## Слайд 1

(turn on the RTX, boy)

## Слайд 2

Начнём с того, что это вообще такое и почему оно важно.

Трассировка лучей — это метод рендеринга освещения, имитирующий поведение лучей в реальной жизни, с учетом законов отражения, преломления и рассеивания.

Это позволяет нам получать реалистичные: рефлексы, отражения, мягкие тени, glossiness

— то, чего сложно добиться с помощью стандартных методов.

## Слайд 3

Глобально конвейер трассировки лучей делится на 3 этапа:

Начальная генерация самих лучей, поиск пересечений с объектами (на этом этапе обычно применяются оптимизации) и расчет освещенности для каждого пикселя Viewport (т.е. того кадра, который будет выведен на экран).

## Слайд 4

И вот, собственно, чуть более детализированная схема того, как устроен конвейер. К ней вернемся позже.

## Слайд 5

Перед разбором генерации лучей обозначим важную вещь. Обычно лучи идут из источника в камеру, и мы видим только те лучи, которые в нее, собственно, попали. Но как видно на примерах – это не экономично, да и источников света может быть много. Так что пускаем лучи из камеры, а выбираем только те, которые попали в источник света.

## Слайд 6

Теперь сама генерация. Перед камерой размещаем viewport — это экран, на который будет проецироваться изображение. Каждый его пиксель — точка, через которую проходит луч.

Если мы пошлём только один луч на каждый пиксель, изображение будет очень шумным, потому что многие лучи просто не найдут источника света.

Поэтому через один пиксель отправляют несколько лучей (иногда их еще дополнительно немного смещают друг относительно друга). См видео, на котором показано, как каждый новый луч через пиксель убирает шум.

## Слайд 7

Далее луч летит по сцене, ударяется об объекты и в момент удара строится новый луч. Так происходит пока не выполнено 1 из условий:

1. Луч попал в источник света за определенное число шагов. Тогда это реальный луч от источника света, и мы можем посчитать освещенность. Оранжевый луч.
2. Луч не встретил источник света, но достигнуто максимальное количество отскоков (может показаться, что их должно быть много, но самое сильное изменение качества освещенности между 1 и 2 отскоками, дальше разница не так заметна). Синий луч.
3. Луч полностью затух, не дойдя до источника света (зачастую это происходит на 2-3 отскоке). Зеленый луч.

В 2 и 3 случаях пиксели считаются неосвещенными.

## Слайд 8

В случае, если у нас поверхность не зеркальная – при столкновении строится новый луч в СЛУЧАЙНОМ направлении, т.к. рассеивание. При этом луч затухает в зависимости от того, насколько объект будет поглощать свет. Ну и дальше весь цикл начинается заново.

Расчет освещенности происходит по формуле на слайде (тут на каждом шаге мы определенным образом храним добавочное освещение и насколько затухания). Если зеркало не идеальное, значит

## Слайд 9

Теперь разберемся с отражениями. Тут интересный момент появляется на этапе столкновения. В случае с идеальным зеркалом, новый луч пойдет не в каком-то случайном направлении, а по принципу «угол преломления равен углу отражения». Если же зеркало не идеальное – при каждом столкновении на основе коэффициента с помощью рандома выбирается, отражаемся мы сейчас или преломляется.

Основное отличие при обработке отражения заключается в том, что нам надо знать полный путь луча до источника света. Т.е. что бы понять, что отразить в зеркале после хита и как покрасить – надо дойти до света. Получается своеобразная рекурсия и мы не можем, как в случае с рассеиванием, хранить просто коэффициент затухания и добавочную освещенность. Поэтому вычисления становятся крайне ресурсоемкими.

## Слайд 10

Из всего вышесказанного не сложно заметить, что алгоритм сам по себе выглядит достаточно затратным. А мы вообще-то хотим делать 60 рендеров в секунду. Значит нужна оптимизация. И если посмотреть - самыми затратными отказаваются просчеты столкновений. При наивном подходе мы будем перебирать все полигоны всех моделей что бы проверить столкновения.

Так мы приходим к **Иерархии ограниченных объемов** или BVH. Работает это следующим образом: полигоны на сцене объединяются в большие ящики. Дальше в этих ящиках мы строим ящики чуть меньшего размера со своими примитивами и так далее. Получаем дерево ящиков, где корень – вся схема, а листья – отдельные примитивы. Когда луч попадает в ящик, мы переходим на уровень ниже и среди потомков смотрим, в какой именно блок мы попали. И так пока не дойдем до листа.

## Слайд 11

Но есть один момент. Как нам сделать разбиение на ящики таким, что бы потребовалось сделать как можно меньшее число проверок? Для этого мы используем **Эвристику площади поверхности** или SAH. Она позволяет на каждом разбиении на квадраты жадно подобрать объемы, на которые мы разобьем ящик с помощью некоторой метрики (ожидаемой стоимости трассировки, например площадь поверхности области \* на число полигонов в ней или стоимость проверки). Жадность алгоритма заключается в том, что мы предполагаем каждое разбиение последним и не задумываемся, как оно может повлиять на дальнейшее разбиение.

## Слайд 12

Современные GPU (NVIDIA RTX, AMD RDNA 2+) имеют специализированные блоки:

* RT Cores (NVIDIA) — ускоряют поиск пересечений лучей с BVH. При этом они заточены под особую структуру: acceleration structures — аппаратно оптимизированую для хранения и обработки структур ускорения.
* Они не заменяют шейдерные ядра, но ускоряют самые «тяжёлые» части алгоритма трассировки, особенно:
  + Построение и обход BVH-деревьев
  + Проверка пересечений луча с треугольниками
* **Что внутри RT Core?**

Каждый RT Core содержит:

1. BVH Traversal Unit

* + Аппаратный модуль, который быстро проходит по дереву объемлющих объёмов (BVH) .
  + Позволяет находить потенциальные пересечения без участия шейдерных ядер.

2. Intersection Acceleration Unit

* + Аппаратный модуль, который выполняет точную проверку пересечения луча с треугольником .
  + Это самая "тяжёлая" часть трассировки — и она теперь делается аппаратно , а не программно.

Acceleration Structures – это иерархические структуры ускорения, хранящие: BVH, информацию о геометрии, трансформации объектов (для TLAS).

Структуры делятся на 2 типа:

1. Top-Level Acceleration Structure (TLAS), cодержит ссылки на Bottom-Level AS .

2. Bottom-Level Acceleration Structure (BLAS), cодержит геометрию (треугольники или пользовательские примитивы).

## Слайд 13

Читать вот тут