях, в анимации, в фильмах, а в последнее время она начинает внедряться в игры. Без каустики реалистичность сгенерированного изображения падает, так как в нашей жизни мы можем наблюдать каустику ежедневно. Одной из наиболее часто используемых программ для генерации фотореалистичных изображений и моделирования является программа “Blender”. Однако в ней нет поддержки для генерации каустики.

Проблема: на данный момент в рендер движке “Cycles”, встроенном в бесплатную программу для 3D моделирования “Blender” невозможна качественная и полная симуляция каустики.

Цель: разработать плагин для программы Blender, для рендеринга каустики, используя язык программирования C++ и Python. Показать возможности программы и полученные результаты – рендеры каустики.

Задачи:

- Найти и изучить алгоритм работы трассировки лучей.
- Изучить проблемы и недостатки “наивной” трассировки лучей.
- Изучить алгоритмы трассировки лучей, решающие проблемы и недостатки “наивной” трассировки лучей, с помощью которых можно симулировать каустику.
- Создать собственную программу для рендеринга каустики с использованием алгоритма фотонных карт.
- Изучить требования к реализации плагинов на Python для программы Blender.
- Разработать плагин для программы Blender для рендеринга каустики.

Объект исследования: Световые узоры на поверхностях – каустика.

Предмет исследования: Реалистичная симуляция каустики.

Глава 1. Теоретическая часть – Алгоритм работы трассировки лучей

1.1 Уравнение рендеринга

Основная задача рендеринга – это построение изображения, где расположение каждого объекта известно.

Каждый объект состоит из треугольников, а координаты каждой вершины записаны. Чтобы получить изображение, мы можем добавить виртуальную камеру в сцену, испустить лучи из каждого пикселя и найти ближайший объект, который блокирует этот луч – найти первую точку пересечения луча со всеми объектами в сцене. Чтобы узнать цвет пикселя, нам необходимо узнать, сколько света отражает или испускает этот объект.

С помощью Уравнения рендеринга [Kaj86] мы можем посчитать исходящую энергетическую яркость по направлению выпущенного луча, исходящего от камеры. Ключевой принцип, который используется для вывода этого уравнения, – это закон сохранения энергии: разность между выходящей из системы энергией Φ_o и входящей Φ_i должна быть равна разности между излученной Φ_e и поглощенной энергией Φ_a.

$$\boldsymbol{\Phi_o - \Phi_i = \Phi_e - \Phi_a} \quad (1)$$

Так, чтобы выполнялся закон сохранения энергии на поверхности, исходящая энергетическая яркость $L_o(p, \omega_o)$ должна быть равна сумме излученной объектом энергетической яркости и части рассеянной объектом энергетической яркости:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_\Omega f(\omega_o, \omega_i) L_i(p, \omega_i) \cos\theta_i d\omega_i \quad (2)$$

- p – координата точки поверхности, куда пришел луч.
- ω_o – направление исходящего луча (если ω – направление испущенного луча, то $\omega_o = -\omega$).
- ω_i – направление приходящего на поверхность луча.
- $L_o(p, \omega_o)$ – излученная объектом энергетическая яркость в направлении ω_o.

- Ω - множество всех направлений ω_i, содержащихся в единичной полусфере, ориентированной по нормали к поверхности.
- $\int_\Omega \dots d\omega_i$ – интеграл по Ω.
- $f(\omega_o, \omega_i)$ - двулучевая функция отражательной способности, показывает, какая часть энергетической яркости, пришедшей в направлении ω_i, отразится от поверхности в направлении ω_o. Эта функция определяется по-разному для различных типов поверхностей – различные типы поверхностей по-разному отражают свет.
- $L_i(p, \omega_i)$ – энергетическая яркость, пришедшая в точку p по направлению ω_i. Чтобы посчитать эту яркость, пустим луч в направлении ω_i, и пусть p' - первая точка пересечения луча с объектами в сцене. Теперь мы можем посчитать энергетическую яркость, исходящую из точки p' в направлении с помощью функции, которую мы уже определили (2) - $L_o(p, \omega_o)$. То есть $L_i(p, \omega_i) = L_o(p', -\omega_i)$ – так функция $L_o(p, \omega_o)$ на самом деле рекурсивная.
- $\cos\theta_i$ – косинус угла между ω_o и ω_i ($\cos\theta_i = \omega_o \cdot \omega_i$, где $\cdot$ обозначает скалярное произведение векторов).

Теперь с помощью функции (2) $L_o(p, \omega_o)$ мы можем посчитать яркость, отраженную от любого объекта в сцене. Испустив из каждого пикселя луч, посчитав $L_o(p, \omega_o)$ и присвоив это значение пикселю, мы получим изображение.

Для генерации изображения нам необходимо посчитать многомерный интеграл $\int_\Omega \dots d\omega_i$ для миллионов пикселей в изображении, по несколько раз для каждого пикселя, из-за того, что функция $L_o(p, \omega_o)$ рекурсивная.

Оценку этого интеграла затрудняет также тот факт, что его невозможно посчитать аналитически в общем случае – такая сложность возникает из-за бесконечности вариантов расположения объектов и источников света (аналитическое решение возможно только для очень простых частных случаев).

1.2 Двулучевая функция отражательной способности

Двулучевая функция отражательной способность – это четырёхмерная функция, которая определяет, как свет отражается от данной поверхности.

Поэтому для того, чтобы было возможно рендерить разные материалы, необходимо реализовать несколько основных функций:

1. Функция для поверхностей с полностью диффузным отражением $f_{diffuse}$. Ключевой особенностью диффузного отражения является то, что $f_{diffuse} = const$, то есть свет, падая на поверхность, отражается одинаково во все стороны. Так,

$$f_{diffuse}(\omega_i, \omega_o) = \frac{albedo}{\pi}, \quad (3)$$

   где $albedo$ – цвет поверхности.
2. Функция для материалов – металлов, $f_{specular}$. Эта функция определяется так:

$$f_{specular}(\omega_i, \omega_o) = \frac{D(h)F(\omega_o,h)G(\omega_i,\omega_o,h)}{4(n\cdot\omega_i)(n\cdot\omega_o)} \quad (4)$$

   - $h = \frac{\omega_o + \omega_i}{2}$
   - $D(h) = \frac{a^2}{\pi((n\cdot h)^2(a^2-1)+1)^2}$ [WMLT07]
   - $G(\omega_i, \omega_o, h) = (n\cdot\omega_i)(n\cdot\omega_o)$ [Sigg13]
   - $F(\omega_o, h) = F_o + (1 - F_o)(1 - (\omega_o \cdot h))^5$ [Sigg13]
3. Функция для материалов – диэлектриков, поверхность которых может разниться от полностью глянцевых до шершавых f_{glossy}. Её можно определить, как совокупность двух функций (3) и (4).

$$\boldsymbol{f_{glossy}(\omega_i, \omega_o) = \frac{1}{2}(f_{glossy} + f_{diffuse})} \quad (5)$$

1.3 Метод Монте-Карло

Идея численного интегрирования Монте-Карло [Veach97] заключается в подсчете интеграла,

$$I = \int_{\Omega} f(x)\, dx \tag{6}$$

используя выборку случайных значений из Ω (семплинг). Выберем N значений из Ω $X_1, \dots \dots, X_N$ по функции распределения $p(X_i)$.
Теперь посчитаем

$$\hat{I} = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)} \tag{7}$$

Так как математическое ожидание $\mathrm{E}\left[\hat{I}\right] = \int_{\Omega} f(x)\, dx = I$, то мы можем посчитать любой интеграл, генерируя случайные значения из Ω и оценивая значение $f(x)$. Чем больше будет выборка значений, тем точнее будет посчитано значение интеграла I.

Теперь применим метод Монте-Карло для подсчёта интеграла из (2):

$$\boldsymbol{g(p, \omega_o, \omega_i) = \int_{\Omega} f(p, \omega_o, \omega_i) L_i(p, \omega_i) \cos\theta_i d\omega_i} \tag{8}$$

Будем генерировать N случайных направлений луча ω_i равномерно по всей полусфере (тогда $p(x) = \frac{1}{2\pi}$). Тогда, используя метод Монте-Карло, энергетическая яркость будет равна:

$$\boldsymbol{L_o(p, \omega_o) = \frac{1}{N} \sum_{j=1}^{N} \frac{g(p, \omega_o, \omega_j)}{p(x)} = \frac{2\pi}{N} \sum_{j=1}^{N} g(p, \omega_o, \omega_j)} \tag{9}$$

1.4 Проблемы “наивной” трассировки лучей

Из-за того, что функция $L_i(p, \omega_i)$ может иметь совершенно различные формы – свет может падать в точку p из совершенно разных сторон, то при оценке интеграла с помощью метода Монте-Карло могут возникать проблемы. В частности, если на поверхности, где мы оцениваем интеграл, сфокусирован свет – есть каустика, то форма у этой функции будет иметь резкий пик в этом месте.

Чтобы понять, почему при оценке интеграла функции с резкими и узкими пиками, посмотрим на простую $1D$ функцию.

Рассмотрим $f(x) = \sqrt{\frac{450}{\pi}}e^{-450x^2} + \frac{1}{6}$ на промежутке $[-3; 3]$.
По графику функции (**Рис. 1**) видно, что функция имеет резкий пик в 0.

Рис. 1

Нам известно, что $\int_{-3}^{3} f(x)\,dx \approx 2.$

Если же мы попытаемся посчитать такой же интеграл с помощью метода Монте-Карло, мы можем столкнуться с двумя проблемами:

1. Мы получим, что интеграл равен 1. Такое может произойти, если во время генерации случайного x на $[-3;3]$ мы ни разу не попали в промежуток $[-0.1;0.1]$, где функция принимает большие значения, то есть при всех сгенерированных $\mathrm{X}_1, \ldots\ldots, \mathrm{X}_\mathrm{N}$, где $p(\mathrm{X}_i) = \frac{1}{6}$, функция возвращала бы $\frac{1}{6}$. Тогда мы бы получили:

$$\hat{I} = \frac{1}{\mathrm{N}} \sum_{i=1}^{\mathrm{N}} \frac{f(\mathrm{X}_i)}{p(\mathrm{X}_i)} = \frac{1}{\mathrm{N}} \frac{1}{p(\mathrm{X}_i)} \sum_{i=1}^{\mathrm{N}} \frac{1}{6} = \frac{1}{p(\mathrm{X}_i)} = \frac{\frac{1}{6}}{\frac{1}{6}} = 1$$

2. Мы получим, что интеграл равен большому числу примерно равному от 5 до 9. Такое произойдет, если мы сгенерируем случайный x на $[-0.1;0.1]$. Из-за того, что шанс этого ≈ 0.03, то при маленьком количестве сгенерированных x, значение суммы будет слишком большим. Так если из десяти x, девять из них равны $\frac{1}{6}$, а один 9, то получится, что интеграл равен:

$$\hat{I} = \frac{1}{\mathrm{N}} \sum_{i=1}^{\mathrm{N}} \frac{f(\mathrm{X}_i)}{p(\mathrm{X}_i)} = 6\frac{1}{10}\left(\frac{1}{6} \cdot 9 + 9\right) = 6.3$$

В 1 случае, получилось, что посчитанное значение интеграла меньше настоящего значения – при подсчете интеграла энергетической яркости этому случаю соответствует темному пикселю, а вот во 2 случае значение интеграла гораздо больше – этому соответствует яркий белый пиксель.

Когда при подсчете (8) $g(p, \omega_o, \omega_i)$ в точке p есть каустика, то форма $L_i(p, \omega_i)$ будет с резким пиком, где пик будет являться каустикой. Поэтому при расчёте интеграла мы сталкиваемся с теми же проблемами: или каустики нет, или на изображении есть немного очень ярких пикселей. В том числе эти же проблемы есть и у рендера “Cycles”.

1.5 Метод фотонных карт

Все это время мы испускали лучи из камеры, и при оценке интеграла, из-за того что каустика – это сфокусированный свет, то был маленький шанс

испустить луч так, чтобы он попал в маленький источник света, а ведь только от маленьких или удаленных источников света появляется каустика.

Эту проблему можно решить, с помощью алгоритма фотонных карт [Jen96, HJ08, ФК]. Мы будем испускать лучи из источников света, а после хранить место, куда они упали и из какого направления.

Теперь, когда мы будем пустим луч из камеры и найдем точку пересечения на поверхности, мы можем посмотреть, какие фотоны пришли в эту точку от источников света. Однако вероятность того, что в этой точке будет хотя бы один фотон равна 0, поэтому мы можем посмотреть, какие фотоны пришли в место, неподалеку от этой точки, в круг, небольшого радиуса.
Пренебрежем тем, что фотоны расположены не в точке, где нам нужно оценить энергетическую яркость, и сложим энергетические яркости фотонов, поделив на площадь круга, в котором мы находили фотоны.

$$L_o(p,\omega_o) \approx \frac{1}{\pi r^2}\sum_{x=1}^{N} f(p,\omega_o,\omega_i)\Phi_x \qquad (10)$$

- p – позиция фотона
- Φ_x – энергетическая яркость, пришедшая из направления ω – это значение хранится в фотоне.

Полученная энергетическая яркость будет приблизительно равна настоящей яркости. Также в результате сбора соседних фотонов, при недостаточно маленьком радиусе сбора на изображении каустика получится размытой.

Теперь генерация изображения разделяется на 2 этапа:

1. Испускание фотонов из всех источников света, после чего для каждого фотона: присвоить первоначальное значение энергии фотона $\Phi_x = L_e(p,\omega)$, найти первую точку пересечения с объектами p'в сцене, сохранить фотон в p' вместе с направлением ω и энергией Φ_x. Выбрать случайное направление ω', и испустить фотон в направлении ω', а также обновить значение $\Phi_x^i = \Phi_x^{i-1} f(p,\omega',\omega_o)$.
2. Испускание лучей из камеры, нахождение первой точки пересечения и подсчёт энергетической яркости по формуле (7).

Теперь, с помощью этого алгоритма, мы можем генерировать каустику.

Глава 2. Разработка плагина для Blender

2.1 Соединение языка Python и C++

При выборе языка программирования для написания программы для рендеринга каустики мой выбор пал на C++. Именно этот язык является одним из самых быстрых, а также в нем реализована объектно-ориентированная парадигма (представление программы в виде совокупности объектов – классов, где классы образуют иерархию наследования).

Однако написание плагинов для Blender возможно только с помощью языка программирования Python, поэтому возникает проблема – как связать код плагина на Python с кодом на C++, который генерирует изображение.

Есть разные способы для соединения этих двух языков, и из них я выбрал встроенную в Python библиотеку ctypes [CTDOC], которая позволяет вызывать функции из динамической библиотеки dll. Теперь на C++ можно написать функцию, генерирующую изображение, и вызвать ее в Python в коде плагина для Blender.

2.2 Интерфейс плагина для Blender

Для того, чтобы пользователь смог отрендерить каустику с помощью плагина, необходим простой интерфейс. Поэтому стояла цель рендеринга каустики с минимальным количеством регулируемых пользователем параметров, а также все параметры должны были быть интуитивно понятными и не требующими углубленных технических знаний пользователя.

У пользователя должна быть возможность регулировать следующие параметры:

- Разрешение изображения
- Максимальный радиус сбора фотонов
- Количество испускаемых фотонов
- Настройка материалов для объектов

- Настройка источников света

В боковую панель в блендере были добавлены основные параметры: разрешение, максимальный радиус и количество фотонов для быстрого доступа к ним, а также кнопка для генерирования изображения. **(Рис 2).**

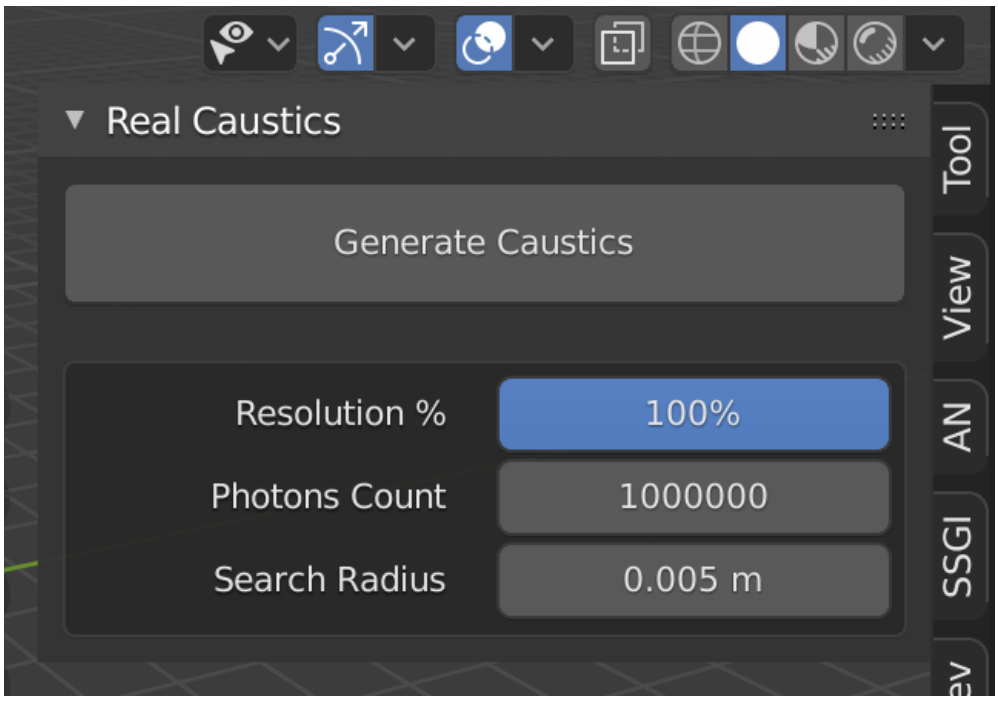

Рис. 2

Рядом с настройкой материалов, используемых Blender была добавлена панель для редактирования настроек материалов, используемых при рендере каустики (**Рис. 3**):

Материал:

- Plastic – материал диэлектрик, реализован по формуле (5) с параметрами:
  - Color – цвет.
  - Roughness (шероховатость) – насколько глянцевая или шероховатая поверхность.
  - Specular (зеркальность) – насколько яркие блики и отражения.
- Metal – материал проводник, реализован по формуле (4) с параметрами:
  - Color – цвет.
  - Roughness (шероховатость) – насколько глянцевая или шероховатая поверхность.

- Glass – прозрачный материал, с параметрами
    - Color – цвет.
    - IOR – показатель преломления для данного материала.

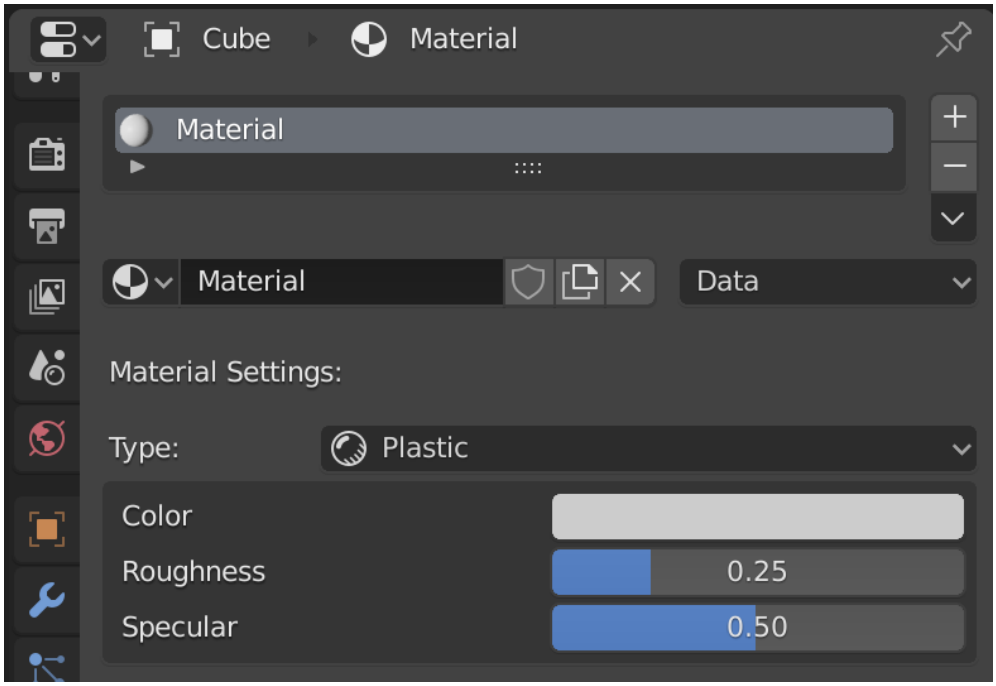

Рис. 3

В панели для настройки источников света возможно настроить тип источника **(Рис. 4):**

- Sun – удаленный источник света.
- Area – прямоугольный, конечный источник света.

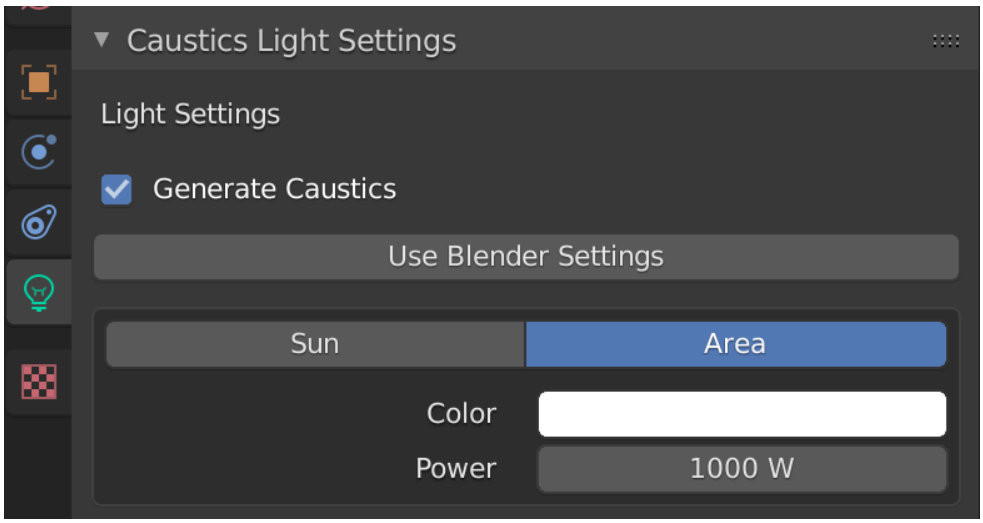

Рис. 4

2.3 Импорт геометрии сцены из Blender

Для того, чтобы сделать рендер каустики, нужно иметь доступ ко всей геометрии сцене: координате каждой вершины треугольника, а также какие из вершин соединены в треугольники.

Чтобы получить эти данные, можно экспортировать их в файл в каком-либо формате (например .obj), а потом считать их в программе. Однако этот способ будет крайне неэффективен по времени, так как запись в файл и чтение из файла миллионов строк занимает значительное количество времени.

Гораздо эффективнее будет напрямую считать геометрию из оперативной памяти – для этого в API Blender [BLAPI] есть функция (.as_pointer()), возвращающая указатель на область памяти, где хранится массив треугольников с вершинами.

Используя такой способ импорта, удается сократить время от 15 секунд (для нескольких миллионов треугольников) до 0 секунд.

2.4 Добавление каустики на рендер, полученный из рендера “Cycles”

Cycles – фотореалистичный рендер-движок, и хорошо справляется со всем многообразием различных сцен – кроме каустики, поэтому нет смысла пытаться полностью заменить изображение, которое может отрендерить Cycles, на изображение, сгенерированное плагином.

Вместо этого в плагин генерирует изображение, в котором присутствует только свет от каустики, а после добавляет его на изображение, отрендеренное Cycles, в котором полностью отключена каустика – просто складывая значения цвета каждого пикселя.

На (**Рис. 5**) показан этот процесс: изображение без каустики, отрендеренное Cycles, изображение только с каустикой из плагина и готовое изображение, полученное совмещением двух изображений.

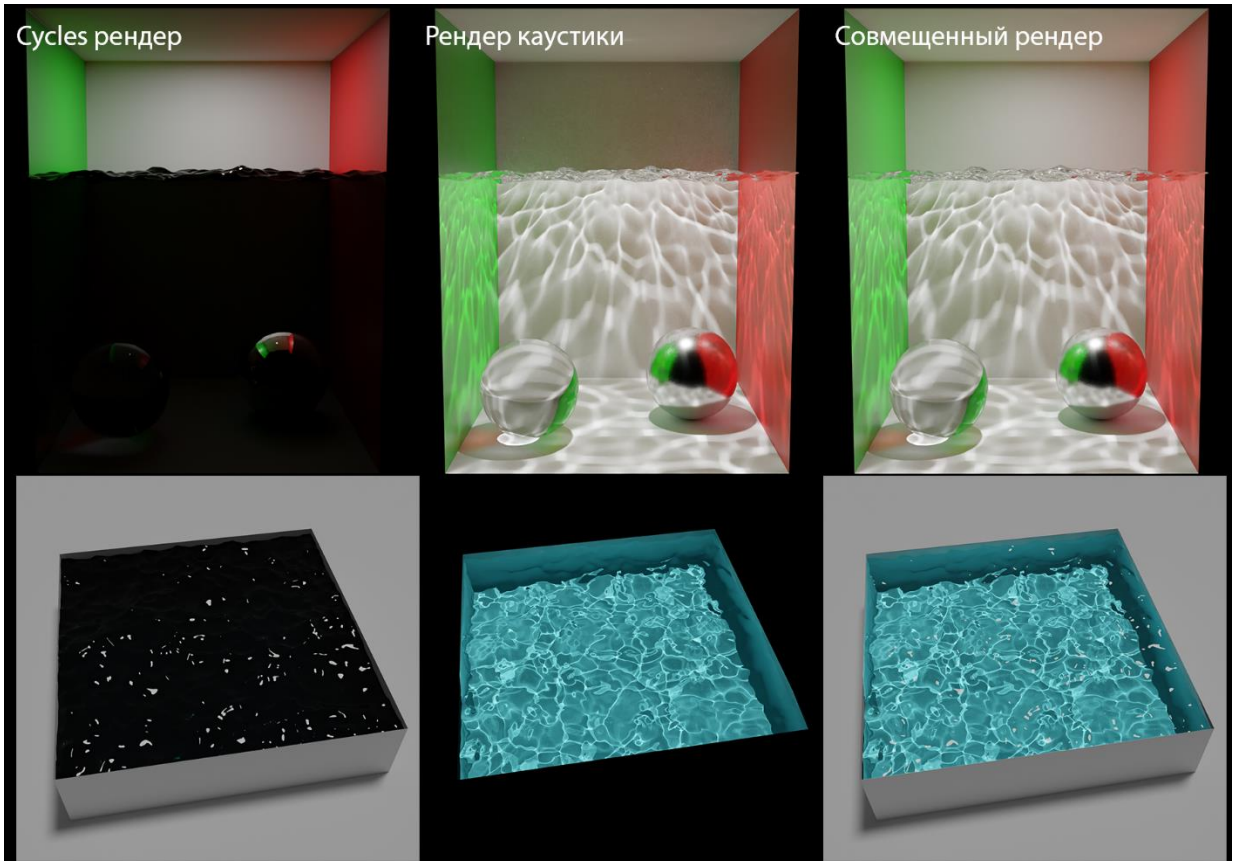

Рис. 5

2.5 Рендеринг каустики с помощью написанного плагина

Теперь для получения каустики пользователю достаточно:

1. Выбрать разрешение изображения.
2. Выбрать количество фотонов – чем их больше, тем лучше качество каустики, но дольше время рендера.
3. Нажать кнопку “Generate Caustics”.
4. Полученное изображение будет сохранено в папку с плагином.

На (**Рис. 6**) показано сравнение каустики, сгенерированной Cycles, и каустики, сгенерированной с помощью плагина.

Как мы видим из полученных изображений, качество каустики в Cycles очень плохое и непригодно для получения фотореалистичных изображений, а также ее генерирование занимает большое количество времени.

Для каждого изображения также приведено время рендера, в каждом тесте было испущено 1 000 000 миллионов фотонов.

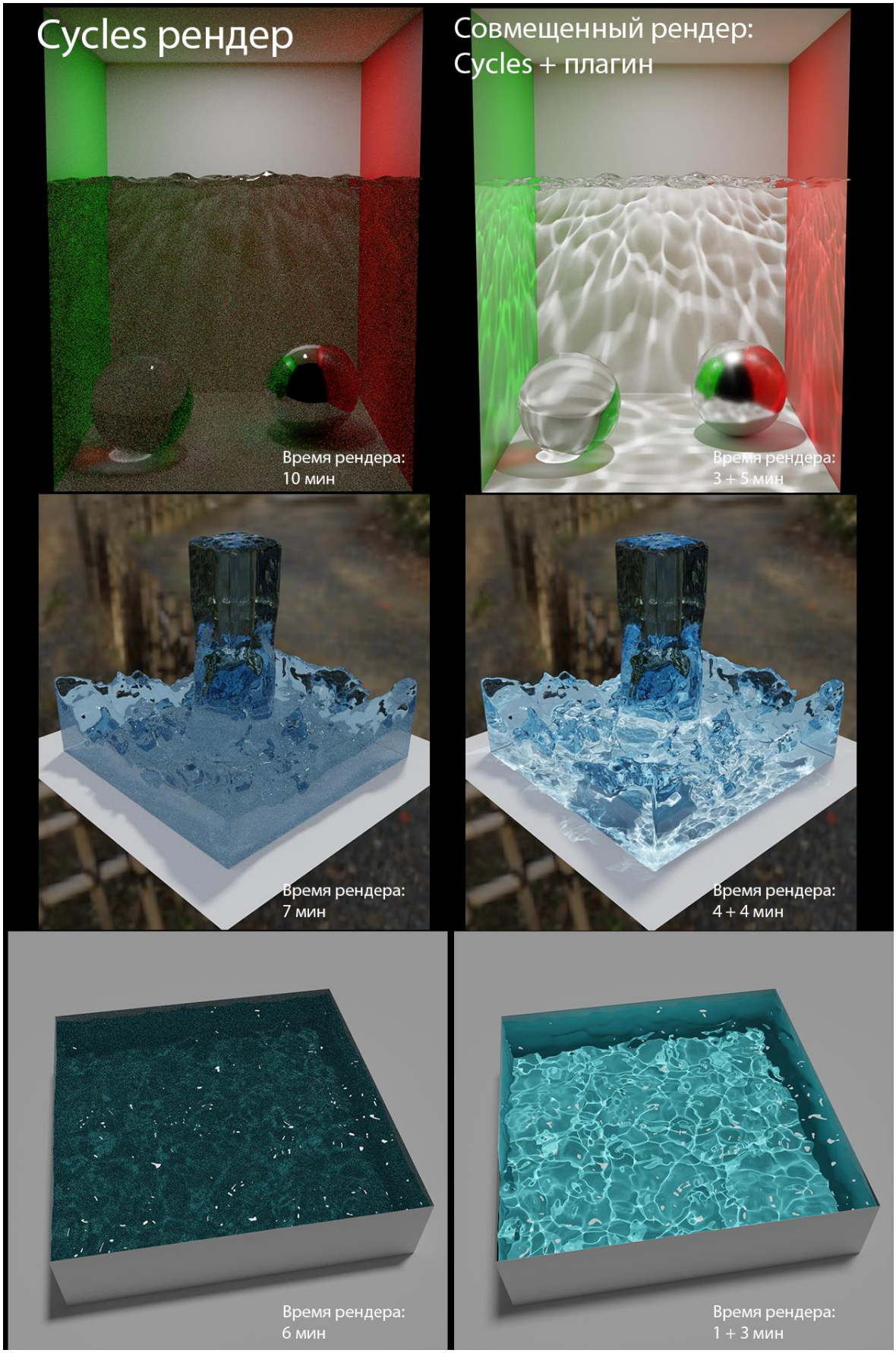
Cycles рендер
Время рендера:
10 мин
Совмещенный рендер:
Cycles + плагин
Время рендера:
3 + 5 мин
Время рендера:
7 мин
Время рендера:
4 + 4 мин
Время рендера:
6 мин
Время рендера:
1 + 3 мин

Рис. 6

Заключение

В результате проделанной работы был разработан плагин для Blender для генерации каустики. Это делает возможным ранее не доступную в Blender фотореалистичную генерацию каустики.

Генерация каустики почти полностью автоматическая – пользователю необходимо регулировать всего 2 параметра: количество фотонов и радиус сбора, и для этого не требуется углубленные технические знания со стороны пользователя.

В результате тестов было показано, что плагин не только лучше встроенной в Blender каустики, которая непригодна для фотореалистичной генерации каустики, но и в большинстве случаев генерирует каустику быстрее.

В будущем возможны следующие улучшения:

- Увеличить скорость работы программы, путем оптимизирования алгоритма нахождения пересечения с объектами в сцене, а также алгоритма нахождения фотонов в определённом радиусе.
- Добавить возможность использования текстур в материалах.
- Добавить испускание фотонов только в область видимости камеры – в настоящее время фотоны испускаются равномерно по всей сцене, и поэтому многие испускаются зря – они невидимы в конечном рендере.
- Добавить возможность использования hdri карт окружения.
- Добавить параметр roughness (шероховатость) в материал Glass.

Каустика – важный эффект в рендеринге архитектурных визуализаций, визуальных эффектов и при рендринге продуктов. Каустика также является невероятно красивым эффектом, и многие изображения после добавления каустики станут красивее и реалистичнее.

Многие люди хотели бы иметь возможность генерировать каустику в Blender, и благодаря разработанному плагину, у них теперь есть эта возможность.

Плагин размещен в открытом доступе на Github – крупнейшем веб-сервисе для хостинга IT проектов. Так, на этом сайте люди смогут увидеть плагин, скачать его и воспользоваться.

Список литературы

[Jen96] Jensen H. W.: Global illumination using photon maps., в Proceedings of the Eurographics Workshop, издатель Eurographics Association, Порто, Португалия, 17-19 июня 1996 г. с. 21-30. Url: http://graphics.ucsd.edu/~henrik/papers/photon_map/

[Kaj86] Kajiya J. T.: The rendering equation., в ACM Siggraph Computer Graphics, издатель Association for Computing Machinery, Нью-Йорк, США, т 20, 18-22 августа 1986 г. URL: https://inst.eecs.berkeley.edu//~cs294-13/fa09/lectures/p143-kajiya.pdf

[WMLT07] Walter B., Marschner S. R., Li H., Torrance K. E.: Microfacet models for refraction through rough surfaces. In Proceedings of the 18th Eurographics Conference on Rendering Techniques, издатель Eurographics Association, Гослар, Германия, 2007 г. с. 195–206. URL: https://www.cs.cornell.edu/~srm/publications/EGSR07-btdf.html

[Sigg13] Hoffman N.: SIGGRAPH 2013 Course: Physically Based Shading in Theory and Practice, 2013 г. URL: https://blog.selfshadow.com/publications/s2013-shading-course/

[Veach97] Eric Veach: Robust Monte Carlo Methods for Light Transport Simulation, Диссертация, гл. 2, с. 29 – 72, 1997 г. URL: http://graphics.stanford.edu/papers/veach_thesis/

[HJ08] Hachisuka T., Ogaki S., Jensen H. W: Progressive Photon Mapping, в ACM Transactions on Graphics, издатель Association for Computing Machinery, декабрь 2008 г. URL: https://www.ci.i.u-tokyo.ac.jp/~hachisuka/ppm.pdf

[PBRT] Matt P., Jakob W., Humphreys G.: Physically Based Rendering: From Theory to Implementation 3rd Edition, издатель Morgan Kaufmann, 25 ноября 2016 г.

[CTDOC] Документация к библиотеке ctypes. URL: https://docs.python.org/3/library/ctypes.html

[ФК] Метод Фотоных карт. URL: http://www.ray-tracing.ru/articles165.html

[BLAPI] Документация к API Blender: URL: https://docs.blender.org/api/current/index.html