

0.1 Трассировка лучей

Алгоритм трассировки лучей был впервые продемонстрирован Turner Whitted-ом в 1980-ом году [1]. Этот алгоритм часто называют обратной трассировкой лучей или "Whitted-style" трассировкой. Он позволяет получить такие эффекты как тени, отражения и преломления.

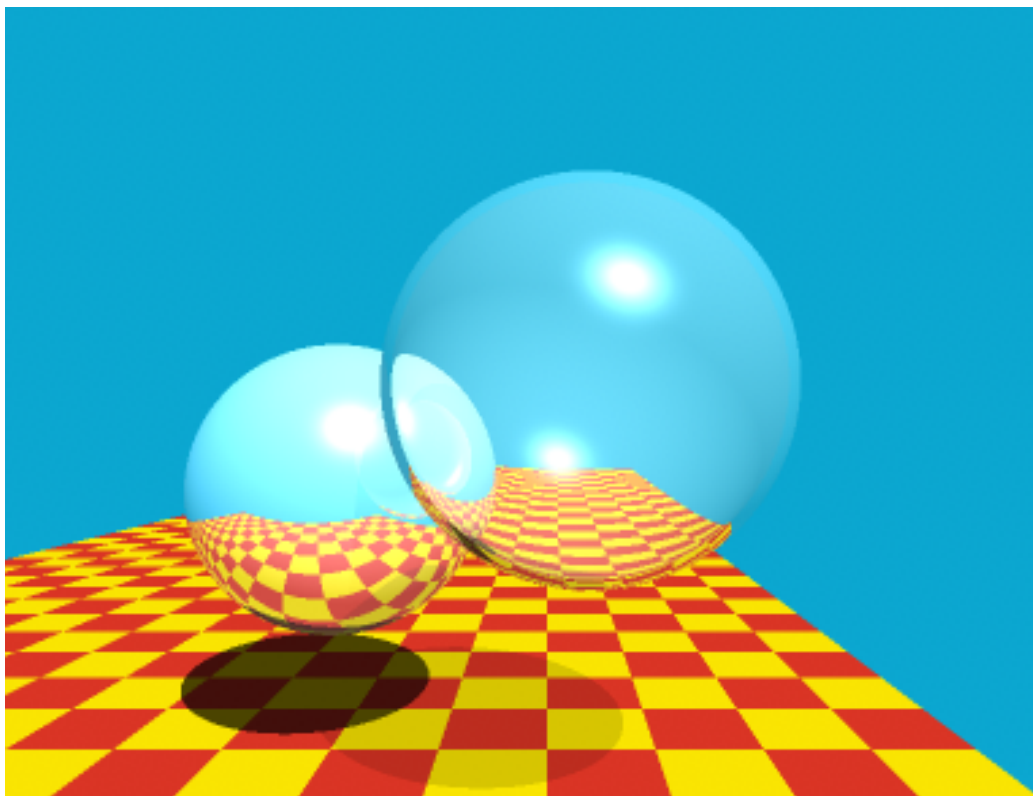
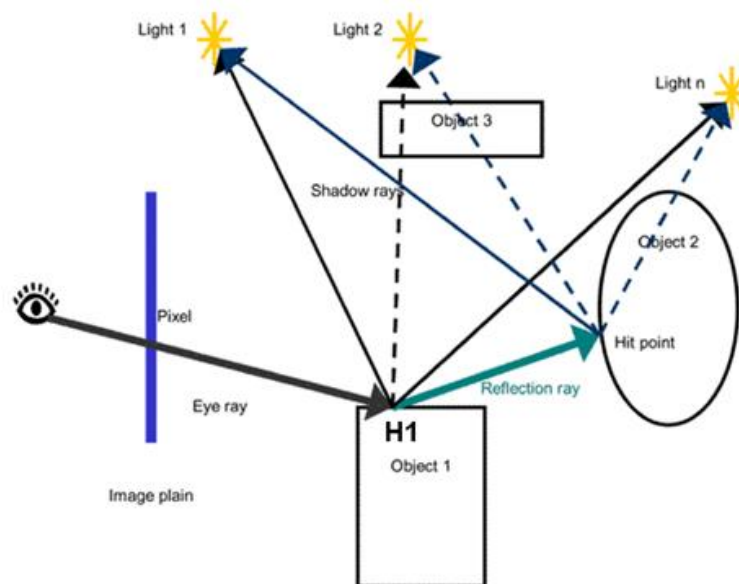


Рис. 1: Whitted-style трассировка лучей.

Обратная трассировка лучей выглядит следующим образом: из виртуального глаза через каждый пиксел изображения испускается луч и находится точка его пересечения с поверхностью сцены (для упрощения изложения мы пока не рассматриваем объемные эффекты вроде тумана). Лучи, выпущенные из глаза называют первичными. Допустим, первичный луч пересекает некий объект в точке $H1$ (рис. 2).

Для расчета тени необходимо определить для каждого источника освещения, видна ли из него эта точка. Предположим пока, что все источники света точечные. Тогда для каждого точечного источника света до него испускается теневой луч из точки $H1$ и проверяется, пересекает ли луч какие-то объекты на своем пути. Если теневой луч находит пересече-



2

Рис. 2: Схема обратной трассировки лучей.

ние с другими объектами, расположенными ближе чем источник света, значит, точка H1 находится в тени от этого источника и освещать ее не надо. Иначе, считаем освещение по некоторой локальной модели (Модель Фонга, Блина-Фонга, Кука-Торранса и.т.д.).

При учете вклада освещения от точечного источника важно делить интенсивность источника света на квадрат расстояния до него. Этот прием позволяет корректно учитывать вклад источников и мы дадим его объяснение позже, в разделе про трассировку путей.

Освещение со всех видимых (из точки H1) источников света просто складывается. Далее, если материал объекта 1 имеет отражающие свойства, из точки H1 испускается отраженный луч и для него вся процедура трассировки рекурсивно повторяется. Аналогичные действия должны быть выполнены, если материал имеет преломляющие свойства.

```
float3 RayTrace(Ray ray)
{
    float3 color(0,0,0);

    Hit hit = RaySceneIntersection(ray);
    if (!hit.exist)
        return color;

    float3 hit_point = ray.pos + ray.dir*hit.t;
```

```

for(int i=0;i < lights.size();i++)
{
    if(Visible(hit_point, lights[i]))
    {
        float R = dist(lights[i].pos, hit_point);
        float I = lights[i].Intensity;
        color += Shade(Ray, I, hit_point, hit.normal)/(R*R);
    }
}

if (hit.material.reflection > 0)
{
    Ray reflRay = reflect(ray, hit);
    color += hit.material.reflection*RayTrace(reflRay);
}

if (hit.material.transparency > 0)
{
    Ray refrRay = refract(ray, hit);
    color += hit.material.transparency*RayTrace(refrRay);
}

return color;
}

```

Listing 1: Трассировка лучей

Поясним фрагмент программы (листинг 1). Луч представлен двумя векторами. Первый вектор – `pos` – точка испускания луча. Второй – `dir` – нормализованное направление луча. Цвет – вектор из трех чисел – красный, зеленый, синий. В самом начале функции `RayTrace` мы считаем пересечение луча со сценой (представленной просто списком объектов пока что) и сохраняем некоторую информацию о пересечении в переменной `hit` и расстояние до пересечения в переменной `hit.t`. Далее, если луч промахнулся и пересечения нет, нужно вернуть фоновый цвет (в нашем случае черный). Если пересечение найдено, мы вычисляем точку пересечения `hit_point` используя уравнение луча (эквивалентное уравнению прямой с условием $t > 0$). Листинг 2 отображает представление луча в виде структуры. `pos` - точка начала луча. `dir` - нормализованное направление луча. Согласно уравнению прямой, точка на прямой вычисляется как $p = ray.pos + t * ray.dir$. Где t - одномерная координата.

```

struct Ray
{
    float3 pos;
    float3 dir;
}

```

```
}
```

Listing 2: Структура луча на основе уравнения прямой.

Когда мы вычислили точку пересечения в мировых координатах, приступаем к расчету теней и затенения (shading). Пусть источники лежат в массиве `lights`. Тогда проходим в цикле по всему массиву и для каждого источника света проверяем (той же трассировкой луча), виден ли источник света из данной точки `hit_point`. Если виден, прибавляем освещение от данного источника, вычисленное по некоторой локальной модели (например модели Фонга). Этим занимается функция `Shade`. При расчете затенения важно не забыть поделить интенсивность света источника на квадрат расстояния до него.

Если у материала объекта, о который ударился луч, есть отражающие или преломляющие свойства, трассируем лучи рекурсивно, умножаем полученный цвет на соответствующий коэффициент отражения или преломления и прибавляем к результирующему цвету. Коэффициенты `reflection` и `transparency` могут быть как монохромными так и цветными. Всё зависит от того, какая используется математическая модель для представления материалов. Для расчета преломленного луча обычно используется закон Снелла.

Важным моментом при вычислении отраженного и преломленного лучей является необходимость добавлять небольшое смещение к позиции луча. В случае отраженного луча нужно прибавлять `eps*hit.normal`, в случае преломленного - `(-1)*sign(dot(normal, ray.dir))*eps*hit.normal`. Это делается для того чтобы отраженный (или преломленный) луч не ударился на следующем шаге трассировки о ту же самую поверхность. `sign(x) = if (x > 0) 1 else -1`. `dot(x,y)` - скалярное произведение векторов `x` и `y`.

0.1.1 Генерация луча

Итак, на начальном этапе алгоритма трассировки лучей через каждый пиксел экрана необходимо выпустить луч. Это можно сделать руководствуясь несложными геометрическими соображениями:

```
float3 EyeRayDir(float x, float y, float w, float h)
{
    float fov = 3.141592654f/(2.0f);
    float3 ray_dir;

    ray_dir.x = x+0.5f - (w/2.0f);
    ray_dir.y = y+0.5f - (h/2.0f);
    ray_dir.z = -(w)/tan(fov/2.0f);
```



```

    return normalize(ray_dir);
}

```

Listing 3: Генерация 'eye' луча для камеры-обскура.

Данный подход моделирует так называемую камеру-обскура. Однако он обладает одним недостатком. Если вы создаете систему визуализации, в которой сцена может отрисовываться как трассировкой лучей, так и растеризацией (например при помощи библиотеки OpenGL), у Вас могут возникнуть трудности с получением геометрически-совпадающей картинки для растеризатора и трассировщика лучей. Причина этого заключается в том, что во всех современных системах, основанных на растеризации (и не только), используется понятия видовой матрицы (mView) и матрицы проекции (mProj). Видовая матрица переводит мир в такое пространство, где камера стоит в точке (0,0,0) и смотрит в положительном направлении оси z. Эта матрица не вызывает осложнений, т.к. вы всегда можете преобразовать луч при помощи обратной матрицы, чтобы эмитировать перемещение камеры. Однако матрицу проекции Вам необходимо пересчитать. Следующая формула используется для вычисления матрицы перспективной проекции в OpenGL:

$$mProj = \begin{bmatrix} e & 0 & 0 & 0 \\ 0 & e/a & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Где:

- $e = 1/\tan(\text{FOV}/2)$
- FOV - Field Of View; угол обзора. Обычно равен $\frac{\pi}{2}$.
- a - aspect ratio; $a = \text{height}/\text{width}$;
- f - far clip plane; дальняя плоскость отсечения.
- n - near clip plane; ближняя плоскость отсечения.

В случае камеры-обскура, f стремится к бесконечности, а n - к нулю. Таким образом, получаем:

$$mProj = \begin{bmatrix} e & 0 & 0 & 0 \\ 0 & e/a & 0 & 0 \\ 0 & 0 & -1 & -2n \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} e & 0 & 0 & 0 \\ 0 & e/a & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Если же вашей системе рендеринга требуется обеспечить совместимость с какой-либо библиотекой растеризации или движком, где матрица проекции уже задана (или задается извне и менять ее Вы не можете), необходимо использовать другой алгоритм генерации луча:

```
float3 EyeRayDir2(float x, float y, float w, float h)
{
    float4 pos( 2.0f * (x + 0.5f) / w - 1.0f,
               -2.0f * (y + 0.5f) / h + 1.0f, 0.0f, 1.0f );

    pos = g_mViewProjInv*pos; // inverse(mView*mProj)*pos
    pos /= pos.w;             // homogeneous coordinates

    pos.y *= (-1.0f);

    return normalize(pos.xyz);
}
```

Listing 4: Генерация 'eye' луча для камеры с заданной извне матрицей проекции.

Здесь `g_mViewProjInv` - обратная матрица от произведения видовой матрицы (`mView`) и матрицы проекций (`mProj`).

0.1.2 Устранение ступенчатости

Рассмотрим причину появления ступенчатости. Изображение хранится в памяти как двумерная матрица пикселей. Однако, считается, что изображение предметов реального мира это не просто набор пикселей. Авторы [2] склонны рассматривать изображение как непрерывную двумерную функцию. То, что мы видим на экране – приближение этой функции, ее дискретизованное в заданном разрешении представление. Ступенчатость - результат дискретизации.

Трассировка лучей в самом общем понимании - это метод, позволяющий восстановить значения функции изображения с помощью точечных выборок, то есть значений этой функции в точках. Самый простой способ устранения ступенчатости - трассировать больше чем 1 лучей на пиксел с некоторыми смещениями внутри самого пиксела, а полученный результат усреднять. Однако для того чтобы уменьшить ступенчатость, достаточно часто не только трассируют больше чем 1 луч на пиксел, но и применяют фильтрацию. Рассмотренное выше усреднение соответствует так называемому *box-фильтру*. Достаточно простым, но более эффективным методом является *билинейная фильтрация*, при которой вклад от луча распределяется по 4 ближайшим пикселям.

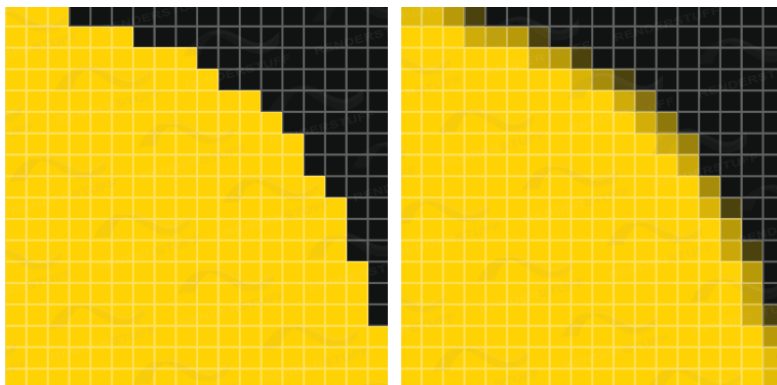


Рис. 3: Устранение ступенчатости.

Как правило процесс вычисления значения функции стараются отделить от способа хранения изображения. По этой причине во многих системах рендеринга присутствуют такие классы как *Sampler* (отвечает за фильтрацию) и *Integrator* (отвечает собственно за вычисление значения цвета).

0.1.3 Прозрачные объекты и тени

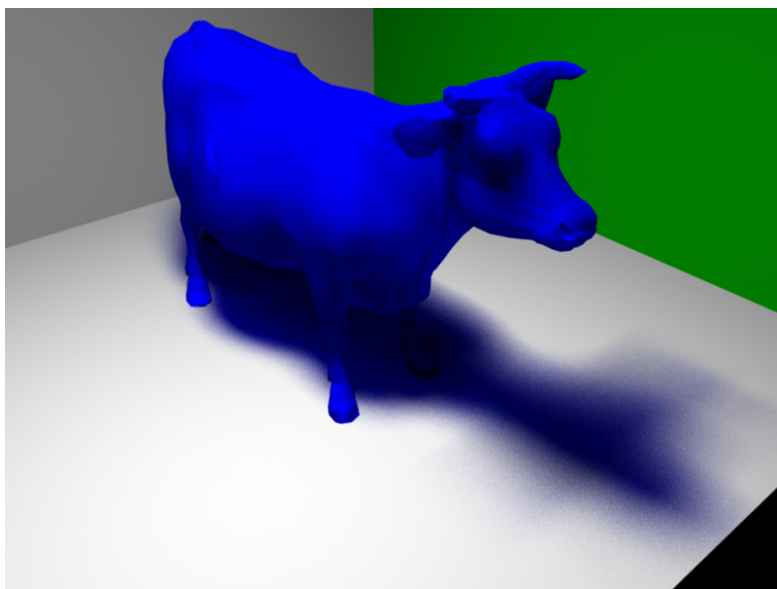


Рис. 4: Цветная тень.

При расчете тени, можно использовать более сложную модель. Если ес-

ли есть вероятность того, что один объект перекрывается другим прозрачным объектом, можно трассировать теневой луч сквозь прозрачный объект, рассчитывая по закону Бугера-Ламберта-Бэра затухание в прозрачной среде в зависимости от пройденного расстояния (этот же закон нужно использовать и при учете затухания внутри прозрачных объектов для обычных лучей). При таком алгоритме тень становится цветной. Разумеется, тени, рассчитанные таким образом корректны, только если прозрачный объект, отбрасывающий тень, имеет близкий к единице коэффициент преломления (считаем что коэффициент преломления воздуха равен 1). Если это не так, то под прозрачным объектом образуется сложная картина, называемая каустиком. Типичный пример каустика – солнечный зайчик от стакана воды, когда через него просвечивает солнце. Мы рассмотрим корректные способы расчета каустиков в разделе про вычисление интеграла освещенности.

0.1.4 Корректный расчет преломлений

Рассмотренный ранее алгоритм не позволит получить реалистичные изображения преломляющих объектов. Причин этому 2. Во-первых существует эффект так называемого полного внутреннего отражения при котором выходя из более плотной среды в менее плотную, при большом угле падения, луч полностью отражается внутрь более плотной среды и преломления не происходит. Во-вторых, реальное поведение света для преломляющих объектов не описывается простыми коэффициентами отражения (reflection) и пропускания (transparency). Корректное описание в этом случае может быть получено при помощи формул Френеля, которые устанавливают зависимость между долей отраженного и преломленного света в зависимости от угла падения луча и показателя преломления материала (material.IOR). Алгоритм учитывающий корректное преломление представлен ниже.

```
float3 RayTrace(Ray ray)
{
    float3 color(0,0,0);

    Hit hit = RaySceneIntersection(ray);
    if (!hit.exist)
        return color;

    float3 hit_point = ray.pos + ray.dir*hit.t;

    for(int i=0; i < lights.size(); i++)
    {
        if(Visible(hit_point, lights[i]))
```

```

    {
        float R = dist(lights[i].pos, hit_point);
        float I = lights[i].Intensity;
        color += Shade(Ray, I, hit_point, hit.normal)/(R*R);
    }
}

float3 reflection    = hit.material.reflection;
float3 transparency = hit.material.transparency;

if(hit.material.type == FRESNEL)
    ApplyFresnel(ray, hit.norm, hit.material.IOR,
                 &reflection, &transparency);

if (reflection > 0)
{
    Ray reflRay = reflect(ray, hit);
    color += reflection*RayTrace(reflRay);
}

if (transparency > 0)
{
    Ray refrRay;
    if(refract(ray, hit, &refrRay))
        color += transparency*RayTrace(refrRay);
}

return color;
}

```

Listing 5: Трассировка лучей с корректным учетом преломлений.

В приведенном выше листинге считается, что функция 'refract' возвращает false при возникновении полного внутреннего отражения. В соответствии с [3] в листинге 6 мы приводим формулы Френеля для стекла в виде исходных кодов. `cosTheta1` - угол между падающим лучом и нормалью. Эта величина может быть вычислена как `dot(ray.dir, hit.normal)`. `etaExt` - коэффициент преломления внешней среды; `etaInt` - внутренней. Обратите внимание, что в листинге 6 ситуация, при которой луч ударяет поверхность снизу (`cosTheta1`) моделируется при помощи смены местами коэффициентов преломления для `etaInt` и `etaExt`. Полученное при помощи функции 'fresnel(...)' значение, таким образом, есть коэффициент отражения; '1.0-fresnel(...)' - коэффициент пропускания.

```

float fresnelDielectric(float cosTheta1, float cosTheta2,
                        float etaExt, float etaInt)
{
    float Rs = (etaExt * cosTheta1 - etaInt * cosTheta2)

```

```

        / (etaExt * cosTheta1 + etaInt * cosTheta2);

float Rp = (etaInt * cosTheta1 - etaExt * cosTheta2)
        / (etaInt * cosTheta1 + etaExt * cosTheta2);

return (Rs * Rs + Rp * Rp) / 2.0f;
}

float fresnel(float cosTheta1, float etaExt, float etaInt)
{
    // Swap the indices of refraction if the interaction starts
    // at the inside of the object
    //
    if (cosTheta1 < 0.0f)
        swap(etaInt, etaExt);

    // Using Snell's law, calculate the sine of the angle
    // between the transmitted ray and the surface normal
    //
    float sinTheta2 = etaExt/etaInt *
        sqrt(max(0.0f, 1.0f - cosTheta1*cosTheta1));

    // Total internal reflection!
    //
    if (sinTheta2 > 1.0f)
        return 1.0f;

    // Use the sin^2+cos^2=1 identity - max() guards against
    // numerical imprecision
    //
    float cosTheta2 = sqrt(max(0.0f, 1.0f - sinTheta2*sinTheta2));

    // Finally compute the reflection coefficient
    //
    return fresnelDielectric(abs(cosTheta1), cosTheta2, etaInt, etaExt);
}

```

Listing 6: Формулы Френеля для диэлектрика [3].

0.1.5 Поиск пересечений

Поиск пересечений луча с геометрическими объектами сводится к нахождению всех общих точек луча и объекта. Мы рассмотрим наиболее важные виды примитивов, оставив читателю иные виды геометрических фигур в качестве самостоятельной работы.

Треугольник, барицентрический тест

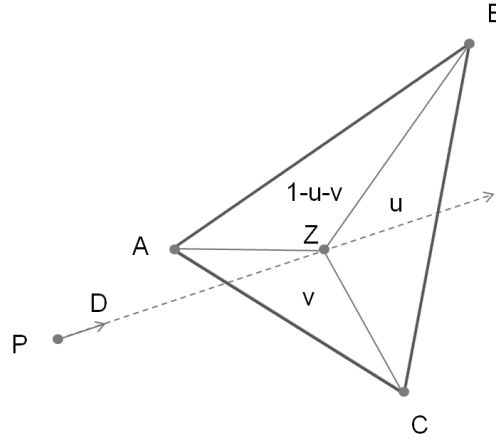


Рис. 5: Барицентрический тест треугольника и луча.

Введем следующие обозначения:

- P - точка из которой вылетает луч;
- D - направление луча;
- A, B, C - вершины треугольника;
- Тройка $(u, v, 1-u-v)$ - барицентрические координаты. Они представляют собой отношения площадей маленьких треугольников к большому треугольнику. То есть $u = S(ZCB)/S(ABC)$, $v = S(ZAC)/S(ABC)$, $1-u-v = S(ZAB)/S(ABC)$;

$$Z(u, v) = u * A + v * B + (1 - u - v) * C \quad (1)$$

$$Z(u, v) = P + t * D \quad (2)$$

$$P + t * D = u * A + v * B + (1 - u - v) * C \quad (3)$$

Имея 3 точки на плоскости, можно выразить любую другую точку через ее барицентрические координаты. Первое уравнение получается из определения барицентрических координат, выражая точку пересечения Z . С другой стороны, эта же точка Z лежит на прямой. Второе уравнение таким образом, это параметрическое уравнение прямой. Приравняв правые части уравнений 1 и 2 получаем третье уравнение, которое, по сути,

является системой 3-х уравнений (P, D, A, B, C - векторы) с 3-мя неизвестными (u, v, t). Проведя алгебраические преобразования получим ответ в следующем виде:

$$\begin{aligned} E1 &:= B - A \\ E2 &:= C - A \\ T &:= P - A \\ P &:= \text{cross}(D, E2) \\ Q &:= \text{cross}(T, E1) \end{aligned}$$

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\text{dot}(P, E1)} * \begin{bmatrix} \text{dot}(Q, E2) \\ \text{dot}(P, T) \\ \text{dot}(Q, D) \end{bmatrix}$$

Треугольник, юнит-тест (Woops unit test)

Основная идея данного алгоритма заключается в том, чтобы посчитать матрицу преобразования треугольника в некий единичный треугольник (отсюда название) с вершинами $(1,0,0)$; $(0,1,0)$; $(0,0,0)$; и нормалью $(0,0,1)$. Во время подсчета пересечения с треугольником луч преобразуется этой матрицей в пространство, где треугольник имеет единичное представление. Назовем такое преобразование T_{Δ} . После преобразования вычислить пересечение намного проще, так как нужно считать пересечение с заранее известным треугольником - $(1,0,0)$; $(0,1,0)$; $(0,0,0)$.

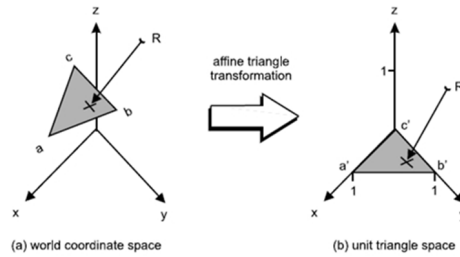


Рис. 6: Преобразование T_{Δ} .

Пусть задан треугольник с вершинами A, B, C . Рассмотрим преобразование T_{Δ}^{-1} :

$$T_{\Delta}^{-1} = \begin{bmatrix} A_x - C_x & B_x - C_x & N_x - C_x \\ A_y - C_y & B_y - C_y & N_y - C_y \\ A_z - C_z & B_z - C_z & N_z - C_z \end{bmatrix} * X + \begin{bmatrix} C_x \\ C_y \\ C_z \end{bmatrix} \quad (4)$$

Заметим, что применив данное преобразование к треугольнику $(1,0,0); (0,1,0); (0,0,0)$ - получим исходный треугольник (A,B,C) . То есть T_{Δ}^{-1} в действительности является обратным преобразованием к T_{Δ}

$$T_{\Delta}^{-1} * \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = A \quad T_{\Delta}^{-1} * \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = B \quad (5)$$

$$T_{\Delta}^{-1} * \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = C \quad T_{\Delta}^{-1} * \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = N \quad (6)$$

Для того чтобы найти нужное нам преобразование необходимо дополнить матрицу T_{Δ}^{-1} до 4x4 и (добавить еще одну строчку $(0,0,0,1)$) и найти обратную матрицу. Это будет соответствовать преобразованию T_{Δ} .

Для каждого треугольника матрицу T_{Δ} нужно рассчитать один раз и сохранить. Код, вычисляющий пересечение луча и треугольника приведен ниже (здесь m - матрица для преобразования T_{Δ}).

```
float3 o = mul3x4(m, ray_origin);
float3 d = mul3x3(m, ray_direction);

float t = -o.z/d.z;
float u = o.x + t*d.x;
float v = o.y + t*d.y;
```

Listing 7: Код пересечения луча и треугольника (woops unit test).

Функция `mul3x4` выполняет умножение подматрицы 3x3 на трехмерный вектор и добавляет к результату последний столбец (3 его компоненты). Функция `mul3x3` просто умножает подматрицу 3x3 на трехмерный вектор.

Прямоугольный параллелипипед (AABB)

Прямоугольный параллелипипед (по англ. Axis Aligned Bounding Box - или AABB) обычно представляется в виде координат 2 точек. Нижнего левого угла (третье измерение опущено) - `boxMin` и правого верхнего угла - `boxMax`. Все точки, координаты которых находятся в интервале

$[\text{boxMin}, \text{boxMax}]$ по всем 3 измерениям находятся внутри параллелипипеда. Для того чтобы вычислить пересечение луча и прямоугольного параллелипипеда сначала вычисляют точки пересечения луча со всеми 6 плоскостями параллелипипеда (получая значения координат t в уравнении луча), после чего из каждой пары плоскостей (для x, y и z) выделяют минимум и максимум (рис. 7). Результирующие координаты ищут как максимальный из минимумов (t_{\min}) и минимальный из максимумов (t_{\max}).

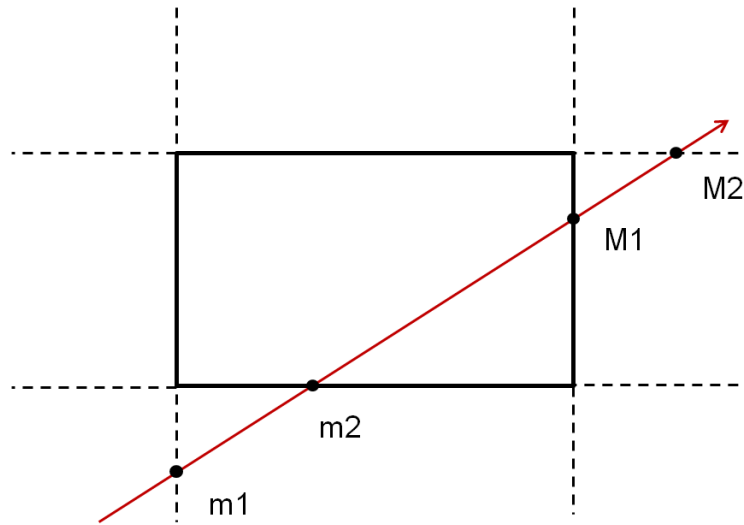


Рис. 7: Пересечение луча и AABB. Точки $m1$ и $m2$ - минимумы пересечения луча с парой плоскостей. $M1$ и $M2$ - максимумы.

Нетрудно заметить, что алгоритм работает корректно, рассмотрев различные случаи расположения луча и AABB.

```
bool RayBoxIntersection(float3 ray_pos, float3 ray_dir_inv,
                        float3 boxMin, float3 boxMax,
                        float& tmin, float& tmax)
{
    float lo = ray_dir_inv.x*(boxMin.x - ray_pos.x);
    float hi = ray_dir_inv.x*(boxMax.x - ray_pos.x);

    tmin = fminf(lo, hi);
    tmax = fmaxf(lo, hi);

    float lo1 = ray_dir_inv.y*(boxMin.y - ray_pos.y);
    float hi1 = ray_dir_inv.y*(boxMax.y - ray_pos.y);

    tmin = fmaxf(tmin, fminf(lo1, hi1));
```

```

    tmax = fminf(tmax, fmaxf(lo1, hi1));

    float lo2 = ray_dir_inv.z*(boxMin.z - ray_pos.z);
    float hi2 = ray_dir_inv.z*(boxMax.z - ray_pos.z);

    tmin = fmaxf(tmin, fminf(lo2, hi2));
    tmax = fminf(tmax, fmaxf(lo2, hi2));

    return (tmin <= tmax);
}

```

Listing 8: Код пересечения луча и AABB.

В приведенном листинге `ray_dir_inv` - инвертированное направление луча ($1.0/\text{ray_dir}$), которое для каждого луча можно вычислять 1 раз. Функции `fmaxf` и `fminf` вычисляют соответственно максимальное и минимальное значение из 2 чисел и обычно входят в пакет `<math.h>` компилятора GCC.

Функции расстояний (distance-aided ray marching)

Пересечение луча и поверхности не всегда удобно искать в явном виде. Вместо этого существует простой алгоритм, позволяющий сделать это итеративно, даже для тех поверхностей, для которых явная формула пересечения луча и поверхности вообще не существует (или ее очень сложно вывести) [4]. Мы будем называть данный алгоритм маршированием по лучу при помощи функций расстояний.

Пусть существует функция $f(x,y,z)$, задающая поверхность уравнением вида $f(x,y,z) = 0$. Для краткости можно будем писать $f(p) = 0$. Если в функцию f вместо p подставить, некоторую точку, не лежащую на поверхности, мы получим некое не равное нулю число. Нетрудно заметить, что это число будет представлять собой знаковое (хотя это может зависеть от формулы) расстояние до поверхности.

Далее мы шагаем по лучу на полученное расстояние и применяем эту процедуру еще раз, до тех пор, пока наше расстояние не станет меньше заданного порога. Размер шага следует ограничивать снизу некоторой константой, для т.к. в противном случае трассировка будет очень медленной вблизи границ объектов.

Для того чтобы не продолжать марширование по лучу до бесконечности, необходимо в самом начале найти пересечение луча и ограничивающего всю сцену bounding box-а. Как только вы выходите за этот бокс ($t > t_{\max}$) – остановить реймарчинг. Либо вы просто можете ограничить максимальное число шагов по лучу какой-либо достаточно большой константой.

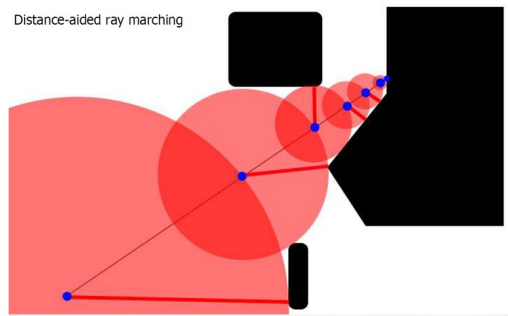


Рис. 8: Иллюстрация итеративного поиска пересечения с поверхностями, заданным как функции расстояний [4].

```
Hit DistanceAidedRayMarch(float3 ray_pos, float3 ray_dir, \
                          float tmin, float tmax)
{
    float epsilon = 0.001f;

    float minStep = epsilon;
    float maxStep = 10.0f;

    float t = tmin;
    float dt = 1.0f;

    float dist = 1.0f; // signed distance

    while(dist > epsilon && t < tmax)
    {
        float3 p = ray_pos + t*ray_dir;
        dist = DistanceEvaluation(p);
        dt = clamp(dist, minStep, maxStep);
        t += dt;
    }

    float3 hitPoint = ray_pos + (t-dt)*ray_dir;

    Hit hit;

    if(dist<=epsilon)
    {
        hit.norm = EstimateNormal(hitPoint, epsilon);
        hit.pos = hitPoint;
    }

    return hit;
}
```

```
}
```

Listing 9: Поиск пересечения луча и поверхности заданной при помощи функции расстояния.

В приведенном выше листинге функция DistanceEvaluation как раз и является той $f(x,y,z)$ которая задает поверхность. Функция EstimateNormal вычисляет нормаль на основе того факта что она совпадает с градиентом $f(x,y,z)$:

```
// Calculate the gradient in each dimension
// from the intersection point
//
float3 EstimateNormal(float3 z, float eps)
{
    float3 z1 = z + float3(eps, 0, 0);
    float3 z2 = z - float3(eps, 0, 0);
    float3 z3 = z + float3(0, eps, 0);
    float3 z4 = z - float3(0, eps, 0);
    float3 z5 = z + float3(0, 0, eps);
    float3 z6 = z - float3(0, 0, eps);

    float dx = DistanceEvaluation(z1) - DistanceEvaluation(z2);
    float dy = DistanceEvaluation(z3) - DistanceEvaluation(z4);
    float dz = DistanceEvaluation(z5) - DistanceEvaluation(z6);

    return normalize(float3(dx, dy, dz) / (2.0*eps));
}
```

Listing 10: Вычисление нормалей для поверхности заданной при помощи функции расстояния [4].

Важное замечание состоит в том, что для некоторых видов поверхностей, часто используемых в компьютерной графике (например NURBS) не только не существует аналитической формулы для вычисления пересечения, но также не существует и аналитического представления в виде $f(x,y,z) = 0$. Такие поверхности не удастся визуализировать при помощи рассмотренного метода напрямую.

0.1.6 Ускорение поиска пересечений

Ускорение поиска пересечений в трассировке лучей - довольно обширная тема. Мы начнем рассмотрение с наиболее простых регулярных разбиений пространства и впоследствии перейдем к нерегулярным структурам, учитывающим особенности трассировки лучей.

Если сцена представлена набором примитивных объектов (сфера, AABB, треугольник), то ускорять процесс поиска пересечений при помощи струк-

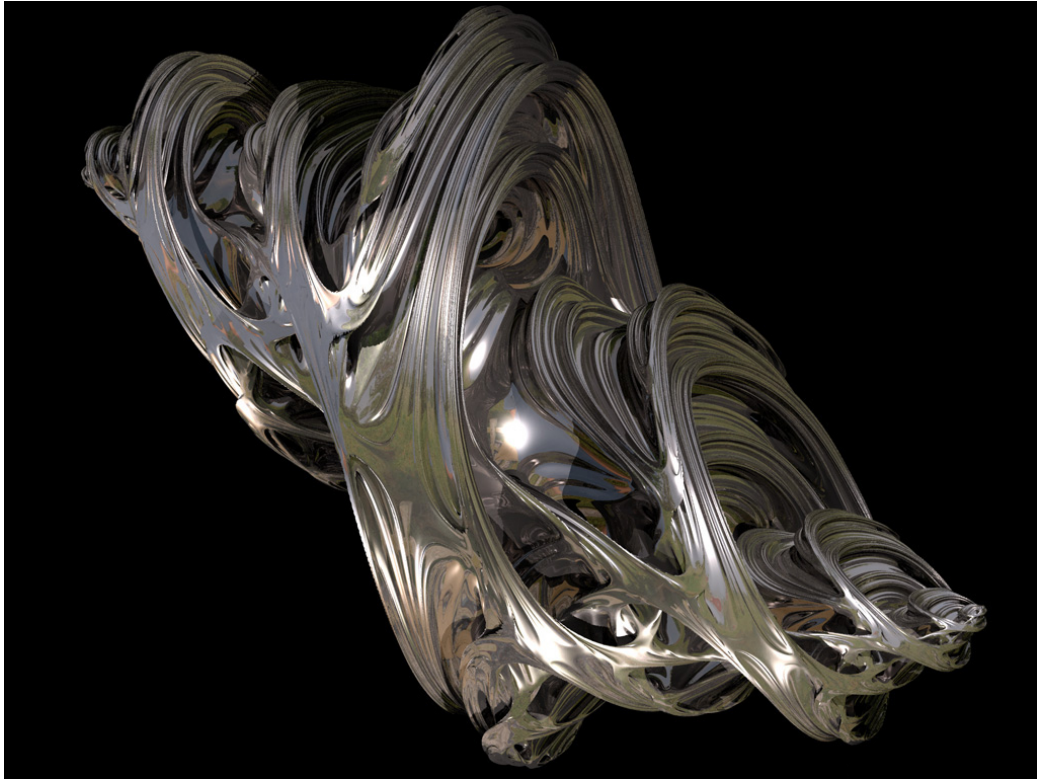


Рис. 9: Фрактал Julia, рассчитанный при помощи трассировки лучей и функций расстояния [4].

тур пространственного разбиения имеет смысл, как правило, если число объектов больше одного-двух десятков. В противном случае, стоимость поиска может быть сравнима со стоимостью перебора всех объектов вслепую. Как правило, это происходит из-за большего количества ошибок предсказателя ветвлений при выполнении сложного кода (особенно верно для иерархического поиска в дереве).

Регулярные сетки

Данная ускоряющая структура регулярно разбивает пространство на N^3 вокселей. При поиске в регулярной сетке проверяются только те объекты, которые оказались внутри вокселей на пути луча.

Алгоритм построения сетки выглядит следующим образом:

```
for(int objId=0;objId<objects.size();objId++)
{
    AABB3f box = roundedBoundingBox(objects[objId]);
```

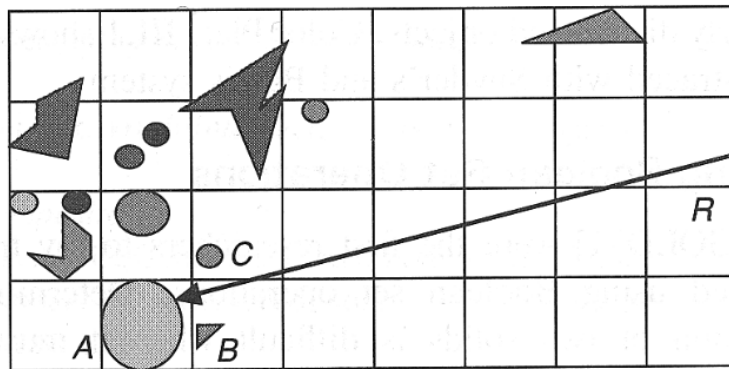
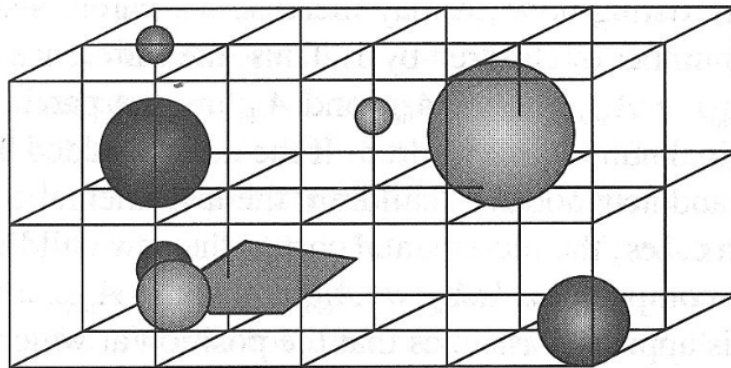


Рис. 10: Регулярное разбиение пространства.

```

for(int x = box.min.x; x <= box.max.x; x++)
{
    for(int y = box.min.y; y <= box.max.y; y++)
    {
        for(int z = box.min.z; z <= box.max.z; z++)
        {
            if(objects[objId].overlaps(voxel(x,y,z)))  ]
                voxel(x,y,z).appendRef(&objects[objId]);
        }
    }
}
}

```

Listing 11: Построение регулярной сетки.

В данном листинге для каждого объекта находится его ограничивающий AABV и координаты найденного AABV округляются функцией `roundedBoundingBox` в нижнюю сторону. После чего для всех вокселей внутри этого AABV

проверяется их пересечение с объектами.

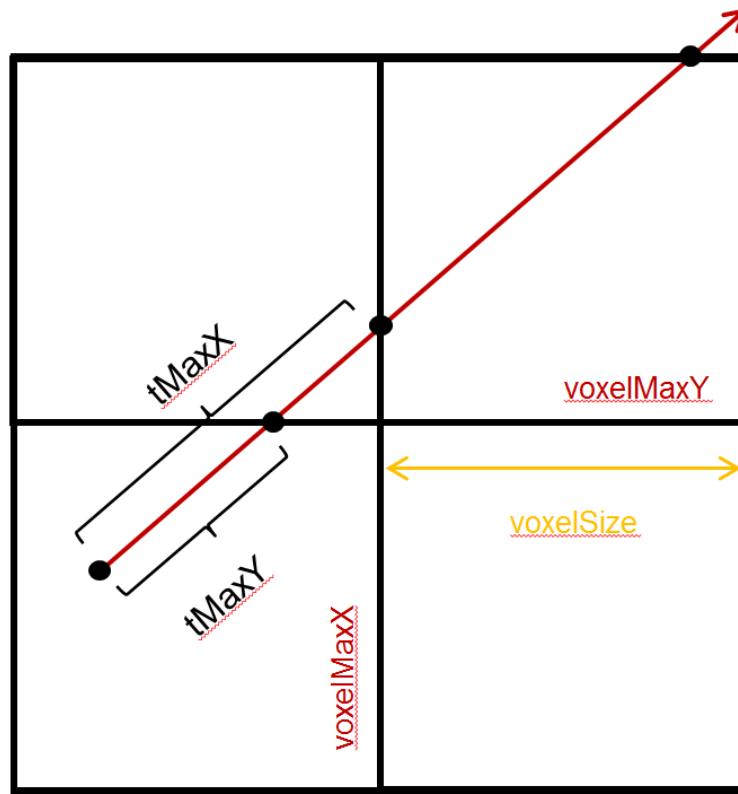


Рис. 11: Поиск в регулярной сетке.

Алгоритм поиска носит фамилию своего автора [5]. Функция поиска состоит из 2 частей. Инициализация и перебор вокселей. Инициализирующая часть довольно объемная, однако, она выполняется для каждого луча 1 раз.

```
float FujimotoTraverse2D(Ray2f ray, AABB2f box)
{
    float t_min;

    if(!IntersectRay2fBox2f(ray, box, t_min))
        return;

    if(t_min < 0)
        t_min = 0;

    // initializing part
    //
```



```

float startX = ray.pos.x + t_min*ray.dir.x;
float startY = ray.pos.y + t_min*ray.dir.y;

int x = int( (startX-box.vmin.x)/(box.vmax.x-box.vmin.x)*CELL_NUMBER );
int y = int( (startY-box.vmin.y)/(box.vmax.y-box.vmin.y)*CELL_NUMBER );

if(x == CELL_NUMBER) x--;
if(y == CELL_NUMBER) y--;

float2 boxSize = box.vmax - box.vmin;

float tVoxelX, tVoxelY;
int stepX, stepY;

if(ray.dir.x > 0)
{
    tVoxelX = float(x+1)/float(CELL_NUMBER);
    stepX = 1;
}
else
{
    tVoxelX = float(x)/float(CELL_NUMBER);
    stepX = -1;
}

if(ray.dir.y > 0)
{
    tVoxelY = float(y+1)/float(CELL_NUMBER);
    stepY = 1;
}
else
{
    tVoxelY = float(y)/float(CELL_NUMBER);
    stepY = -1;
}

float voxelMaxX = box.vmin.x + tVoxelX*boxSize.x;
float voxelMaxY = box.vmin.y + tVoxelY*boxSize.y;

float tMaxX = t_min + (voxelMaxX - startX)/ray.dir.x;
float tMaxY = t_min + (voxelMaxY - startY)/ray.dir.y;

const float voxelSize = (box.vmax.x - box.vmin.x)/CELL_NUMBER;
const float tDeltaX = voxelSize/fabs(ray.dir.x);
const float tDeltaY = voxelSize/fabs(ray.dir.y);

// core of the traversal
//
while(x < CELL_NUMBER && x >= 0 && y < CELL_NUMBER && y >= 0)

```

```

{
    float tHit = IntersectAllObjectsInVoxel(x, y, box, ray);

    if(tHit < fminf(tMaxX,tMaxY))
        return tHit;

    if(tMaxX < tMaxY)
    {
        tMaxX += tDeltaX;
        x += stepX;
    }
    else
    {
        tMaxY += tDeltaY;
        y += stepY;
    }
};

return 1e38f; // no hit were found
}

```

Listing 12: Поиск в регулярной сетке.

Константа CELL_NUMBER задает разрешение сетки. Переменным VoxelMaxX и VoxelMaxY - соответственно x и y координаты ближайших плоскостей-границ сетки; voxelSize - размер вокселя, а tMaxX и tMaxY - расстояние от текущей точки на луче до пересечения с плоскостями VoxelMaxX и VoxelMaxY (рис. 11). Модификация части тела цикла перебора вокселей для 3D случая представлена ниже:

```

if(tMaxX <= tMaxY && tMaxX <= tMaxZ)
{
    tMaxX += tDeltaX;
    x += stepX;
}
else if(tMaxY <= tMaxZ && tMaxY <= tMaxX)
{
    tMaxY += tDeltaY;
    y += stepY;
}
else
{
    tMaxZ += tDeltaZ;
    z += stepZ;
}

```

Listing 13: Модификация основной части поиска в регулярной сетке для трехмерного пространства.

Несмотря на простой алгоритм перебора вокселей, регулярная сетка - не

очень хорошая ускоряющая структура. Причин этому несколько:

1. Высокие затраты памяти, пропорционально N^3 .
2. Отсутствие адаптивности или проблема чайника на стадионе. Регулярная сетка одинаково разбивает все пространство независимо от того, какая геометрия и как расположена на сцене. Размер воксела выбирается исходя из некоторого среднего размера объектов. При этом если выбрать размер сетки большим, поиск пересечений значительно замедлится для небольших высокополигональных объектов. А если размер воксела выбрать маленьким, не только значительно возрастут затраты памяти но и замедлится трассировка на всей остальной сцене, т.к. в процессе трассировки луча придется перебирать большее число вокселей.
3. Проблема повторных пересечений. Эта проблема также усугубляется с ростом разрешения сетки. Объекты, пересекающие несколько вокселей при поиске пересечений будут встречаться несколько раз. Следовательно, при трассировке в регулярной сетке луч будет вычислять пересечение с такими примитивами более чем один раз. Проблему можно частично амортизировать сохраняя идентификатор луча в примитивах, с которым пересечение уже было посчитано. Если в данный момент времени идентификатор луча равен идентификатору сохраненному в примитиве, вычислять пересечение не нужно. Однако этот подход трудно применять при параллельной реализации трассировки лучей.

Окто-деревья и иерархические сетки

Перейдя к иерархическому представлению, можно в некоторой степени решить первые 2 проблемы регулярной сетки (затраты памяти и неадаптивность). Иерархическая сетка представляет из себя дерево (обычно небольшой глубины, но зато очень широкое), в узлах которого лежат относительно небольшие регулярные сетки. Если размер сетки положить равным $2 \times 2 \times 2$ получаем окто-дерево. Таким образом, окто-дерево представляет из себя иерархическое разбиение пространства на 8 частей (рис. 12).

Окто-деревья широко используются в компьютерной графике для различных задач пространственного поиска, но они довольно плохо подходят для ускорения трассировки лучей. Алгоритм обхода окто-деревьев достаточно сложен. При этом адаптивность у окто-дерева низкая, так как чтобы ограничить небольшой объект (чайник на стадионе) нужно

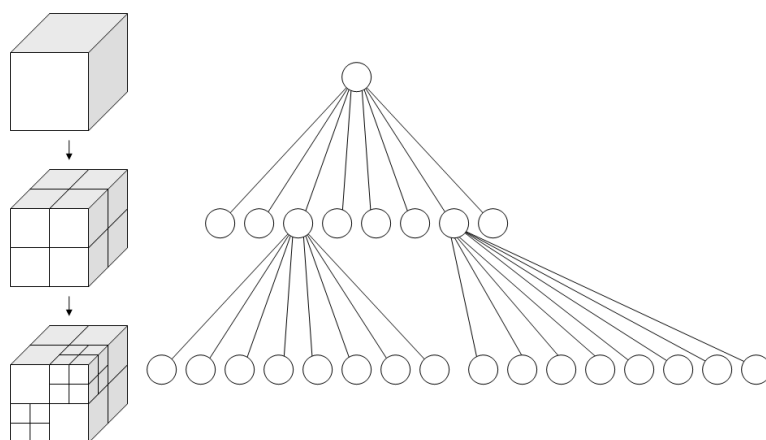


Рис. 12: Окта-дерево.

много уровней подразбиения. Окта-дерево, таким образом, обычно содержит множество пустых узлов и сложные регионы обходятся медленно (рис. 13). При этом, проблема повторных пересечений по-прежнему остается актуальной для окта-деревьев, поскольку при разбиении один объект очень часто попадает сразу в несколько дочерних узлов.

kd-деревья

Рассмотрим структуру бинарного пространственного разбиения, называемую kd-дерево (аббревиатура kd расшифровывается как k-dimensional). Эта структура представляет собой бинарное дерево ограничивающих параллелепипедов, вложенных друг в друга. Каждый параллелепипед в kd-дерево разбивается плоскостью, перпендикулярной одной из осей координат на два дочерних параллелепипеда.

Вся сцена целиком содержится внутри корневого параллелепипеда, но, продолжая рекурсивное разбиение параллелепипедов, можно прийти к тому, что в каждом листовом параллелепипеде будет содержаться лишь небольшое число примитивов. Таким образом, kd-дерево позволяет использовать бинарный поиск для нахождения примитива, пересекаемого лучом.

Если плоскость, разбивающую пространство выбирать каждый раз по середине параллелипипеда (при этом текущий узел можно разбивать всегда по оси, в которой он имеет максимальный размер), kd-дерево будет эквивалентно окта-дереву и наследует все его недостатки (за исключением сложного алгоритма обхода). Однако, основное преимущество kd-дерева над окта-деревом заключается как раз в том, что плоскость разби-

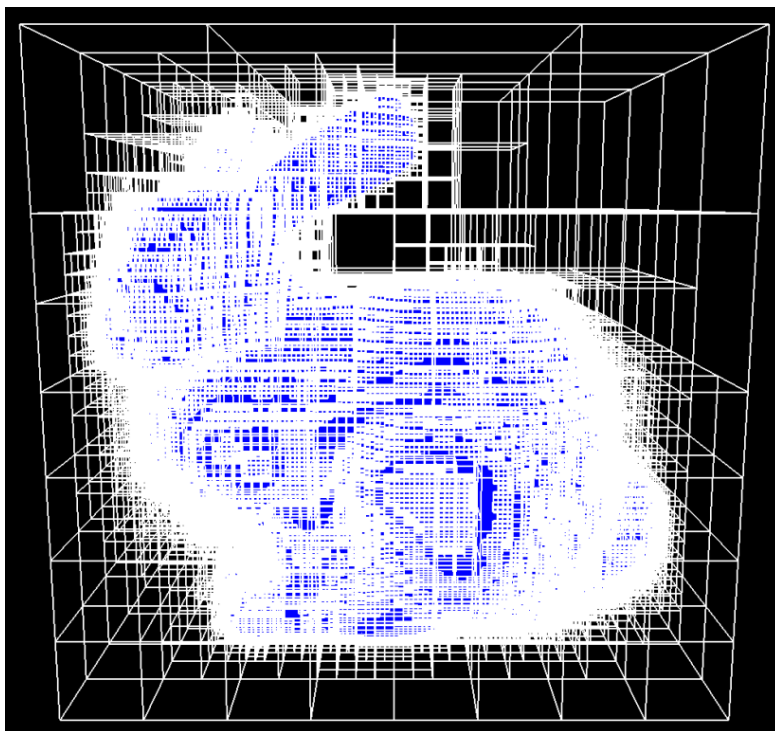


Рис. 13: Модель стэнфордского кролика в окто-дереве. Количество белого цвета на изображении визуально позволяет оценить сложность трассировки.

ения можно ставить не только по середине разбиваемого узла, а в любом месте.

Алгоритм построения kd-дерева можно представить следующим образом (будем называть прямоугольный параллелепипед сокращенно AABV).

1. "Добавить" все примитивы в ограничивающий AABV. Т.е построить ограничивающий все примитивы AABV, который будет соответствовать корневому узлу дерева.
2. Если примитивов в узле мало или достигнут предел глубины дерева, завершить построение.
3. Выбрать плоскость разбиения, которая делит данный узел на два дочерних. Будем называть их правым и левым узлами дерева.
4. Добавить примитивы, пересекающиеся с AABV левого узла в левый узел, примитивы, пересекающиеся с AABV правого узла в правый.

5. Для каждого из узлов рекурсивно выполнить данный алгоритм начиная с шага 2.

На рисунке 14 изображен процесс построения kd-дерева с учетом некоего оптимального алгоритма выбора разбивающей плоскости. Самым сложным в построении kd-дерева является 3-ий шаг. От него напрямую зависит эффективность ускоряющей структуры. Существует несколько способов выбора плоскости разбиения, рассмотрим их по порядку.

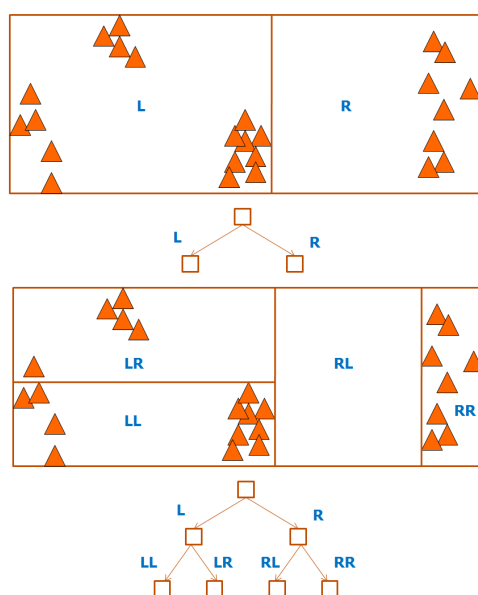


Рис. 14: Построение kd-дерева. L - обозначает левое поддерево. R - правое.

Разбиение по середине.

Самый простой способ - выбирать плоскость разбиения по центру. Сначала выбираем ось (x, y или z), в которой AABV имеет максимальный размер, затем разбиваем его по центру. Ранее мы обсуждали что kd-дерево в этом случае имеет плохую адаптивность и эквивалентно окто-дереву.

Разбиение по медиане.

Если разбивать узел на два дочерних таким образом, чтобы в правом и левом поддереве количество примитивов было одинаково, мы построим сбалансированное дерево. Это не очень удачная идея. Все дело в том, что сбалансированные деревья могут помочь только если искомый элемент каждый раз находится в случайном узле, то есть если распределение



Рис. 15: Разбиение по середине.

лучей по узлам во время поиска будет равномерно. В действительности, это не так. Лучей в среднем пойдет больше в тот узел, который больше по своей площади поверхности, а при медианном разбиении эти площади у узлов могут быть разные.



Рис. 16: Разбиение по медиане.

Разбиение на основе оптимизации функции стоимости.

Каковы же критерии хорошо-построенного kd-дерева? На интуитивном уровне такой критерий можно описать фразой "как можно больше пустого пространства должно быть отброшено как можно быстрее". Для решения поставленной задачи используем формальный подход. Введем функцию стоимости поиска, которая будет отражать, насколько дорого по вычислительным ресурсам производить поиск в данном узле случайным набором лучей. Будем вычислять ее по следующей формуле [6]:

$$SAH(x) = CostEmpty + SA(Left) * N(Left) + SA(Right) * N(Right)$$

В приведенной выше формуле $SA(Left)$ и $SA(Right)$ - площадь поверхности (SurfaceArea) соответственно левого и правого дочерних узлов, получающихся при разбиении. $N(Left)$ и $N(Right)$ - количество примитивов, попавших при разбиении соответственно в левое и правое поддерево. Аргумент функции x является одномерной координатой плоскости разбиения. На рисунке 17 можно увидеть, что SAH сразу отбрасывает большие пустые пространства, плотно ограничивая геометрию.

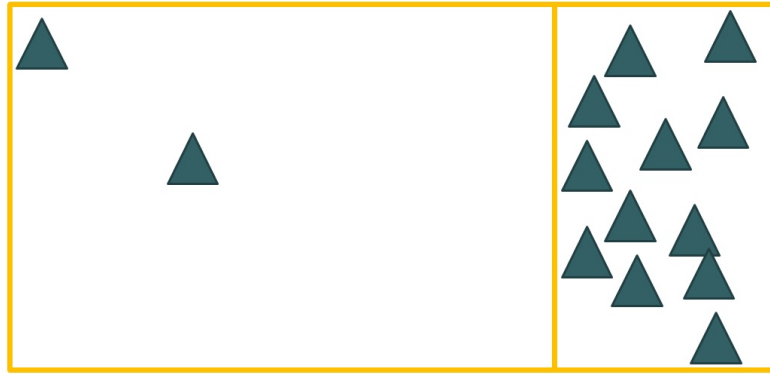


Рис. 17: Разбиение с учетом оптимизации стоимости.

Хорошими кандидатами на минимум SAH могут служить границы примитивов. Простой алгоритм посторения выглядит следующим образом: каждый раз при выборе плоскости нужно перебрать всевозможные границы примитивов по трем измерениям, вычислить в них значение функции стоимости и найти минимум среди всех этих значений. Когда мы вычисляем SAH для каждой плоскости, то нам необходимо знать $N(left)$ и $N(right)$ - количества примитивов справа и слева от плоскости. Если вычислять N простым перебором, в итоге получится квадратичный по сложности алгоритм построения.

Действенным способом ускорения поиска минимума Φ -ии SAH является использование какого-либо метода минимизации одномерной функции с несколькими начальными приближениями. Например, метод золотого сечения. Это избавляет нас от полного квадратичного перебора.

Критерий остановки.

Одним из критериев остановки может служить сама функция SAH. Если оцениваемая функцией суммарная стоимость поиска в дочерних узлах

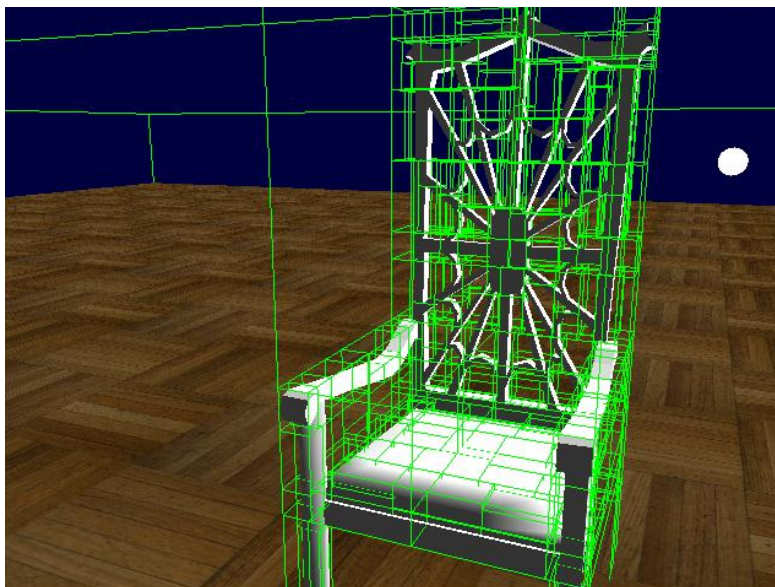


Рис. 18: kd-дерево, построенное с учетом SАН.

больше стоимости поиска в родительском узле, разбиение стоит остановить. Однако, весьма неочевидным фактом является то, что такой критерий остановки далеко не оптимален. Дело в том, что увеличение SАН при разбиении текущего уровня дерева еще не означает замедление трассировки для всего дерева. Рассмотрим цилиндр, составленный из полигонов на рисунке 19. Узел kd-дерева, обозначенный буквой А является проблемным. Какую-бы плоскость разбиения мы не выбрали, SАН полученных дочерних узлов будет больше чем SАН родительского узла. Таким образом, SАН в данном случае говорит о том, что подразбиение нужно остановить. Но это не всегда так.

Если представить, что цилиндр - высокополигональный объект, дальнейшее подразбиение (даже с учетом выбора плоскости по центру) приведет к тому, что в листьях окажется не более чем 1-2 примитива. Однако остается открытым вопрос - будет ли при этом быстрее трассировка лучей и если да, то на сколько. И очередной возникающий вопрос формулируется так: Если существуют такие плоскости разбиения, которые увеличивают суммарный SАН текущего узла (который может быть оценен как $SАН(левого) + SАН(правого)$), но при этом все-же позволяют ускорить трассировку, когда именно следует выбирать такие плоскости (т.е. на каком уровне дерева)?

Техника, позволяющая решить описанную выше проблему и ответить на поставленные вопросы называется в англоязычной литературе "Early

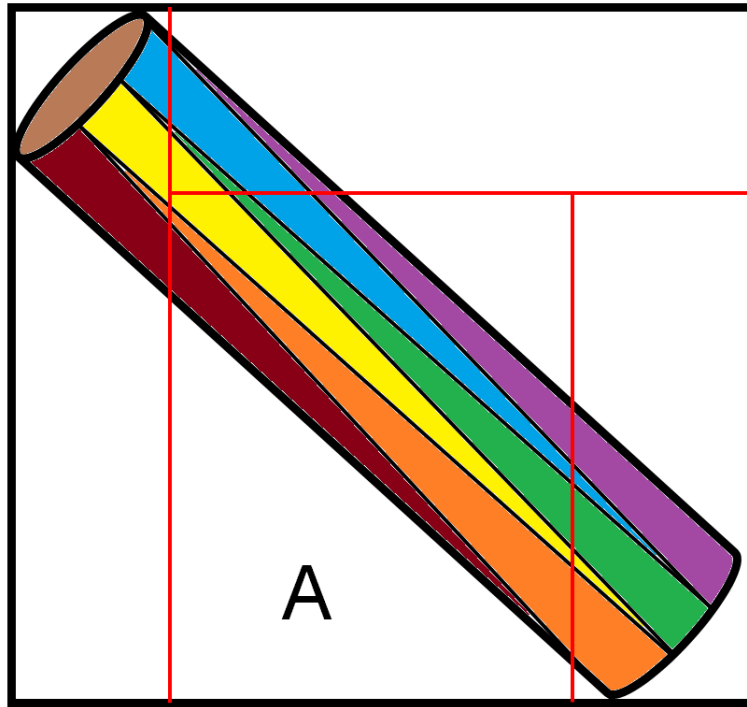


Рис. 19: Цилиндр, составленный из треугольников. Какую бы плоскость разбиения мы не выбрали для узла A, это не позволит уменьшить SAH.

Split Clipping" [7]. Мы будем называть ее ранним подразбиением примитивов. Она говорит, что примитивы, которые плохо аппроксимируются при помощи AABV нужно подразбивать на несколько более мелких примитивов еще до построения дерева с тем, чтобы рассматривать такие 'сложные' плоскости наравне со всеми остальными в процессе оптимизации SAH. Мы подробнее рассмотрим данную технику при обсуждении BVH деревьев.

Поиск в kd-дереве.

Классический алгоритм бинарного поиска в kd деревьях (kd-tree traversal в англоязычной литературе), состоит в следующем: На первом шаге алгоритма необходимо посчитать пересечение луча с ограничивающим сцену корневым параллелепипедом (AABV) и запомнить информацию о пересечении в виде двух координат (в пространстве луча) – t_{near} и t_{far} , обозначающих пересечение с ближней и дальней плоскостями соответственно. На каждом следующем шаге необходима информация только о текущем узле (его адрес) и этих двух координатах.

При этом нет необходимости вычислять пересечение луча и дочернего параллелепипеда, достаточно лишь узнать пересечение с разбивающей параллелепипед плоскостью (обозначим соответствующую координату как t_split).

В случае если $t_split \geq t_far$ (рис. 20) или если $t_split < t_near$ (рис. 21) луч пересекает только один дочерний узел, поэтому можно просто отбросить правый (соответственно левый) узел и продолжить поиск пересечения в оставшемся узле.

В случае если луч пересекает оба дочерних узла (рис. 22), поэтому необходимо сначала поискать пересечение в ближнем узле и если оно не найдено, искать его в дальнем. Так как в общем случае неизвестно, сколько раз произойдет последнее событие, необходим стек. Каждый раз, когда луч пересекает оба дочерних узла, адрес дальнего узла, t_near и t_far помещаются в стек и поиск продолжается в ближнем. Если в ближнем узле пересечение не найдено, из стека достаются адрес дальнего узла, t_near , t_far и поиск продолжается в дальнем узле.

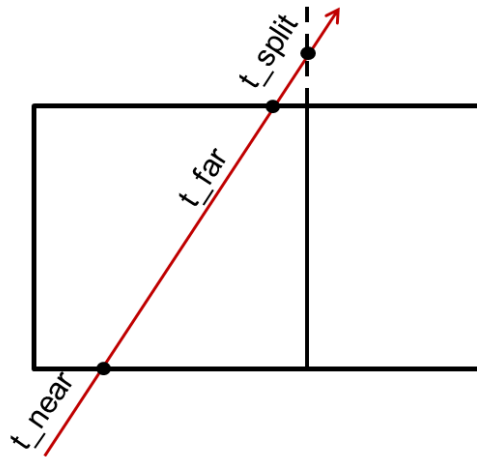


Рис. 20: Поиск в kd-дереве. ($t_split \geq t_far$).

```
bool KdTreeTraverse(Ray ray)
{
    (hit, t_near, t_far) = RayBoxIntersection(ray, scene_box);

    if (!hit || t_near > t_far)
        return false;

    if (t_near < 0)
        t_near = 0;
```

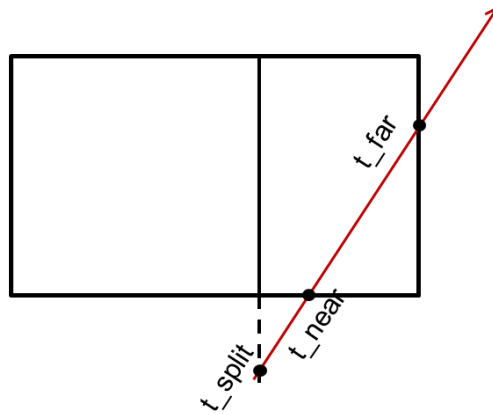


Рис. 21: Поиск в kd-дереве. ($t_{split} < t_{near}$).

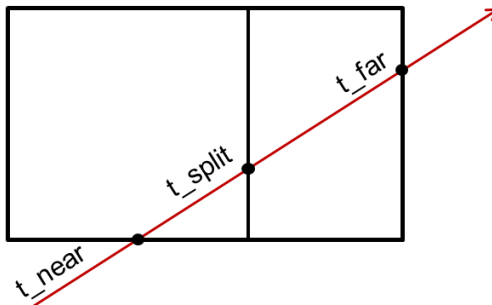


Рис. 22: Поиск в kd-дереве. ($t_{near} \leq t_{split} \leq t_{far}$).

```
while (true)
{
    while (!node.Leaf())
    {
        axis = node.GetAxis(); // axis = 0..2
        t_split = (node.GetSplitPos() - ray.pos[axis])/ray.dir[axis];

        if (t_split < t_near)
        {
            node = node.NearChild();
        }
        else if (t_split > t_far)
        {
            node = node.FarChild();
        }
        else
        {
            stack.push(node.FarChild(), t_far)
        }
    }
}
```

```

        t_far = t_split;
    }
}

t_hit = IntersectAllPrimitivesInLeaf(ray, node)
if (t_hit <= t_far)
    return true;

if (stack.empty())
    return false;

t_near = t_far;
(node, t_far) = stack.pop();
}
}

```

Listing 14: Поиск в kd-дереве.

Почти все рассмотренные ранее проблемы регулярных сеток kd-дереву успешно решает. Оно обладает простым алгоритмом поиска, хорошей адаптивностью, быстро отбрасывая пустые пространства и значительно амортизирует проблему повторных пересечений. Однако серьезный недостаток kd-деревя - сложный алгоритм построения. Дело здесь не только в необходимости вычисления функции стоимости SAH, но, что гораздо более важно, перед началом построения kd-деревя невозможно сказать заранее, сколько памяти потребуется для его построения. Поскольку при разбиении узла все примитивы теоретически могут попасть как в левый, так и в правый дочерний узел, при разбиении узла с числом примитивов N необходимо иметь $2N$ свободной памяти.

BSP-деревья

В отличие от kd-деревьев, в которых плоскости разбиения имеют всего 3 возможных ориентации (XoY, XoZ, YoZ), в BSP (Binary Space Partion) деревьях плоскости разбиения могут быть ориентированны произвольным образом 23. Это значительно усложняет выбор плоскости с учетом оптимизации SAH (и как следствие сам алгоритм построения), но частично решает проблему обозначенную на рисунке 19. В работе [8] было показано преимущество BSP-деревьев в скорости поиска над kd-деревьями.

Тем не менее, даже BSP дерево не всегда гарантирует уменьшение числа примитивов в дочерних узлах (рис. 23, узел с тремя треугольниками справа вверх). На практике из этого следует, что проблема с неизвестным количеством памяти, требующимся для построения дерева, все еще актуальна для BSP деревьев.

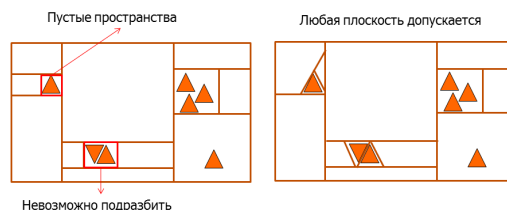


Рис. 23: kd-дерево (слева); BSP-дерево (справа).

BVH-деревья

Этот вид ускоряющей структуры является наиболее практичным и часто используемым в современных трассировщиках лучей. BVH расшифровывается как Bounding Volume Hierarchy - Иерархия Ограничивающих Объемов. BVH-дерево состоит из вложенных друг в друга объемов, ограниченных некоторыми примитивами (рис. 24). Такие деревья можно классифицировать по типу ограничивающего примитива.

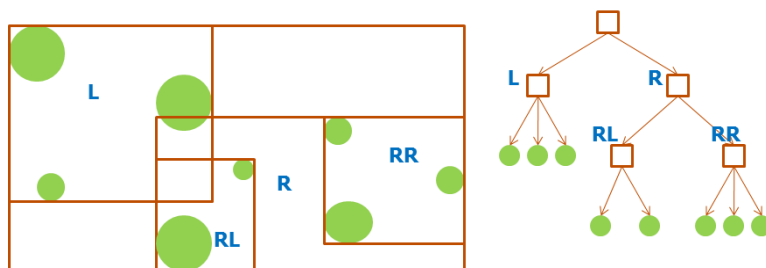


Рис. 24: BVH-дерево. В отличие от других видов пространственных деревьев, узлы дерева могут пересекаться друг с другом.

В дальнейшем мы будем рассматривать только деревья, использующие AABB в качестве ограничивающих примитивов и под BVH деревом понимать дерево состоящее из вложенных друг в друга прямоугольных параллелипипедов, стороны которых выровнены по осям координат.

Две стратегии разбиения в BVH дереве

При построении дерева существует небольшая, но довольно существенная путаница в том, какие примитивы включать в дочерние узлы и как именно производить разбиение. Вопрос, который порождает путаницу звучит так: "Что делать, если примитив пересекает оба дочерних узла?". Существует 2 стратегии разбиения.

Первая стратегия называется разбиением по объектам (рис. 25). При использовании этой стратегии необходимо найти некоторое разбиение на-

шего исходного множества примитивов на 2 подмножества. Затем достаточно лишь посчитать ограничивающий объем для обоих подмножеств. В этом случае ответ на поставленные ранее вопрос звучит так: Такой примитив всегда идет в тот узел, в котором он содержится полностью. Такой узел всегда существует по построению указанного разбиения. Вторая стратегия называется пространственным разбиением (рис. 25). При использовании этой стратегии сначала выбирается некоторая плоскость, которая делит объем текущего узла на 2 части. Для полученных дочерних объемов ограничивающие AABV пересчитываются в соответствии с геометрией. В этом случае ответ на поставленный вопрос звучит так: "Такой примитив нужно добавлять в каждый из дочерних узлов, который он пересекает".

Рассмотренные стратегии порождают 2 принципиально разных вида BVH дерева. Стратегия разбиения по объектам позволяет строить такое дерево, в котором для каждого объекта хранится не более 1 ссылки. Мы будем называть такие деревья деревьями с одиночными ссылками (single reference trees).

В противоположность этому, стратегия пространственного разбиения может породить несколько ссылок на 1 объект. Мы будем называть такие деревья деревьями со множественными ссылками (multiple reference trees).

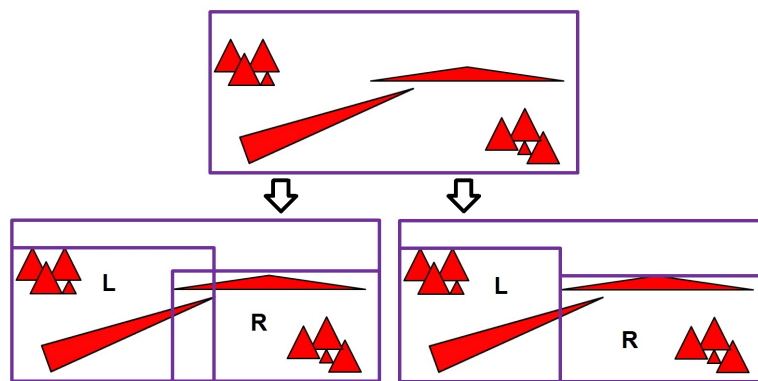


Рис. 25: Разбиение по объектам (слева) и пространственное разбиение (справа). Тонкий длинный треугольник справа при стратегии разбиения по объектам попадет только в левый узел L. А при стратегии пространственного разбиения он попадет в оба дочерних узла L и R.

BVH деревья с одиночными ссылками позволяют значительно упростить алгоритм построения при относительно небольшой потере в эффективности поиска. В дальнейшем под BVH деревом мы будем понимать BVH

дерево с одиночными ссылками, построенными при помощи стратегии разбиения по объектам.

Построение BVH дерева.

Алгоритм построения оптимального BVH дерева столь-же сложен, сколь и алгоритм построения оптимального kd-дерева. В теории даже сложнее. Когда мы строим kd-дерево, то на каждом шаге подразбиения минимизируем функцию стоимости SAH. В случае kd-дерева мы выбираем только плоскость разбиения и можем перебрать все возможные границы примитивов по трем измерениям (обычно SAH вычисляют на границах), чтобы найти такую плоскость, в которой функция SAH будет минимальна. То есть в худшем случае, при отсутствии оптимизаций нам нужно вычислить SAH лишь $6 \cdot N$ раз (где N - число примитивов) и найти минимум. В случае BVH всё сложнее, поскольку вариантов возможных разбиений не $6 \cdot N$ а $S(N, 2) = 2^{N-1} - 1$ (где S - число Стирлинга 2 рода). Перебрать их все не представляется возможным. В отличие от kd-дерева, BVH на каждом уровне вообще говоря может произвести разбиение на 2 совершенно произвольных подмножества примитивов. Всего таких подмножеств $2^{N-1} - 1$.

Однако можно использовать упрощение, не сильно ухудшающее качество дерева, но позволяющее значительно упростить процедуру его построения. Будем строить BVH дерево сверху вниз. При построении, в каждом узле дерева сортируем примитивы по минимумам (или центрам) их AABV и 3 осям координат. Таким образом, мы получаем 3 массива. Первый отсортирован по X координатам минимумов (или центров) AABV объектов, второй по их Y координатам и третий по их Z координатам. Затем для каждого из полученных массивов (длинной n) мы перебираем все разбиения на подмножества вида:

$$\begin{aligned} &[0; 1..n] \\ &[0..1; 2..n] \\ &[0..3; 4..n] \\ &\dots \\ &[0..n-1; n] \end{aligned}$$

Для каждого из этих разбиений вычисляем значение SAH. Запоминая ограничивающие AABV в специальном массиве для получаемых подмножеств (rightBounds) мы можем не пересчитывать эти AABV полностью,

что сводит наш поиск минимума SAH всего к 2 проходам по массиву. С конца к началу и с начала к концу. Ниже приведен фрагмент исходного кода описанного алгоритма.

```
SplitData FindObjectSplit(PrimitiveList a_plist, AABB3f a_box)
{
    struct BoxLessMax
    {
        BoxLessMax(int a) { axis=a; }
        bool operator()(const PrimitiveRef& p1, const PrimitiveRef& p2) const
        { return (p1.Box().vmax[axis] < p2.Box().vmax[axis]); }
        int axis;
    };

    PrimitiveList plist[3] = {a_plist, a_plist, a_plist};

    BVHTree::SplitData res;

    res.sah = SA(a_box)*a_plist.size(); // SA is SurfaceArea
    res.subdivideNext = false;
    res.axis = -1;
    std::vector<AABB3f> rightBounds(a_plist.size());

    for (int dim=0;dim<3;dim++)
    {
        // sort data according to axis
        //
        PrimitiveList& plist = *plist[dim];
        std::sort(plist.begin(), plist.end(), BoxLessMax(dim));

        // Sweep right to left and determine bounds.
        //
        AABB3f rightBounds1;
        for (uint i = plist.size() - 1; i > 0; i--)
        {
            rightBounds1.include(plist[i].Box());
            rightBounds[i-1] = rightBounds1;
        }

        // Sweep left to right and select lowest SAH.
        //
        AABB3f leftBounds;

        for (uint i = 1; i < plist.size(); i++)
        {
            leftBounds.include(plist[i-1].Box());
            float CE = CostEmpty
```

```

        float sah = CE+SA(leftBounds)*i+SA(rightBounds[i-1])*(plist.size()-i);

        if (sah < res.sah)
        {
            res.sah = sah;
            res.axis = dim;
            res.primsOnLeftSide = i;
            res.primsOnRightSide = plist.size() - i;
            res.leftBounds = leftBounds;
            res.rightBounds = rightBounds[i-1];
        }
    }

} // for (int dim=0;dim<3;dim++)

res.subdivideNext = (res.axis!=-1) && (res.sah<SA(a_box)*a_plist.size());

return res;
}

```

Listing 15: Поиск разбиения по объектам при построении BVH дерева.

Описанный способ выбора разбиения позволяет найти близкое к оптимальному решение за линейное время. Алгоритм построения самого дерева приведен ниже:

```

void Subdivide(BVHNode* a_currNode, const AABB3f& a_box,
               int a_currDeep, PrimitiveList& plist)
{
    if(a_currDeep == 0 || plist.size() <= recomendedPrimInLeaf)
    {
        InsertListInLeaf(a_currNode, plist);
        return;
    }

    SplitData split = FindObjectSplit(plist, a_box);

    if(!split.subdivideNext)
    {
        InsertListInLeaf(a_currNode, plist);
        return;
    }

    // split primitives and construct subtrees
    //
    BVHNode (*leftNode, *rightNode) = NewNodePair(a_currNode);

    iterator middle = plist.begin() + split.primsOnLeftSide;

```

```

PrimitiveList leftList(plist.begin(), middle);
PrimitiveList rightList(middle, plist.end());

Subdivide(leftNode, split.leftBounds, a_currDeep-1, leftList);
Subdivide(rightNode, split.rightBounds, a_currDeep-1, rightList);
}

```

Listing 16: Построение BVH дерева.

Раннее подразбиение примитивов.

У BVH дерева использующего AABV в качестве ограничивающих примитивов есть один существенный недостаток, свойственный также и kd-деревьям - треугольники (и некоторые другие объекты) плохо аппроксимируются при помощи прямоугольных параллелипипедов, оставляя внутри ограничивающего примитива большое количество пустого пространства (рис 26). Даже треугольники, лежащие в плоскостях XOY, XOZ или YOZ обладают площадью поверхности, вдвое меньшей чем ограничивающий их AABV. Для всех остальных треугольников это соотношение еще выше.

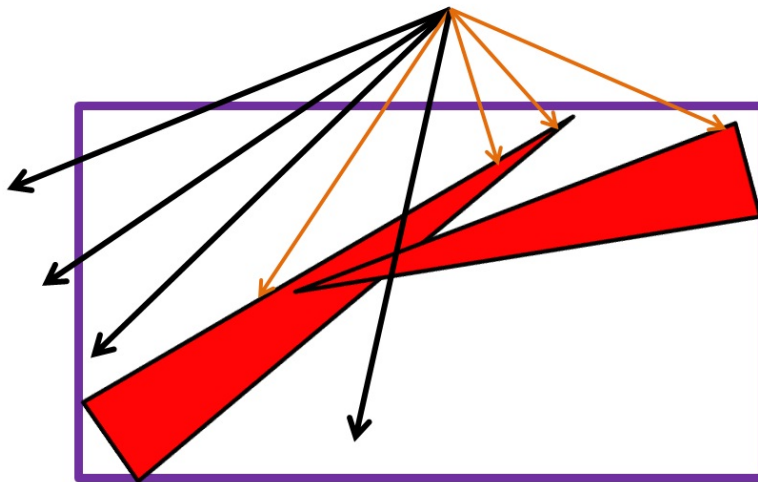


Рис. 26: Лист BVH дерева при отсутствии раннего разбиения. Через ограничивающий AABV проходит значительное количество лучей, которые на самом деле не пересекают ни один треугольник.

На рисунке 26 черным цветом обозначены лучи которые не пересекли ни один треугольник (представьте эту картинку в 3D). Если в геометрии присутствует много длинных и тонких треугольников, получается BVH

с огромным количеством пересекающихся друг с другом узлов, содержащих много пустого пространства.

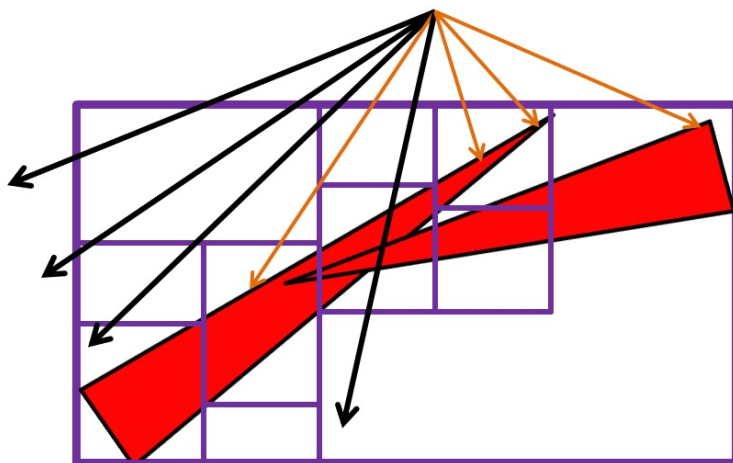


Рис. 27: Раннее подразбиение. Полученное множество AABV подается на вход построителя BVH дерева.

Чтобы побороть этот недостаток, можно подразбивать сами треугольники на более мелкие, но существует более элегантно и экономное в плане памяти решение, называемое ранним подразбиением примитивов (Early Split Clipping) [7]. Идея раннего подразбиения изображена на рисунке 27. Она предлагает представлять треугольник не одним AABV а несколькими. То есть один длинный треугольник подается на вход BVH построителю в виде набора небольших AABV, содержащих ссылки на этот треугольник.

Поск в BVH дереве.

Ниже будет представлен алгоритм поиска пересечений в BVH дереве. Предполагается что дерево вытянуто в виде линейного массива, а узлы в массиве упакованы таким образом, что каждый правый потомок следует непосредственно за левым. Вместо указателей мы будем использовать смещения от начала массива. Такой способ хранения дерева удобен при необходимости его сериализации и реализации поиска на GPU. Помимо этого, данный способ уменьшает количество кэш-промахов при реализации на центральном процессоре, повышая скорость поиска.

```
struct BVHNode
{
    int    leftChildOffset;
```

```

    bool isLeaf;
    AABB3f box;
};

```

Listing 17: Узел BVH дерева.

Рассмотрим представление BVH узла на литинге 17. Если узел не является листом, поле `leftChildOffset` следует трактовать как смещение до левого потомка. Смещение до правого потомка при этом вычисляется как `leftChildOffset+1`. Если узел является листом, `leftChildOffset` следует трактовать как смещение в массиве списков примитивов, некий указатель передаваемый в функцию `IntersectAllPrimitivesInLeaf`. Для сокращения записи будем считать присваивание вида `'(leftNodeOffset, isLeaf) = node'` автоматической распаковкой структуры `BVHNode` в кортеж из 2 переменных в соответствии с порядком объявления полей в структуре `BVHNode`. То есть `leftNodeOffset = node.leftChildOffset` и `isLeaf = node.isLeaf`. Поле `'box'` при этом не используется.

```

Hit BVHTraversal(Ray ray)
{
    Hit hit;

    int top=0;
    int leftNodeOffset = 1;
    bool isLeaf = false;

    bool searchingForLeaf = true;

    while(top >= 0)
    {
        while(searchingForLeaf)
        {
            float t_minLeft    = 0.0f;
            float t_maxLeft    = 0.0f;
            float t_minRight   = 0.0f;
            float t_maxRight   = 0.0f;

            BVHNode leftNode   = GetBVHNode(leftNodeOffset);
            BVHNode rightNode  = GetBVHNode(leftNodeOffset+1);

            bool traverseChild0 = RayBoxTest(ray, leftNode, &t_minLeft, &t_maxLeft);
            bool traverseChild1 = RayBoxTest(ray, rightNode, &t_minRight, &t_maxRight);

            traverseChild0 = traverseChild0 && (t_maxLeft >= t_rayMin) \
                                && (t_minLeft <= hit.t);

            traverseChild1 = traverseChild1 && (t_maxRight >= t_rayMin) \
                                && (t_minRight <= hit.t);
        }
    }
}

```

```

    // traversal decision
    //
    (leftNodeOffset, isLeaf) = traverseChild0 ? leftNode : rightNode;

    if(traverseChild0 && traverseChild1)
    {
        bool l = (t_minLeft <= t_minRight);
        (leftNodeOffset, isLeaf) = l ? leftNode : rightNode;
        stack[top] = l ? rightNode : leftNode;
        top++;
    }

    if(!traverseChild0 && !traverseChild1) // both miss, stack.pop()
    {
        top--;
        (leftNodeOffset, isLeaf) = stack[top];
    }

    searchingForLeaf = !isLeaf && (top >= 0);
}

if(isLeaf && top >= 0)
{
    IntersectAllPrimitivesInLeaf(ray, leftNodeOffset, t_rayMin, &hit);
}

top--;
(leftNodeOffset, isLeaf) = stack[top];

searchingForLeaf = !isLeaf;
}

return hit;
}

```

Listing 18: Поиск пересечений в BVH дереве.

Резюме по ускоряющим структурам

В сочетании с ранним подразбиением, BVH дерево на практике показало себя с наилучшей стороны [9]. Алгоритм построения BVH достаточно прост и занимает определенный объем памяти, пропорционально количеству входных примитивов.

0.2 Проблема глобальной освещенности

Один из способов получения фотореалистичных изображений - моделирование оптических процессов с целью точного вычисления освещенности каждой точки сцены. Трассировка лучей, описанная нами выше, позволила бы рассчитать изображение корректно, если бы поверхности объектов реального мира были гладкими и отражали свет строго по закону "угол падения равен углу отражения". Однако, из-за наличия микрорельефа поверхностей это не так, и свет приходящий строго с одного направления, распределяется по полусфере вокруг нормали к поверхности в соответствии со свойствами материала.

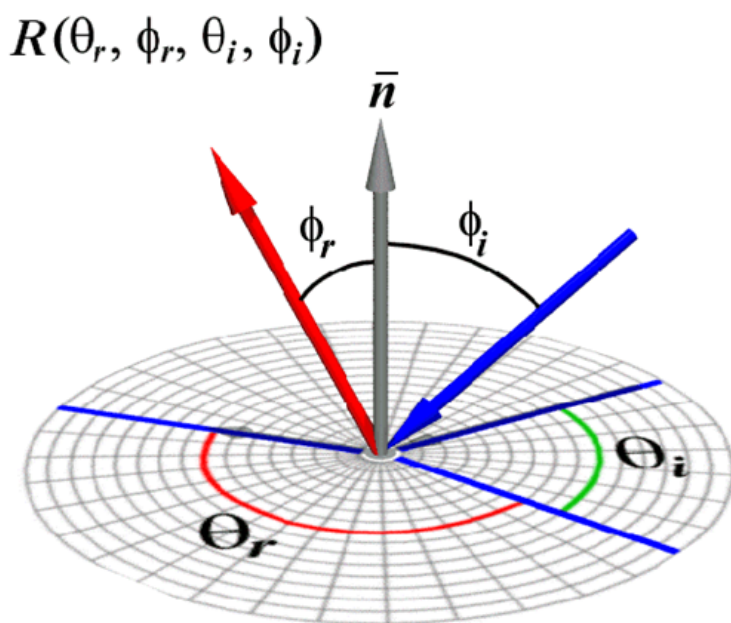


Рис. 28: Двухнаправленная Функция Отражения.

Для моделирования свойств материала вводят Двухнаправленную Функцию Отражения (ДФО) [10]. В английской терминологии обозначается как BRDF (Bidirectional Reflectance Distribution Function). ДФО связывает по некоторому закону интенсивность и угол падающего света с интенсивностью и углом отраженного поверхностью света. Зная ДФО материала, можно вычислить освещенность точки поверхности при помощи интеграла освещенности [11]:

$$I(\phi_r, \theta_r) = \iint_{\phi_i \theta_i} L(\phi_i, \theta_i) R(\phi_i, \theta_i, \phi_r, \theta_r) \cos(n, l_{\phi_i, \theta_i}) d\phi_i d\theta_i \quad (7)$$

В уравнении 7 функция $L(\phi_i, \theta_i)$ задает яркость света, падающего с направления задаваемого вектором l_{ϕ_i, θ_i} . $R(\phi_i, \theta_i, \phi_r, \theta_r)$ - ДФО. n - вектор нормали к поверхности. Вычисляемая интегрированием интенсивность освещения в точке $I(\phi_r, \theta_r)$ является не числом, а функцией. Другими словами, она дает значения интенсивности света, отражаемой поверхностью под разными углами. Таким образом, выполняя интегрирование по полусфере приходящего (падающего) в точку освещения L с учетом свойств поверхности R , мы получаем новую функцию распределения освещения в пространстве I , обусловленную свойствами отражения поверхности (материала) в некоторой точке x . Если бы мы рассматривали конкретный луч (например выпущенный для конкретного пиксела и виртуальной камеры), направление ϕ_r, θ_r было бы фиксированно и задавалось бы этим лучом.

Формула 7 берется из тех соображений, что освещенность в точке поверхности складывается из света, падающего на поверхность со всех направлений и отражающегося в заданном направлении по определенному закону (который задает ДФО). Эта формула не более чем математическая модель и она верна не всегда. Например, в случае стекла нужно учитывать свет, приходящий из под поверхности и проводить интегрирование по полной сфере. В случае кожи ситуация еще сложнее, так как необходимо учитывать такой эффект как подповерхностное рассеивание. Размерность интеграла освещенности (формула 7) зависит от количества учитываемых переотражений света, поскольку функция L в некоторой точке x будет зависеть от функции I в некоторой другой точке y . Из-за большой размерности интеграла для его вычисления обычно используют метод Монте-Карло, поскольку его скорость сходимости не зависит от размерности интеграла.

0.2.1 Метод Монте-Карло

Пусть u - случайная величина, равномерно распределенная на отрезке $[a, b]$. Тогда $f(u)$ - тоже случайная величина. Исходя из определения, ее математическое ожидание равно:

$$Ef(u) = \int_a^b f(x)p(x)dx \quad (8)$$

$$p(x) = \frac{1}{b-a} \quad (9)$$

Где $p(x)$ - плотность вероятности равномерного распределения. Выражая $\int_a^b f(x)dx$ из формулы 8 получаем:

$$\int_a^b f(x)dx = (b-a)Ef(u) \approx \frac{b-a}{N} \sum_{i=1}^N f(u_i) \quad (10)$$

В применении к интегралу освещенности (имеем право взять полусферу единичной площади):

$$I(\phi_r, \theta_r) \approx \frac{\sum_{i=1}^N L(\phi_i, \theta_i) R(\phi_i, \theta_i, \phi_r, \theta_r) \cos(n, l_{\phi_i, \theta_i})}{N} \quad (11)$$

Итак, мы можем оценивать значения интеграла при помощи вычисления суммы достаточно большого числа значений под-интегральной функции [12]. Далее мы познакомимся с некоторыми важными для понимания основного материала определениями математической статистики.

Выборка по значимости (Importance Sampling).

Эффективным способом улучшения сходимости метода Монте-Карло является выборка по значимости (метод существенной выборки, importance sampling) [12]. Данный метод предполагает использование случайной величины не с равномерным распределением, а с таким распределением, которое пропорционально модулю вычисляемой функции. Идея выборки по значимости базируется на том, что некоторые значения случайной величины в процессе моделирования имеют большую значимость для оцениваемой функции, чем другие. Если эти 'более вероятные' значения будут появляться в процессе выбора случайной величины чаще, дисперсия оцениваемой функции уменьшится и скорость сходимости, таким образом, увеличится [12]. При этом, для того чтобы метод давал корректный результат, необходимо учитывать получаемые значения с весами, обратно-пропорциональными вероятностям появления выборки.

При этом, если распределение случайных величин не равномерное, оценка для метода Монте-Карло запишется в более общем виде [12]:

$$\int_a^b f(x)dx \approx \frac{1}{N} \sum_{i=1}^N \frac{f(u_i)}{p(u_i)} \quad (12)$$

где $p(u_i)$ - функция плотности вероятности случайной величины u .

Смещенные и несмещенные оценки и методы.

В данном разделе мы кратко рассмотрим понятие смещенных и несмещенных оценок в применении к вычислению интеграла освещенности чтобы пояснить такие часто встречающиеся англо-язычные термины как *biased renderer* и *unbiased renderer*.

Пусть $X_1..X_n$ случайная выборка из распределения, зависящего от параметра $\theta \in \Theta$

Статистикой называется любая измеримая функция от выборки θ_i , не зависящая от θ (важно, что значение статистики можно вычислить при каждой реализации выборки, не зная значения θ). В случае метода Монте-Карло под-интегральное выражение может называться статистикой.

Точечной оценкой называется любая статистика, принимающая некоторые значения на области определения Θ .

Оценка называется несмещенной (*unbiased*) если ее математическое ожидание равно оцениваемому параметру. В противном случае оценка называется смещенной (*biased*).

Оценка называется консистентной если она сходится по вероятности (сходимость почти наверное) к оцениваемому параметру.

Будем называть метод или алгоритм вычисления интеграла освещенности несмещенным, если он позволяет получить несмещенную оценку. В противном случае будем называть метод смещенным.

Будем называть метод вычисления интеграла освещенности консистентным, если он позволяет получить консистентную оценку.

Таким образом, если программное обеспечение для вычисления освещенности и фотореалистичной визуализации (англ. *renderer*) использует в том или ином виде метод Монте-Карло и при этом позволяет при бесконечно-большом времени расчета получить абсолютно точное решение, говорят что это '*unbiased renderer*'. К таким программам, например, можно отнести все виды трассировщиков путей. Если программа использует метод Монте-Карло, но при этом в пределе (при бесконечно-большом времени расчета) не позволяет получить абсолютно точное решение, говорят, что такой рендерер является '*biased*' (например программы, использующие фотонные карты и/или кэш освещенности). К программам, не использующим метод Монте-Карло понятия *biased/unbiased*

неприменимы. Например, некорректно использовать указанные термины по отношению к программе, использующей метод излучательности. Смещенность алгоритма еще не означает, что он вычислит интеграл с низкой точностью. На практике смещение (bias) проявляется в виде цветных пятен. Если смещение небольшое, пятна незаметны для глаза. Как правило, смещенные методы на начальном этапе работы (при небольшом числе выборок или небольшом времени работы) дают более приемлимую для человеческого глаза оценку, чем несмещенные. Однако, в пределе, при длительном расчете несмещенные методы обычно дают более приемлимое изображение.

0.3 Стохастическая трассировка лучей

Стохастическая трассировка лучей (также называемая Монте-Карло трассировкой) вычисляет значение интеграла освещенности (для фиксированного направления ϕ_r, θ_r) напрямую по формуле 11. Метод был предложен James-ом Кэджи в 1986 году [11].

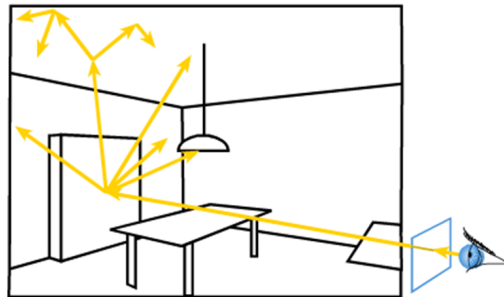


Рис. 29: Иллюстрация монте-карло трассировки лучей.

Из виртуальной камеры испускается луч, который трассируется в сцену. В точке пересечения с поверхностью выпускается некоторое число лучей по полусфере и для каждого луча процедура выполняется рекурсивно (рис. 29). Полученные значения яркости на каждом уровне рекурсии складываются с учетом формулы 11.

В дальнейшем мы будем рассматривать модификацию стохастической трассировки лучей, при которой отраженный луч всегда один (рис 30). Такое упрощение немного замедляет сходимость метода, но позволяет упростить реализацию и при желании избавиться от рекурсии в реализации алгоритма.

0.4 Трассировка путей

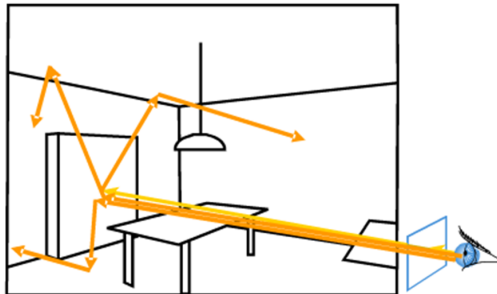


Рис. 30: Иллюстрация монте-карло трассировки путей.

Для начала будем считать, что в сцене все источники света имеют ненулевой размер и заданы в виде геометрических объектов, поверхность которых излучает свет. Тогда следующий алгоритм позволит вычислить интеграл освещенности:

```
Color TracePath(Ray r, int depth)
{
    if (depth == MaxDepth)
        return Black; // Bounced enough times.

    Hit hit = RaySceneIntersection(ray);

    if (!hit.exist)
        return Black; // Nothing was hit.

    Material m = hit.material;
    if (m.isLight())
        return m.emittance;

    // Pick a random direction from here and keep going.
    // This is NOT a cosine-weighted distribution!
    //
    Ray newRay;
    newRay.origin      = hit.pos; // + hit.norm*epsilon
    newRay.direction = RandomUnitVectorInHemisphereOf(hit.norm);

    // Compute the BRDF for this ray (assuming Lambertian reflection)
    //
    float cos_theta = dot(newRay.dir, hit.norm);

    Color BRDF = m.reflectance * cos_theta;
    return BRDF * TracePath(newRay, depth + 1);
}
```

```
}
```

Listing 19: Простейший вид Монте-Карло трассировки путей. В данном псевдокоде `emittance` обозначает светимость и измеряется в ваттах на квадратный метр.

Данный алгоритм также называют обратной трассировкой путей. Для генерации случайного вектора по полусфере Вам может быть полезна функция, отображающая случайные числа с равномерным распределением из интервала $[0,1]$ на сферу [13]:

```
float3 MapSampleToSphere(float r1, float r2) // [0..1]
{
    float phi = r1*3.141592654f*2; // [0..2*PI]
    float h   = r2*2 - 1;          // [-1..1]

    float sin_phi, cos_phi;
    sincosf(phi, &sin_phi, &cos_phi);

    float x = sin_phi*sqrt(1-h*h);
    float y = cos_phi*sqrt(1-h*h);
    float z = h;

    return float3(x,y,z);
}
```

Listing 20: Генерация случайного вектора по сфере из 2 равномерно-распределенных чисел в интервале от 0 до 1.

0.4.1 Прогрессивное вычисление интеграла

Для того чтобы иметь возможность визуализировать промежуточный результат в процессе расчета изображения, удобно использовать прогрессивную схему вычисления интеграла. Будем полагать, что мы используем простейший бокс-фильтр, усредняющий все пути для каждого пиксела. То есть цвет пиксела вычисляется по следующей формуле:

$$C = \sum_{i=1}^N \frac{P_i}{N} \quad (13)$$

Где P_i - цвет полученный от i -ого пути. Будем вычислять цвет пиксела в виде сходящейся к C последовательности $\{C_i\}$ Пусть далее C_i - цвет пиксела изображения на i -ом проходе. Тогда он может быть вычислен прогрессивно по следующей схеме:

$$\begin{aligned}
C_0 &= P_0 \\
C_1 &= \frac{1}{2}C_0 + \frac{1}{2}P_1 \\
C_2 &= \frac{2}{3}C_1 + \frac{1}{3}P_2 \\
C_3 &= \frac{3}{4}C_2 + \frac{1}{4}P_3 \\
&\dots \\
C_n &= \frac{n}{n+1}C_{n-1} + \frac{1}{n+1}P_n
\end{aligned}$$

Данная схема позволяет пользователю наблюдать постепенно улучшающееся изображение (с постепенно снижающимся уровнем шума). Если пользователя устраивает полученное изображение на шаге i , он может остановить расчет.

Критерий остановки

Для того чтобы автоматически останавливать расчет C_i для каждого пиксела, можно применять следующий подход: Последовательность C_i разбивается на 2 подпоследовательности - $C_{i\text{odd}}$ и $C_{i\text{even}}$. Подпоследовательность $C_{i\text{even}}$ содержит сумму цветов путей для всех четных номеров i , а подпоследовательность $C_{i\text{odd}}$ - для всех нечетных. Поскольку $C_{i\text{odd}}$ и $C_{i\text{even}}$ должны сходиться к одному и тому же значению, остановить вычисление интеграла можно оценивая разность между $C_{i\text{odd}}$ и $C_{i\text{even}}$.

$$\begin{aligned}
C_i &= \frac{(C_0 + C_1 + C_2 + C_3 + C_4 + C_5 + \dots)}{N} \\
C_{i\text{even}} &= \frac{(C_0 + C_2 + C_4 + C_6 + C_8 + C_{10} + \dots)}{N/2} \\
C_{i\text{odd}} &= \frac{(C_1 + C_3 + C_5 + C_7 + C_9 + C_{11} + \dots)}{N/2} \\
Error &= |C_{i\text{even}} - C_{i\text{odd}}| \\
C_i &= \frac{1}{2}(C_{i\text{even}} + C_{i\text{odd}})
\end{aligned}$$

0.4.2 Применение выборки по значимости

Известно, что для Ламбретовых поверхностей наибольший вклад дают лучи близкие к направлению нормали, а наименьший - лучи почти пер-

пендекулярные нормали. Это следует из фрагмента исходного кода

```
float cos_theta = dot(newRay.dir, hit.norm);  
BDRF = m.reflectance * cos_theta;
```

Применим выборку по значимости. Будем генерировать такой вектор отражения, который соответствует косинусоидальному распределению вокруг направления нормали (рис. 31).

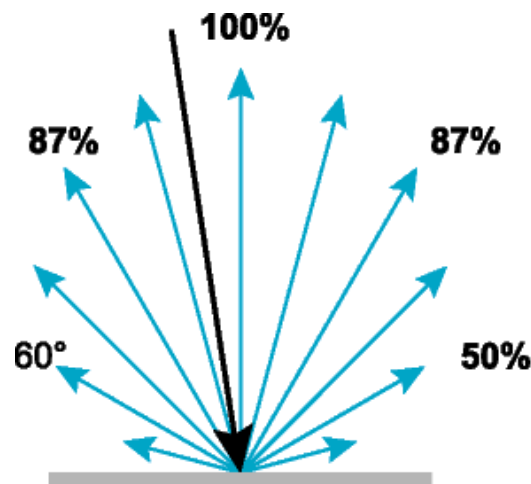


Рис. 31: Выборка по значимости для Ламбертовского отражения.

```
float3 MapSampleToCosineDistribution(float r1, float r2)  
{  
    float e = 1.0;  
    float sin_phi, cos_phi;  
  
    sincosf(2*r1*3.141592654f, &sin_phi, &cos_phi);  
  
    float cos_theta = powf(1.0f-r2, 1.0f/(e+1.0f));  
    float sin_theta = sqrtf(1.0f-cos_theta*cos_theta);  
  
    float x = sin_theta*cos_phi;  
    float y = sin_theta*sin_phi;  
    float z = cos_theta;  
  
    return float3(x,y,z);  
}
```

Listing 21: Генерация вектора с косинусоидальным распределением по нормали [13].

Тогда для того, чтобы результат не изменился, необходимо делить значение BRDF (или приходящей яркости умноженной на BRDF) на плот-

ность вероятности нашего распределения, то есть \cos_theta .

```
Color TracePath(Ray r, int depth)
{
    if (depth == MaxDepth)
        return Black; // Bounced enough times.

    Hit hit = RaySceneIntersection(ray);

    if (!hit.exist)
        return Black; // Nothing was hit.

    Material m = hit.material;
    if (m.isLight())
        return m.emittance;

    // Pick a random direction from here and keep going.
    // This is cosine-weighted distribution
    //
    Ray newRay;
    newRay.origin      = hit.pos; // + hit.norm*epsilon
    newRay.direction = RandomCosineVectorOf(hit.norm);

    Color BRDF = m.reflectance; // cos_theta/cos_theta = 1.
    return BRDF * TracePath(newRay, depth + 1);
}
```

Listing 22: Трассировка путей с учетом косинусоидального распределения отраженных лучей.

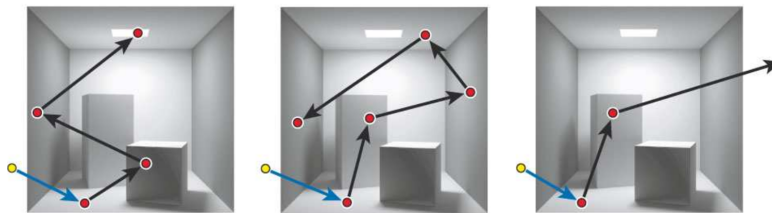


Рис. 32: Иллюстрация алгоритма трассировки путей. Корректный перенос света моделируется путем просчета большого числа путей.

0.4.3 Учет сложных материалов

Ранее мы рассмотрели алгоритм трассировки путей, считая что все материалы в сцене Ламбертовы. Такие материалы отражают свет равномерно

во все стороны. Однако, большинство материалов реального мира обладают намного более сложными ДФО, а также могут не только отражать но и пропускать свет (рис. 33).

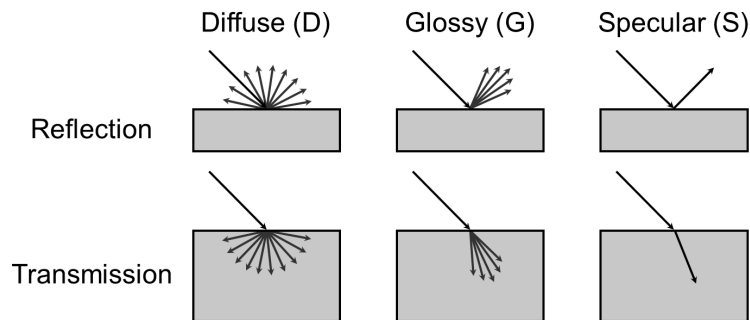


Рис. 33: Различные виды взаимодействий света с материалом. Diffuse(D) - Ламбертовское отражения. Specular(S) - зеркальное. Glossy(G) - матовое.

Идеально-зеркальное(S) и матовое(G) отражения моделируются при помощи единого механизма (мы рассмотрим его ниже), поэтому в дальнейшем мы будем рассматривать взаимодействия G и S совместно и называть такой тип отражения зеркальным (S).

Сложные ДФО часто представляются в виде композиции более простых компонент. Классической композицией является композиция ДФО Ламберта и Фонга (или любой другой ДФО, моделирующей зеркальное и/или матовое (glossy) отражение). В этом случае ДФО получается как линейная комбинация (с коэффициентами k_d и k_s) отдельных компонент. Помимо этого, для прозрачных объектов необходимо учитывать пропускание (по аналогии с ДФО вводят функцию ДПФ или BTDF) с коэффициентом k_t . Для того чтобы учитывать такие композитные свойства материала, используют следующий механизм для выбора отраженного/-преломленного луча:

$$\begin{aligned}
\Sigma &= kd + ks + kt \\
\xi &= rnd(0, 1) \\
\xi \in [0, \frac{kd}{\Sigma}] &\Rightarrow diffuse \\
\xi \in [\frac{kd}{\Sigma}, \frac{kd + ks}{\Sigma}] &\Rightarrow specular \\
\xi \in [\frac{kd + ks}{\Sigma}, 1] &\Rightarrow transmission
\end{aligned}$$

kd - коэффициент диффузного отражения. ks - коэффициент зеркального отражения. kt - коэффициент пропускания. Важно отметить, что при использовании модели выбора компоненты ДФО указанной выше, полученное значение яркости не нужно умножать на значения коэффициентов (kd, ks или kt) т.к. эти коэффициенты получаются стохастически из самой схемы.

Если необходимо учитывать цвет, схема немного усложняется:

$$\begin{aligned}
Pd &= max(kd.r, kd.g, kd.b) \\
Ps &= max(ks.r, ks.g, ks.b) \\
Pt &= max(kt.r, kt.g, kt.b) \\
\Sigma &= Pd + Ps + Pt \\
\xi &= rnd(0, 1) \\
\xi \in [0, \frac{Pd}{\Sigma}] &\Rightarrow diffuse \\
\xi \in [\frac{Pd}{\Sigma}, \frac{Pd + Ps}{\Sigma}] &\Rightarrow specular \\
\xi \in [\frac{Pd + Ps}{\Sigma}, 1] &\Rightarrow transmission
\end{aligned}$$

Однако это еще не все. Теперь полученную яркость необходимо делить на значение плотности вероятности распределения выбранной компоненты. Например, если было выбрано зеркальное отражение, яркость луча необходимо пересчитать по следующей формуле:

$$color.r = color.r / \left(\frac{Ps}{\Sigma}\right)$$

$$color.g = color.g / \left(\frac{Ps}{\Sigma}\right)$$

$$color.b = color.b / \left(\frac{Ps}{\Sigma}\right)$$

Для того чтобы применять выборку по значности с учетом матовых (glossy) ДФО, необходимо генерировать лучи со степенным косинусоидальным распределением в направлении отраженного вектора. Такое распределение представляет собой лепесток (cosine lobe) вокруг вектора отражения [13]:

```
float3 MapSampleToCosineDistribution(float r1, float r2, \
                                     float3 direction, \
                                     float3 hit_norm, \
                                     float power)
{
    float e = power;
    float sin_phi, cos_phi;

    sincosf(2*r1*3.141592654f, &sin_phi, &cos_phi);

    float cos_theta = powf(1.0f-r2, 1.0f/(e+1.0f));
    float sin_theta = sqrtf(1.0f-cos_theta*cos_theta);

    float3 deviation(sin_theta*cos_phi, sin_theta*sin_phi, cos_theta);

    // simple orthogonalisation
    //
    float3 ny = direction;
    float3 nx = normalize(cross(ny, float3(1.04f, 2.93f, -0.6234f)));
    float3 nz = normalize(cross(nx, ny));
    swap(ny, nz); // depends on the coordinate system

    float3 res = nx*deviation.x + ny*deviation.y + nz*deviation.z;

    float invSign = dot(direction, hit_norm) > 0.0f ? 1.0f : -1.0f;

    // resulting vector is below surface ?
    //
    if(invSign*dot(res, hit_norm) < 0.0f)
        res = -nx*deviation.x + ny*deviation.y - nz*deviation.z;

    return res;
}
```

```
}
```

Listing 23: Генерация вектора со степенным косинусоидальным распределением вокруг вектора `direction` [13].

В приведенном выше листинге использовался некоторый произвольный вектор $(1.04f, 2.93f, -0.6234f)$ для реализации ортогонализации Грамма-Шмидта. Более правильный вариант вычисления перпендикуляра представлен ниже:

```
float3 GetPerpendicular(float3 a_vec) const
{
    float3 leastPerpendicular;
    float bestProjection;
    float xProjection = fabs(a_vec.x);
    float yProjection = fabs(a_vec.y);
    float zProjection = fabs(a_vec.z);

    if((xProjection <= yProjection + 1e-5) && \
        (xProjection <= zProjection + 1e-5))
    {
        leastPerpendicular = float3(1, 0, 0);
        bestProjection = xProjection;
    }
    else if((yProjection < xProjection + 1e-5) && \
        (yProjection <= zProjection + 1e-5))
    {
        leastPerpendicular = float3(0, 1, 0);
        bestProjection = yProjection;
    }
    else
    {
        leastPerpendicular = float3(0, 0, 1);
        bestProjection = zProjection;
    }

    return normalize(cross(a_vec, leastPerpendicular));
}
```

Listing 24: Устойчивая ортогонализация Грамма-Шмидта.

Результат работы алгоритма представлен на рисунке 34. Алгоритм позволяет корректно вычислить интеграл освещенности и получить весь спектр эффектов геометрической оптики (рис. 34).

0.4.4 Теневые лучи

Представленный ранее алгоритм работает чрезвычайно долго, причем его скорость сильно зависит от размера источника освещения; алгоритм

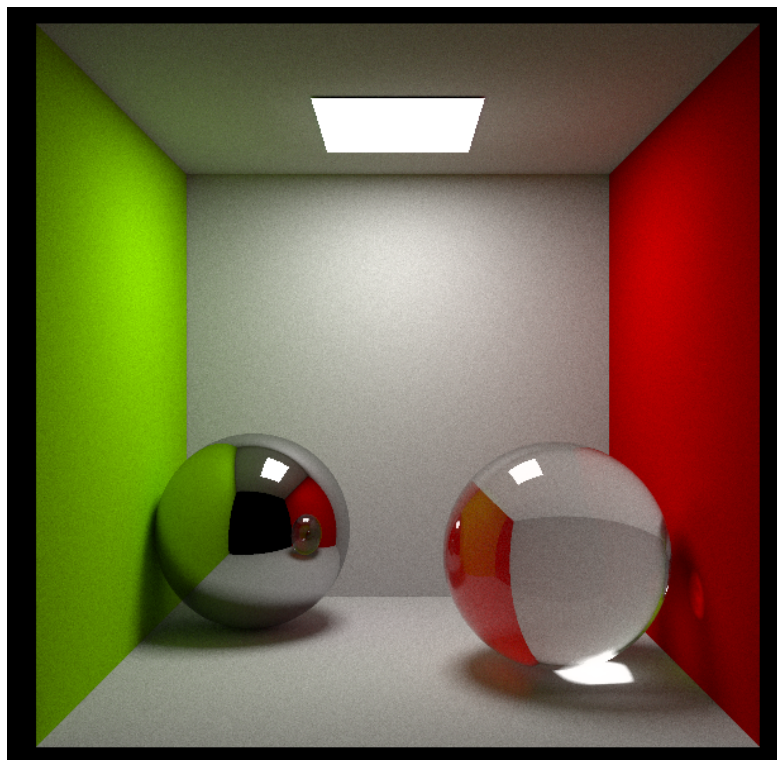


Рис. 34: Пример работы алгоритма трассировки путей на классической сцене Cornell Box.

сходится тем дольше, чем меньше размер источника. Причина такой зависимости заключается в том, что вероятность случайного луча попасть в источник света зависит от его размера и расстояния до источника. Из этого следует, что трассировка путей в том виде, в котором она была рассмотрена нами ранее, не применима для сцен с точечными источниками. Чтобы устранить зависимость скорости сходимости от размера источника и добавить поддержку точечных источников света, вводят так называемые теневые лучи. Идея заключается в том, чтобы сэмплировать не только ДФО (генерировать выборки в соответствие с ДФО), создавая случайные отраженные лучи по полусфере, но также и источники света, генерируя теневые лучи (рис. 35).

При реализации теневых лучей следует иметь ввиду 2 важных момента:

1. Чтобы не учитывать источник света 2 раза, случайные лучи, полученные в результате сэмплирования ДФО при попадании в источник света должны терминироваться и возвращать черный цвет.
2. Если для случайного луча вероятность попасть в источник света

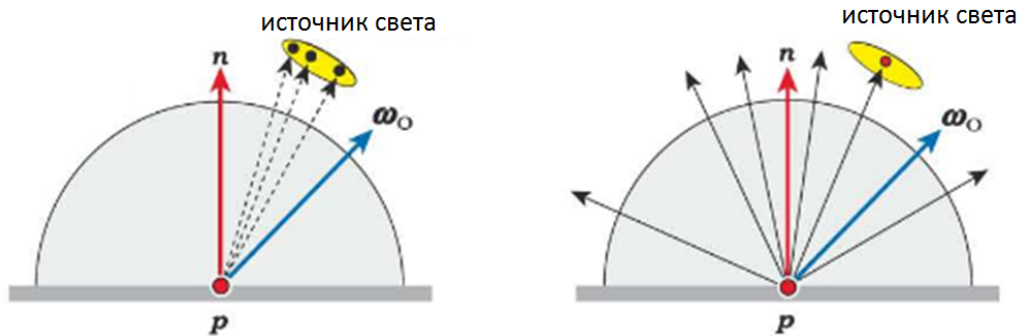


Рис. 35: Сэмплирование (генерация выборок) по ДФО (справа) и по источнику света (слева).

зависит от его площади поверхности и расстояния до него, то для теневых лучей вероятность попадания луча в источник света равна единице (поскольку вы всегда испускаете лучи точно в источник). Для того чтобы корректно применить теневые лучи, необходимо вычислять такое же значение яркости теневого луча, как если бы мы попали в источник случайно, сэмплируя ДФО. Для этого вводится переменная *implicitP*. Она отражает вероятность попадания в источник света при использовании неявной стратегии. R - расстояние от точки, из которой испускается теневой луч до источника света; *light.area* - площадь поверхности источника; *emittance* - ранее использованная светимость поверхности источника света (измеряется в ваттах на квадратный метр).

```
Color TracePath(Ray r, int depth)
{
    if (depth == MaxDepth)
        return Black; // Bounced enough times.

    Hit hit = RaySceneIntersection(ray);

    if (!hit.exist)
        return Black; // Nothing was hit.

    Material m = hit.material;
    if (m.isLight())
        return Black; // different from simple path tracing

    // create shadow ray,
    // eval brdf and compute visibility
    //
```

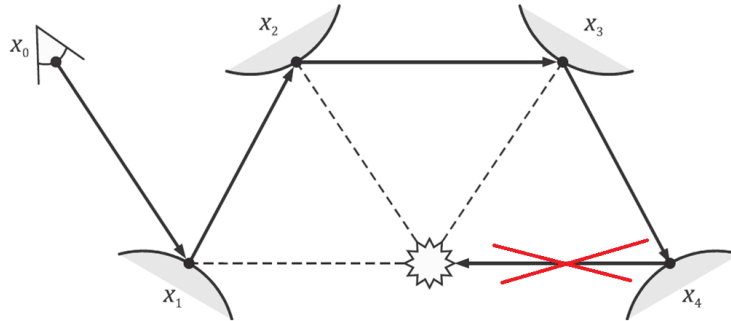


Рис. 36: Теневые лучи обозначены пунктиром. Отраженный луч, попавший в источник зачеркнут перекрестной красной линией. При возникновении такие лучи терминируются с тем чтобы не учитывать источник 2 раза.

```
float3 lpos = LightSample(light);
float shadow = Visibility(hit.pos, lpos);

float R = dist(hit.pos, lpos);
float3 sdir = normalize(lpos - hit.pos);

// calc implicit probability
//
float cos_theta1 = -dot(sdir, light.norm);
float cos_theta2 = dot(sdir, hit.norm);
float PI = 3.1415926535f;
float implicitP = light.area*cos_theta1*cos_theta2/(PI*R*R);

// get brdf value
Color sbrdf = m.reflectance;

// summ up all together
Color explicitColor = shadow*sbrdf*light.emittance*implicitP;

// Pick a random direction from here and keep going.
// This is cosine-weighted distribution
//
Ray newRay;
newRay.origin = hit.pos; // + hit.norm*epsilon
newRay.direction = RandomCosineVectorOf(hit.norm);

Color BRDF = m.reflectance; // cos_theta/cos_theta = 1.
return explicitColor + BRDF * TracePath(newRay, depth + 1);
```

}

Listing 25: Трассировка путей с теньевыми лучами.

Коэффициент *implicitP* получается при переходе от рассмотренной ранее сферической формы уравнения освещенности (уравнение 7) к площадной форме этого уравнения (рис 37, соотношение 14).

$$I(x, \psi) = \int_{\psi'} R(x, \psi') L(x, \psi') \frac{dA' \cos \theta \cos \theta'}{\|x - x'\|} \quad (14)$$

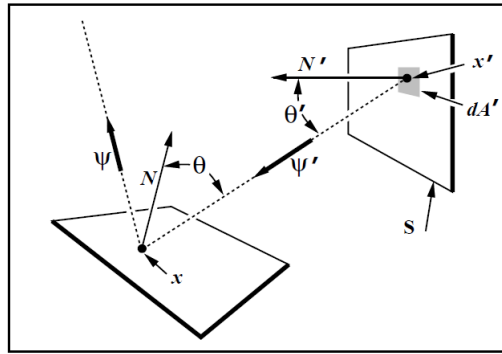


Рис. 37: Иллюстрация к площадной форме интеграла освещенности [14].

На рисунке 36 изображена ситуация, при которой случайный отраженный луч попадает в источник света. Такой луч необходимо терминировать и вернуть 0 в качестве яркости от источника света.

Реализация трассировки путей с теньевыми лучами значительно быстрее сходится (рис. 39) и позволяет обрабатывать источники любых размеров. Однако, как нетрудно заметить (рис. 38), источник света стал черным и на изображении исчезают каустики. Первую проблему (черный источник света) легко исправить если возвращать цвет источника вместо черного при отсутствии диффузных и матовых (glossy) переотражений. Это всегда можно делать поскольку теньевые лучи применяются только для учета матовой и диффузной компонент ДФО.

Вторая проблема - отсутствие каутик. Чтобы вернуть каустики, определим для начала в каких условиях они возникают. Каустик - яркое пятно на диффузной поверхности, вызванное непрямым освещением посредством зеркальных отражений (или преломлений) света, излученного из источника. Таким образом, если в процессе трассировки путей после диффузного переотражения встретилось зеркальное, на месте упомянутого диффузного отражения может возникнуть каустик. Добавить каустики

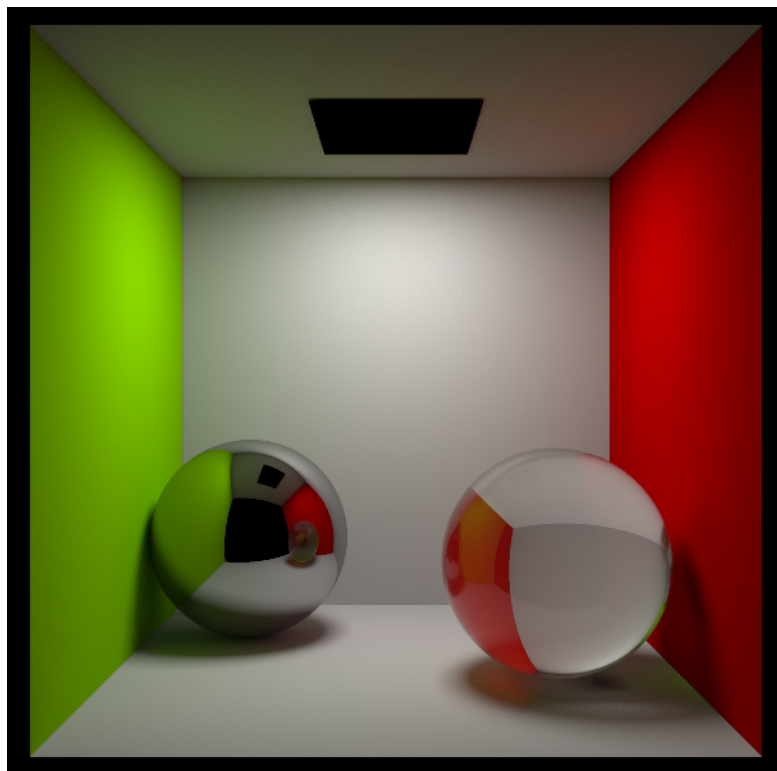


Рис. 38: Пример работы алгоритма трассировки путей с теневыми лучами (из листинга 25) на классической сцене Cornell Box.

в трассировку путей с теневыми лучами, таким образом, достаточно просто:

1. Трассируем путь из глаза с применением теневых лучей до тех пор пока не встретилось зеркальное переотражение после диффузного.
2. Как только указанное условие выполнилось, превращаем оставшуюся часть пути в 'каустический путь'. Такой путь не использует теневые лучи и пытается поймать источник света по простому алгоритму, описанному ранее.

В результате получем изображение на рисунке 34.

Рисунок 39 показывает преимущество трассировки путей с теневыми лучами над простой трассировкой путей, рассмотренной ранее. Поскольку трассировка путей без теневых лучей быстрее обрабатывает пути быстрее (примерно в 2 раза), корректно сравнивать 64 пути на пиксел для трассировки путей без теневых лучей и 32 пути на пиксел с теневыми лучами.

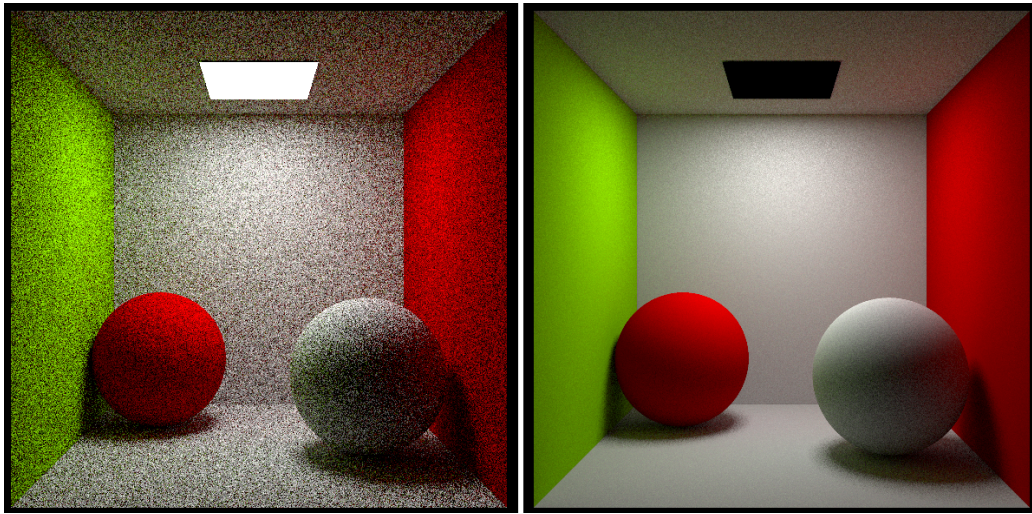


Рис. 39: Сравнение простого алгоритма трассировки путей (64 пути на пиксел, слева на изображении) и алгоритма трассировки путей использующего теневые лучи (32 пути на пиксел, справа на изображении). Обе картанки получены за примерно одинаковое время.

0.4.5 Две стратегии сэмплирования

В предыдущем разделе мы познакомились с двумя стратегиями сэмплирования (стратегиями генерации выборок) - сэмплирование по ДФО и сэмплирование по источнику света (рис. 35). Будем называть сэмплирование по ДФО неявной стратегией, а сэмплирование по источнику - явной.

Для того чтобы иметь корректный результат, мы могли использовать явную и неявную стратегии (превращая оставшуюся часть пути в "каустический" путь), но не могли смешивать их. То есть если сгенерированный по неявной стратегии путь попадал в источник света из той-же точки, в которой уже был учтен теневой луч по явной стратегии, неявный луч приходилось исключать из расчетов. Однако, в этом случае эффективность расчета падает, особенно если неявных лучей попадающих в источник много. Далее мы рассмотрим, как можно смешивать явную и неявную стратегии чтобы учитывать их одновременно и избежать избыточных вычислений.

0.4.6 Многократная выборка по значимости

Для того чтобы корректно комбинировать явную и неявную стратегии, Эриком Вичем был предложен метод многократной выборки по значи-

мости (англ. Multiple Importance Sampling, MIS) [15], который позволяет скомбинировать результаты нескольких способов выборки (нескольких стратегий сэмплирования) в одну несмещенную оценку. Для обратной трассировки путей, ламбертовского отражения и площадного источника света с равномерным распределением, схема многократной выборки по значимости будет иметь следующий вид:

$$\begin{aligned}
 p_i &= \cos(\theta)/\pi \\
 p_e &= 1/A \\
 w_e &= \frac{p_e^\beta}{p_i^\beta + p_e^\beta} \\
 w_i &= \frac{p_i^\beta}{p_i^\beta + p_e^\beta} \\
 \beta &= 2
 \end{aligned}$$

θ - угол между нормалью и отраженным лучом. p_i - плотность вероятности для неявной стратегии. Обратите внимание: в данной формуле считается, что сэмплирование ДФО было произведено при помощи косинусоидального распределения для ламбертовского отражения (в формуле для p_i в знаменателе стоит косинус в первой степени). Если это не так, то p_i нужно вычислять исходя из того, как выбирался случайный отраженный луч. p_e - плотность вероятности для явной стратегии. A - площадь поверхности источника света. w_i - вес для учета яркости полученной по неявной стратегии. w_e - вес для учета яркости полученной по явной стратегии. Обратите внимания, что p_e зависит от площади источника света, которая, в свою очередь, зависит от глобального масштаба сцены. Результат будет корректен если вклад теневого луча L_e учитывается как было описано ранее с использованием вероятности *implicitP*. В этом случае, изменение масштаба не повлияет на результат, поскольку площадь сократится в формуле 15.

Для того чтобы реализовать многократную выборку по значимости нужно отдельно хранить яркость пришедшую от источника света в направлении теневого луча (L_i) и в случае если отраженный луч попал в источник света (и дал яркость L_e) результирующая яркость для данного k -ого сэмпла (выборки) должна быть вычислена по формуле 15:

$$L_k = w_e \frac{L_e}{p_e} + w_i \frac{L_i}{p_i} \quad (15)$$

Оптимальность такого взвешивания показана в [15]. В листинге 26 представлен алгоритм трассировки путей с применением многократной выборки по значимости (MIS). На рис 40 показано сравнение рассмотренной ранее простой трассировки путей и трассировки путей с применением MIS. В действительности, на сцене из рисунка 40 комбинирование стратегий (то есть учет либо, одной либо другой стратегии, но не обеих сразу), дало бы практически неотличимый от правой картинки (на рис. 40) изображение. Ситуация, в которой многократная выборка по значимости действительно дает существенный выигрыш изображена на рис 41.

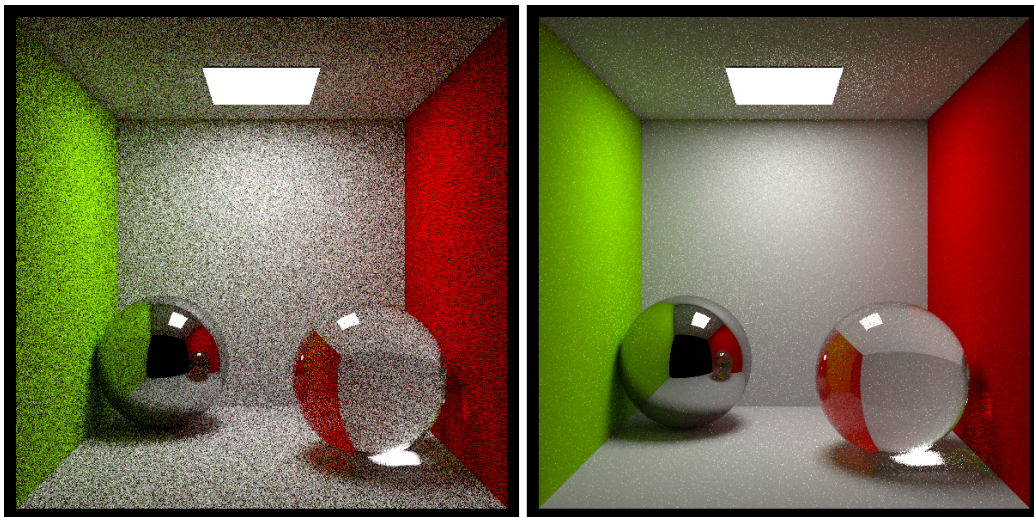


Рис. 40: Сравнение простого алгоритма трассировки путей (64 пути на пиксел, слева на изображении) и алгоритма трассировки путей использующего многократную выборку по значимости (32 пути на пиксел, справа на изображении). Обе картинки получены за примерно одинаковое время.

```
struct RESULT
{
    Color color;
    bool hitLight;
};

RESULT TracePath(Ray r, int depth)
{
    if (depth == MaxDepth)
        return RESULT(Black, false);

    Hit hit = RaySceneIntersection(ray);
```

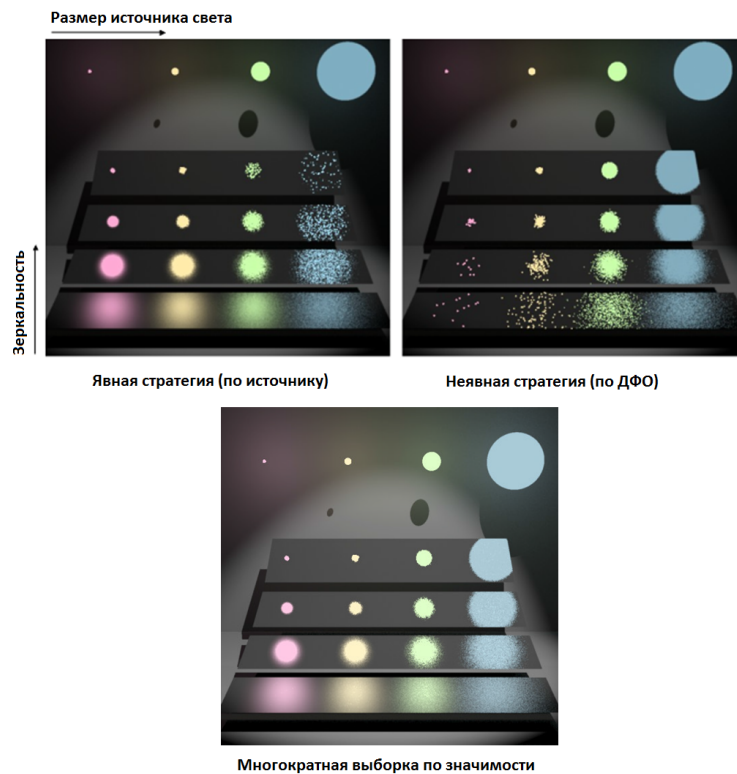


Рис. 41: Сравнение различных стратегий генерации выборки для сцены с матовым (glossy) отражением. Явная стратегия дает лучший результат, если источник находится далеко от поверхности или имеет малый размер. Неявная стратегия дает лучший результат, если источник имеет большой размер или находится близко к поверхности. Многократная выборка по значимости использует преимущества обоих подходов.

```

if (!hit.exist)
    return RESULT(Black, false);

Material m = hit.material;
if (m.isLight())
    return RESULT(m.emittance, true);

// sample explicit
//
float3 lpos = LightSample(light);
float shadow = Visibility(hit.pos, lpos);

float R = dist(hit.pos, lpos);
float3 sdir = normalize(lpos - hit.pos);

```

```

float cos_theta1 = -dot(sdir, light.norm);
float cos_theta2 = dot(sdir, hit.norm);
float PI = 3.1415926535f;
float implicitP = light.area*cos_theta1*cos_theta2/(PI*R*R);

Color sbrdf = m.reflectance;
Color explicitColor = shadow*sbrdf*light.emittance*implicitP;

// sample implicit
//
Ray newRay;
newRay.origin = hit.pos; // + hit.norm*epsilon
newRay.direction = RandomCosineVectorOf(hit.norm);

PRESULT res = TracePath(newRay, depth + 1);

Color BRDF = m.reflectance; // cos_theta/cos_theta = 1.
Color implicitColor = BRDF * res.color;

Color result;

if(res.hitLight)
{
    float pi := 1.0/light.surfaceArea;
    float pe := 3.14159/fmaxf(dot(nextRay.dir, hit.norm),1.0e-5);

    if(not pure specular bounce)
    {
        wi = sqr(pi)/(sqr(pe) + sqr(pi));
        we = sqr(pe)/(sqr(pe) + sqr(pi));

        result = implicitColor*(wi/pi) + explicitColor*(we/pe);
    }
    else
        result = implicitColor;
}
else
    result = implicitColor + explicitColor;

return result;
}

```

Listing 26: Трассировка путей с многократной выборкой по значимости.

0.4.7 Резюме по обратной трассировке путей

Рассмотренный алгоритм обратной трассировки путей с применением теневых лучей и многократной выборки по значимости является несмещенным и достаточно эффективным методом для вычисления интеграла освещенности; за исключением каустик, которые вычисляются этим алгоритмом довольно медленно. Для эффективного вычисления каустик рекомендуется рассмотреть алгоритмы прямой трассировки путей (Light Tracing) и алгоритм фотонных карт, которые будут описаны далее.

0.4.8 Грамматика путей

Для классификации различных ситуаций, возникающих в процессе трассировки путей часто используют понятие грамматики путей [16]. Этот способ классифицирует пути при помощи строк и регулярных выражений. Каждый символ в строке обозначает определенное событие, произошедшее с лучом (путем) в процессе трассировки.

1. S - (Specular); зеркальное отражение/преломление
2. D - (Diffuse); диффузное отражение
3. G - (Glossy); матовое отражение/преломление
4. V - (Volume); объемное рассеивание
5. L - (Light); источник света
6. E - (Eye); глаз

При этом под символами S и G понимается не только отражение но и преломление света. Грамматика удобна не только для описания различных ситуаций, возникающих в трассировке путей, но она позволяет также классифицировать видимые эффекты, однозначно ставя им в соответствие классы путей которые эти эффекты вызывают (для упрощения изложения атериала объемное рассеивание V мы далее не рассматриваем). Например, EDL - прямое освещение диффузной поверхности. $E(S|G)^+L$ - яркий блик. $EDSL$ - каустик, видимый из камеры напрямую, обусловленный одним зеркальным переотражением света (например солнечный зайчик от зеркала). EDS^+L - каустик, видимый из камеры напрямую, вызванный одни или более зеркальным переотражением. ES^+DS^+L - каустик, видимый через стекло или зеркало, вызванный одним или более зеркальным переотражением.

При помощи грамматики путей удобно классифицировать алгоритмы по типу освещения, которые они способны рассчитывать. Например, Трассировка лучей Уиттеда, рассмотренная нами в самом начале, строит пути ES^*L . Обратная трассировка путей строит пути вида $E(S|D|G)^*L$.

0.5 Прямая трассировка (Light Tracing)

В противоположность обратной трассировки путей, рассмотренной выше, алгоритм прямой трассировки путей начинает свою работу из источника света и строит пути вида $L(S|D|G)^*E$. При каждом соударении с поверхностью создается аналог теневого луча, направленный в камеру. Если луч успешно доходит до камеры, значение яркости записывается в пиксель, соответствующий проекции начала луча на экранную плоскость. Для реализации операции проекции вы можете использовать матрицу $mProj$, которая была рассмотрена нами в разделе "Генерация луча".

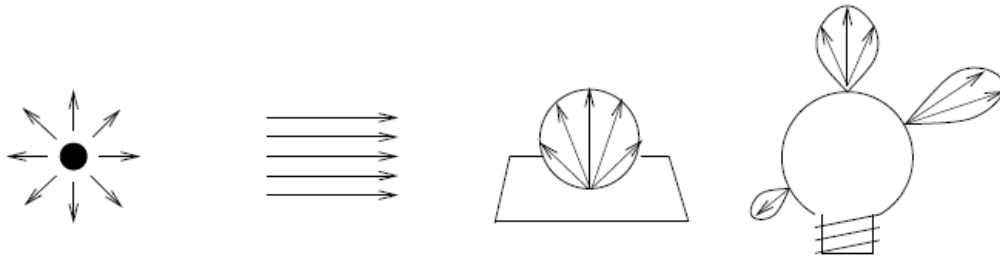


Рис. 42: Иллюстрация различных видов распределения световой энергии для различных источников света.

Пути создаваемые на источнике должны быть сгенерированы в соответствии с распределением световой энергии для источника света (рис. 42). Обратите внимание, что если каждая точка поверхности излучает свет равномерно во все стороны (например площадный источник из рис. 34), то исходя из геометрических соображений и применяя выборку по значимости, необходимо использовать косинусоидальное распределение для расчета направления пути точно так же, как мы вычисляли диффузное отражение (листинги 21, 23). В обратной трассировке путей данный косинус получался стохастически, за счет того что число лучей попавших в источник зависит от направления, с которого они были выпущены. Существующий миф о крайне низкой скорости прямой трассировки путей (или лучей) далеко не всегда соответствует действительности. Такие

эффекты как каустики чрезвычайно быстро могут быть рассчитаны именно при помощи прямой трассировки. В связи с этим, возникает вопрос о том, можно ли комбинировать прямую и обратную трассировку путей для того, чтобы исключить недостатки обоих алгоритмов.

Метод, реализующий эту идею существует и называется "усеченная двунаправленная трассировка путей" [17]. Суть его в следующем:

1. Будем производить обратную трассировку путей с теньевыми лучами. При этом отключим расчет каустиков. Результат сохраняем в изображение с номером 1. На изображении 1 мы получим картинку без каустиков.
2. Будем производить прямую трассировку из источника света, но на этот раз будем сохранять вклад только от тех путей, для которых первое переотражение было зеркальным. Результат сохраняем в изображение с номером 2. На изображении 2 мы получим только каустики.
3. Финальное изображение можно получить простым сложением изображений 1 и 2.

Усеченная двунаправленная трассировка путей дает почти корректный результат, поскольку изображения 1 и 2 будут нести в себе освещение, обусловленное различными типами переотражений. Слово 'почти' означает, что при помощи такого метода не удастся получить каустики, не видимые камерой напрямую. То есть не удастся получить освещение, вызванное путями вида ES^+DS^+L .

0.6 Двунаправленная трассировка путей

Идея объединить преимущества прямой и обратной трассировки привела к созданию алгоритма двунправленной трассировки путей (Bidirectional Path Tracing, BDPT) [15]. Основная идея двунаправленной трассировки путей заключается в том, чтобы соединять теньевыми лучами не только источник с точками пути (как в обратной трассировке) или камеру с точками пути (как в прямой трассировке), но и различные точки путей друг с другом (рис. 43).

Двунаправленная трассировка путей обладает лучшей сходимостью (по сравнению с прямой или обратной трассировкой) на сценах со сложными условиями освещения (преобладающее вторичное освещение) только при реализации многократной выборки по значимости. В противном случае,

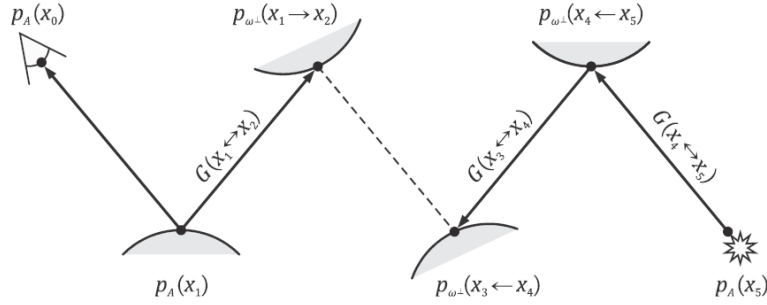


Рис. 43: Идея двунаправленной трассировки путей.

изображение получается сильно зашумленным. Однако даже при реализации многократной выборки по значимости, двунаправленная трассировка путей по прежнему не позволяет эффективно рассчитывать пути вида ES^+DS^+L (каустики, видимые через зеркало или стекло). Такие пути могут быть эффективно вычислены при помощи фотонных карт, переноса света Метрополиса и метода соединения вершин (vertex merging) [18].

Поскольку теньевые лучи в двунаправленной трассировке могут соединять произвольные точки различных путей, реализация многократной выборки по значимости значительно усложняется. Если ранее для вычисления весов достаточно было рассмотреть только 2 стратегии, то теперь число рассматриваемых вариантов значительно увеличивается. Рассмотрим путь на рисунке 43.

0.6.1 Многократная выборка по значимости в BDPT

Рассмотрим некоторый путь $x_0..x_5$. Для такой последовательности точек существует несколько способов соединения их теньевым лучом (обозначено вертикальной чертой):

$$\begin{array}{l|l}
 x_0 & x_1 \ x_2 \ x_3 \ x_4 \ x_5 \\
 x_0 \ x_1 & x_2 \ x_3 \ x_4 \ x_5 \\
 x_0 \ x_1 \ x_2 & x_3 \ x_4 \ x_5 \\
 x_0 \ x_1 \ x_2 \ x_3 & x_4 \ x_5 \\
 x_0 \ x_1 \ x_2 \ x_3 \ x_4 & x_5
 \end{array}$$

При вычислении весов для какого-либо из случаев, многократная выборка по значимости должна учитывать все остальные случаи. Формула 16 применяется для вычисления веса конкретного пути.

$$w_i = \frac{p_i^\beta}{\sum_{j=1}^N p_j^\beta} \quad (16)$$

Например, для случая $x_0 x_1 x_2 | x_3 x_4 x_5$ i равно 2; N равняется 5. p_i - плотность вероятности распределения отраженного луча для точки с номером 2. Таким образом, для указанного случая из 6 точек $x_0..x_5$ необходимо просчитать всевозможные пути (всего 5) и яркость k -ого сэмпла может быть вычислена по формуле 17 ($N = 5$):

$$L_k = \sum_{j=1}^N w_j \frac{L_j}{p_j} \quad (17)$$

0.7 Перенос света Метрополиса

Трассировка путей обладает достаточно низкой скоростью на сценах со сложными условиями освещения. Сценами со сложными условиями освещения будем называть 2 вида сцен:

1. Сцены, на которых преобладает вторичное освещение вызванное узкими и яркими световыми пятнами (рис. 44).
2. Сцены, на которых ES^+DS^+L пути вносят значительный вклад (рис. 45).

Проблема на таких сценах заключается в том, что в большинстве мест на сцене функция падающего освещения имеет один или более резких максимума. Для того чтобы интеграл от такой функции вычислить с высокой точностью методом Монте-Карло с равномерным распределением случайной величины, нужно большое число выборок.

Для улучшения скорости сходимости на таких сценах Э. Вичем и др. был разработан алгоритм переноса света Метрополиса (Metropolis Light Transport) [19]. Идея этого метода аналогична применению выборки по значимости к под-интегральному выражению 18, однако имеет совершенно иную природу.

$$L(\phi_i, \theta_i) R(\phi_i, \theta_i, \phi_r, \theta_r) \cos(n, l_{\phi_i, \theta_i}) \quad (18)$$

Рассмотрим Метод Монте-Карло в общем виде (формула 19).

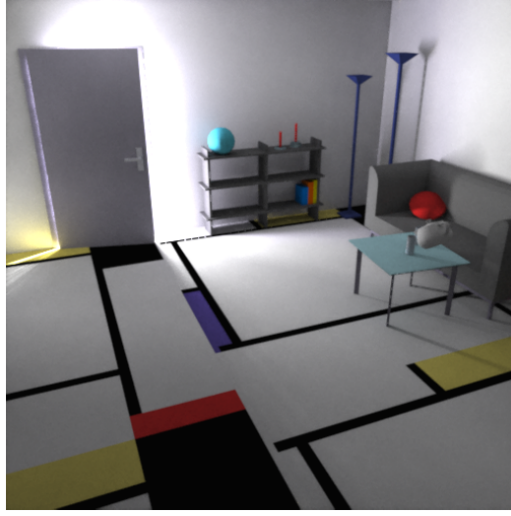


Рис. 44: Пример сцены со сложными условиями освещения.

$$\int_a^b f(x)dx \approx \frac{1}{N} \sum_{i=1}^N \frac{f(u_i)}{p(u_i)} \quad (19)$$

Если бы мы имели функцию распределения $p(x)$ пропорциональную $f(x)$, тогда для некоторой константы c можно записать $p(x) = cf(x)$. Подставив константу в выражение для интеграла 19, получим оценку для константы:

$$c = \frac{1}{\int_a^b f(x)dx} \quad (20)$$

Таким образом, мы имеем возможность оценивать константу c взяв, например, выборки с равномерным распределением и применив метод Монте-Карло. Алгоритм Метрополиса отвечает за оставшуюся часть задачи - создание $p(x)$ пропорционально $f(x)$. Это делается следующим образом:

1. Пусть X - начальный вектор случайных параметров, которые были использованы при вычислении P_i .
2. Сгенерируем путь P_{i+1} путем мутаций случайных параметров X . Т.е. $Y = mutate(X)$.
3. Вычислим некоторым специальным образом вероятность мутации $T(X|Y) = p(X \rightarrow Y)$ и вероятность обратной мутации $T(Y|X) = p(Y \rightarrow X)$.

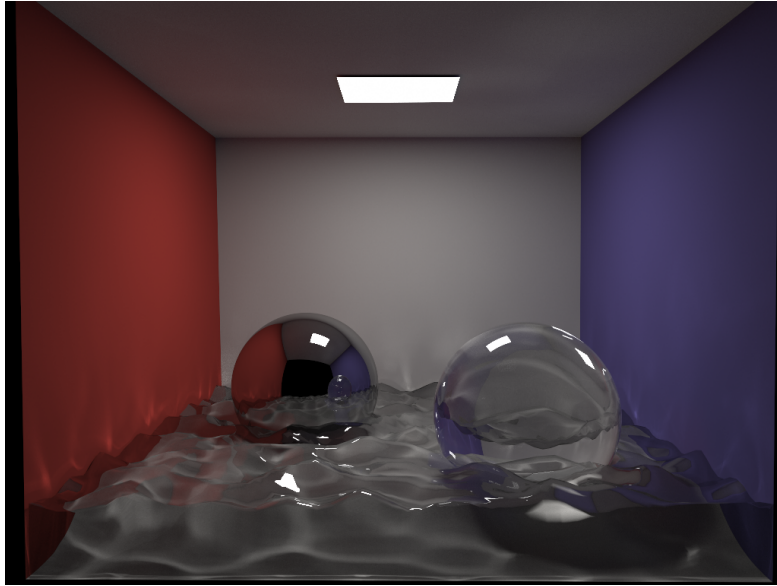


Рис. 45: Пример сцены с ES^+DS^+L путями. Каустики, под водой (рис. 44).

4. Вычислим вероятность принятия мутации $X \rightarrow Y$ по формуле 21.

$$a(Y|X) = \min\{1, \frac{f(y)T(X|Y)}{f(x)T(Y|X)}\} \quad (21)$$

```
void MLT(float imageHist[width][height])
{
    vector<float> X = RandomVector();
    FillWithZerows(imageHist);

    float Fx = TracePath(X);
    int N = width*height*mutationsPerPixel;

    for(int i=0;i<N;i++)
    {
        vector<float> Y = mutate(X);
        float Fy = TracePath(Y);

        float Txy = MutationProbability(X,Y);
        float Tyx = MutationProbability(Y,X);

        float accept = min(1.0, Fy*Txy/Fx*Tyx);
        if (rnd(0,1) < accept)
```

```

{
    X    = Y;
    Fx   = Fy;
}

imageHist[int(X[0]*width)][int(X[1]*height)] += 1.0;
}
return image;
}

```

Listing 27: Упрощенный алгоритм Metropolis Light Transport. Для черно-белого изображения. Алгоритм позволяет сгенерировать пропорциональную $f(x)$ гистограмму. Для получения самого значения $f(x)$ гистограмму нужно масштабировать с учетом средней яркости получаемого изображения.

Обоснование того почему выше-описанный алгоритм работает можно найти в [19]. Наиболее сложная часть в алгоритме переноса света Метрополиса - выбор хорошей стратегии мутации и вычисление вероятности мутаций. Для детального изучения алгоритма мы рекомендуем обратиться к [20] и [19].

0.8 Фотонные карты

Рассмотренные ранее несмещенные методы на основе трассировки путей работают в терминах яркости. Каждый путь в них отвечал за перенос яркости от источника света к камере. Метод фотонных карт работает в терминах потока. Идея этого метода заключается в том, чтобы вычислить распределение световой энергии по сцене при помощи трассировки множества частиц, переносящих порцию световой энергии. После чего можно оценивать интеграл освещенности, выполняя на поверхности поиск ближайших фотонов. Рассмотрим интеграл освещенности (формула 22). Выразим функцию падающей освещенности в терминах потока. Подставим $L(\phi_i, \theta_i)$ в интеграл и получим:

$$I(\phi_r, \theta_r) = \iint_{\phi_i \theta_i} L(\phi_i, \theta_i) R(\phi_i, \theta_i, \phi_r, \theta_r) \cos(n, l_{\phi_i, \theta_i}) d\phi_i d\theta_i \quad (22)$$

$$L(\phi_i, \theta_i) = \frac{d^2\Phi}{\cos(n, l_{\phi_i, \theta_i}) d\phi_i d\theta_i dA} \quad (23)$$

$$(24)$$

$$I(\phi_r, \theta_r) = \iint_{\phi_i \theta_i} \frac{d^2\Phi}{\cos(n, l_{\phi_i, \theta_i}) d\phi_i d\theta_i dA} R(\phi_i, \theta_i, \phi_r, \theta_r) \cos(n, l_{\phi_i, \theta_i}) d\phi_i d\theta_i \quad (25)$$

$$I(\phi_r, \theta_r) = \iint_{\phi_i \theta_i} \frac{d^2\Phi}{dA} R(\phi_i, \theta_i, \phi_r, \theta_r) \quad (26)$$

Дифференциальный поток можно аппроксимировать суммой энергий фотонов в окрестности. Таким образом, в результате преобразований получаем сумму К ближайших фотонов.

$$I(\phi_r, \theta_r) = \iint_{\phi_i \theta_i} \frac{d^2\Phi}{dA} R(\phi_i, \theta_i, \phi_r, \theta_r) \quad (27)$$

$$I(\phi_r, \theta_r) \approx \sum_{i=1}^K R(\phi_i, \theta_i, \phi_r, \theta_r) \frac{\Delta\Phi(\phi_i, \theta_i)}{\Delta A} = \frac{1}{\pi r^2} \sum_{i=1}^K R(\phi_i, \theta_i, \phi_r, \theta_r) \Delta\Phi(\phi_i, \theta_i) \quad (28)$$

В выражении 27 переменная r отвечает за радиус диска на поверхности, в пределах которого выполняется поиск ближайших фотонов. Метод фотонных карт, таким образом, состоит из 4 шагов:

1. Испускание фотонов из источников света
2. Трассировка фотонов
3. Построение фотонной карты
4. Сбор освещенности

Испускание фотонов. Фотоны в данном методе – это частицы, переносящие некоторую небольшую порцию световой энергии. На начальном этапе фотоны испускаются из источника света в соответствие с распределением световой энергии для данного источника. Например известно, что точечный или сферический источник света (такой как солнце) испускают свет равномерно во всех направлениях. Площадные источники света имеют косинусоидальное распределение, имеющее максимум по направлению, совпадающему с нормалью к плоскости источника. Стадия испускания фотонов не отличается от стадии испускания путей вида

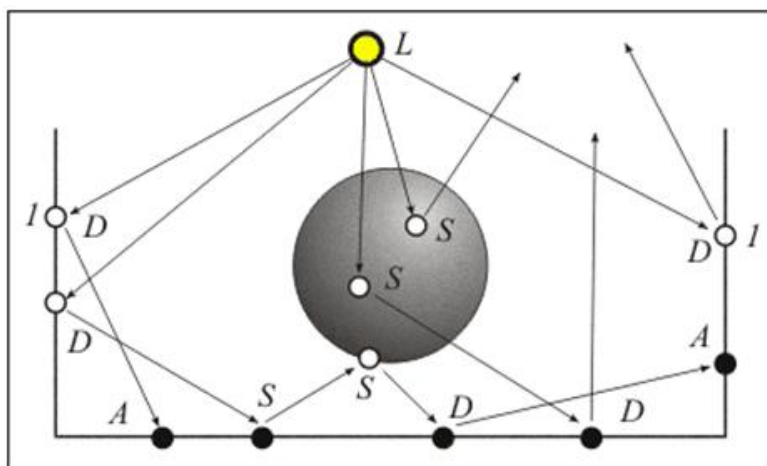


Рис. 46: Процесс трассировки и сохранения фотонов на поверхностях.

$L(S|D|G)*E$ в прямой трассировки путей. Предоставленные ранее функции (листинги 21,23) для генерации лучей в соответствии с косинусоидальным и степенным косинусоидальным распределением могут быть использованы для данной цели без каких-либо модификаций. Что касается порции энергии, которую переносит один фотон, обычно считается что все фотоны испущенны из источника света имеют единичную энергию. Позже, в процессе сбора освещенности результирующая энергия делится на суммарное число выпущенных из источника света фотонов и умножается на мощность (в ваттах) источника света. Таким образом, каждый фотон переносит порцию энергии равную $\frac{P_{light}}{N_{photons}}$.

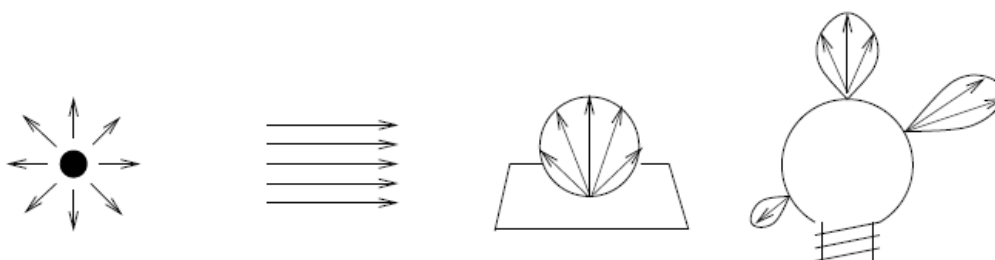


Рис. 47: Иллюстрация различных видов распределения световой энергии для различных источников света.

Трассировка фотонов. Русская рулетка. Трассировка фотонов происходит похожим на прямую трассировку путей образом за исключением одного важного отличия - механизма русской рулетки (этот механизм можно также применять и в трассировке путей [17], но исторически он появился именно в методе фотонны карт) [21]. Легче всего продемонстрировать механизм русской рулетки на примере. Допустим мы имеем ламбертовский материал с коэффициентом отражения 0.25 и фотон падающий на эту поверхность с энергией равной 1. Очевидным решением является модифицировать энергию фотона, умножив ее на 0.25 и продолжить трассировку фотона. Механизм русской рулетки поступает по другому. Вместо того чтобы модифицировать энергию фотона, его энергия сохраняется, но с вероятностью 0.25 фотон отражается и продолжает трассировку, а с вероятностью 0.75 - погибает (рис. 48). Таким образом, если на поверхность падает 1000 фотонов, 250 отразится, сохранив энергию, а остальные 750 - погибнут. За счет такого механизма повышается точность решения, поскольку в местах с высокой освещенностью будет сохраняться больше фотонов, чем в местах с низкой освещенностью; в неосвещенных областях фотоны будут отсутствовать. Нетрудно



Рис. 48: Русская рулетка.

заметить, что при использовании русской рулетки в чистом виде все фотоны будут иметь единичную энергию. То есть, энергию фотона хранить не обязательно, а для того чтобы оценить значение освещенности, достаточно либо посчитать число фотонов в заданном радиусе сбора, либо найти радиус, в котором лежит K ближайших фотонов. Мы вернемся к

этому вопросу позже, когда будем рассматривать процесс сбора освещенности. При этом для того чтобы выполнять правильно сбор освещенности, нам потребуется хранить позицию фотона и нормаль к поверхности в той точке удара о поверхность.

Сохранение фотонов в фотонной карте. Вопрос сохранения фотонов на поверхностях (в фотонной карте) заслуживает отдельного рассмотрения. Фотоны сохраняются на всех диффузных (ламбертовых) поверхностях, которые они ударяют. Как правило фотоны используются только при вычислении части интеграла, обусловленной диффузной компонентой. Зрекальная компонента не может быть вычислена при помощи фотонов по той же причине, по которой прямая трассировка путей (Ligh Tracing) отображает зеркала и стекла черными: вероятность встретить фотон (или путь от источника), отразившийся в строго заданном направлении стремится к нулю. Из этого следует, что компонента, обусловленная матовым отражением (glossy reflection) все же может быть вычислена при помощи фотонных карт, однако эффективность такого расчета будет тем ниже, чем более зеркальный характер приобретает ДФО. Следует подчеркнуть, что фотон в течении трассировки может быть сохранен несколько раз - на всех диффузных поверхностях, которые он ударяет.

Итак, Фотоны сохраняются на всех диффузных поверхностях, которые они ударяют. Однако, одним из часто-применяемых трюков является в буквальном смысле 'пропуск' первого отскока для процедуры сохранения фотона. Фотоны в этом случае сохраняются только после первого отскока а освещение, получаемое в результате интегрирования фотонной карты - целиком вторичное. Этот метод значительно снижает число фотонов, необходимое для расчета вторичного освещения. Первичное освещение гораздо быстрее вычисляется при помощи трассировки лучей/путей, поэтому данный прием имеет важное значение для ускорения процесса рендеринга.

Представление фотона. Для каждого фотона необходимо сохранять его позицию, нормаль поверхности, в которую ударился фотон и переносимую энергию. В самом простом случае фотон не хранит энергию, которую он переносит (все фотоны переносят единичную энергию). Однако, для того чтобы учитывать цвет, понаобится сохранять длину волны в виде индекса выборки из спектра или RGB коэффициентов.

```
struct Photon // 16 bytes per photon
{
```

```

float3 pos;
uint compressedNorm : 30;
uint colorId : 2; // 4 colors max
}

```

Listing 28: Простейшее представление фотона. Нормаль сжата в одно 30-битное целое число в целях экономии памяти. Существующих способов сжатия нормали достаточно много. Для фотонных карт точности в 30 бит более чем достаточно.

Авторы [21] используют представление из листинга 29 для хранения фотона. В переменных (phi, theta) вместо нормали к поверхности авторы [21] хранят направление, с которого фотон пришел. Это необходимо для вычисления матовых отражений (glossy reflection). Однако, если фотонная карта используется для вычисления диффузной составляющей, вместо этого направления удобно хранить нормаль к поверхности в точке удара (в разделе про сбор освещенности мы покажем для чего).

```

struct Photon // 20 bytes per photon
{
    float3 pos;
    char    color[4];
    char    phi, theta;
    short    flag;
}

```

Listing 29: Структура фотона из [21].

Иногда удобно смешивать стратегию русской рулетки и стратегию с модификацией энергии. По этой причине удобно хранить цвет в виде float3. Более того, поскольку существует метод стохастических прогрессивных фотонных карт (позволяющий обработать произвольное число фотонов при фиксированном объеме памяти), компактное представление фотона в памяти теряет свою актуальность. С другой стороны, различные виды сжатия (для цвета фотона, нормали к поверхности) замедляют процесс сбора. Для реализации фотонных карт на центральном процессоре мы рекомендуем хранить фотоны в простом представлении.

```

struct Photon // 40 bytes per photon
{
    float3 pos;
    float3 norm;
    float3 color;
    float    lambda;
}

```

Listing 30: Структура фотона для CPU.

Для графических процессоров мы рекомендуем использовать паттерн хранения 'Structure Of Arrays' вместо 'Array Of Structure'. Поскольку GPU поддерживают 16-битную плавающую точку на аппаратном уровне, в целях снижения нагрузки на шину памяти во время сбора освещенности мы рекомендуем использовать тип `half` (16-ти битный `float`).

```
struct PhotonArray // 32 bytes per photon
{
    float4 pos[N];    // pos.w is free
    half4  norm[N];   // norm.w is free
    half4  color[N];  // color.w is free
}
```

Listing 31: Структура хранилища фотонов для GPU.

Построение фотонной карты. В самом простом случае можно хранить фотонную карту в виде обыкновенного массива. Однако это не позволит в последствие выполнять эффективный поиск ближайших фотонов. Поэтому под построением фотонной карты мы будем понимать построение ускоряющей структуры, позволяющей выполнять эффективный поиск ближайших фотонов. Существует довольно большое число способов, как делать это эффективно. Способы могут различаться для поиска k -ближайших или всех фотонов в заданном радиусе. Среди наиболее эффективны структур следует отметить пространственные хэш-таблицы и деревья, построенные при помощи эвристики VVH [22] аналогично эвристике SAH, рассмотренной нами ранее.

Сбор освещенности. После того как фотонная карта построена, наступает стадия сбора освещенности. Из виртуальной камеры испускаются лучи и выполняется обратная или стохастическая трассировка лучей (или трассировка путей). В местах соударений луча и поверхности производится сбор освещенности. Причем, для стохастической трассировки лучей (или путей), диффузные переотражения не учитываются. Компонента, обусловленная диффузными переотражениями получается при интегрировании фотонной карты (формула 29). Диффузные переотражения в процессе обратной трассировки лучей/путей вычисляются при использовании метода Финального Сбора, который мы рассмотрим позже.

$$I(\phi_r, \theta_r) \approx \frac{1}{\pi r^2} \sum_{i=1}^K R(\phi_i, \theta_i, \phi_r, \theta_r) \Delta\Phi(\phi_i, \theta_i) \quad (29)$$

Существует 2 основные стратегии сбора освещенности:

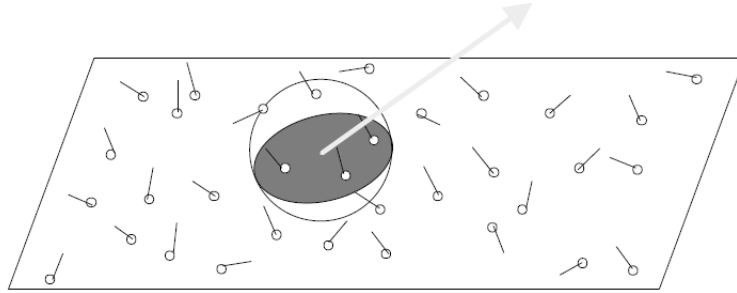


Рис. 49: Иллюстрация к формуле 29. Сбор освещенности или интегрирование фотонной карты. В данной формуле r - радиус диска, полученного при проекции сферы на поверхность. πr^2 - площадь диска.

1. Сбор k -ближайших фотонов (динамический радиус сбора), в соответствии с формулой 29. Рисунок 50.
2. Сбор всех фотонов в заданном радиусе (фиксированный радиус сбора). Данная стратегия упрощает процесс сбора и дает корректный результат в пределе (при стремлении общего числа фотонов к бесконечности), однако не обеспечивает постоянное значение K в формуле 29.

Поиск ближайших фотонов, как правило, использует специальную структуру 'max-heap' для упорядочивания списка фотонов [21]. Однако, поиск k -ближайших включает в своей основе более простой поиск всех фотонов в заданном радиусе, поскольку даже при поиске k -ближайших необходимо начинать с некоторого радиуса и уметь находить все фотоны в нем.

0.8.1 Финальный сбор

Один из артефактов, возникающих при реализации сбора освещенности - темные края поверхностей (рис 52, 53). Данный артефакт возникает по причине того, что на границах поверхностей освещение собирается только с половины диска. При этом площадь, вычисляемая как πr^2 для целого диска остается прежней. Один из способов, позволяющий убрать темные края заключается в том, чтобы найти реальную площадь элемента поверхности на котором происходил сбор при помощи вычисления площади пересечения сферы сбора и всех треугольников с нормалью, близкой

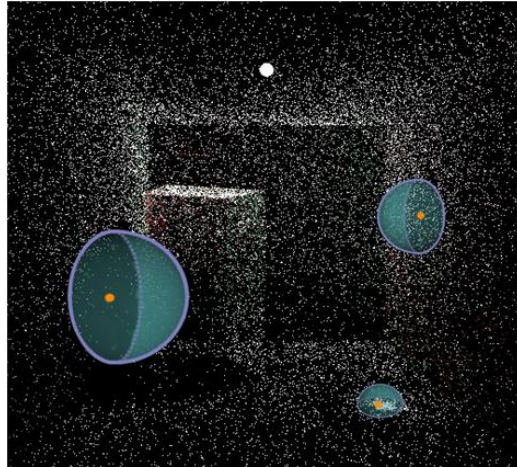


Рис. 50: Сбор k -ближайших фотонов. Радиус сбора выбирается динамически (чтобы ограничить ровно k фотонов) в зависимости от плотности фотонной карты.

к нормали в точке сбора (то есть вычислить честную проекцию сферы на поверхность). Второй метод называется Финальный Сбор (Final Gathering, FG). Вместо непосредственного сбора освещенности с фотонов, в методе финального сбора из заданной точки испускается некоторое число лучей по полусфере и освещенность собирается уже в тех местах, куда попали лучи (рисунок 51).

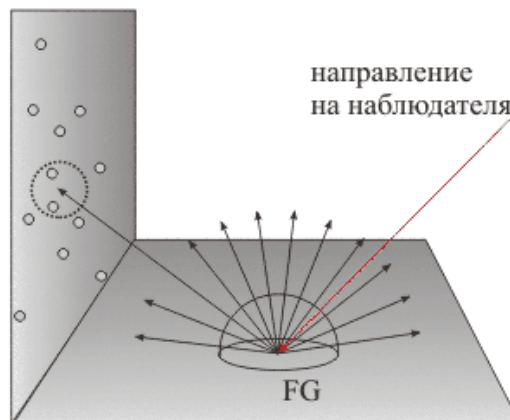


Рис. 51: Иллюстрация метода финального сбора.

При использовании финального сбора сами фотонные карты, таким образом, задействуются только для вычисления аппроксимации вторичной освещенности. Помимо избавления от темных краев, финальный сбор

значительно повышает точность решения, позволяя обрабатывать значительно число фотонов. Хотя скорость финального сбора в среднем сравнима со скоростью обычной трассировки путей, финальный сбор, как правило, дает меньше шума (особенно для сцен со сложными условиями освещения) чем обычная трассировка путей. Метод финального сбора наиболее часто применяется совместно с кэшем освещенности (будет рассмотрен рассморим далее), поскольку для кэша освещенности важна даже не столько точность решения, сколько отсутствие шума в нем.

Карты светимости (radiosity maps). Процесс поиска ближайших фотонов является, как правило, узким местом финального сбора. Поскольку в данном случае при помощи самих фотонных карт достаточно получить лишь грубую оценку освещенности, существуют различные методы ускорения финального сбора, использующие этот факт. К таким методам относятся карты светимости [23] или октанные текстуры [24]. Их смысл заключается в том, чтобы предрасчитать светимость в точках поверхности [23] (или в объеме [24]) и запомнить ее в виде карты светимости - двумерной или трехмерной (возможно разряженной) текстуры. При попадании луча финального сбора в определенную точку поверхности поиск ближайших фотонов не выполняется. Вместо этого производится быстрая выборка значения из карты светимости.

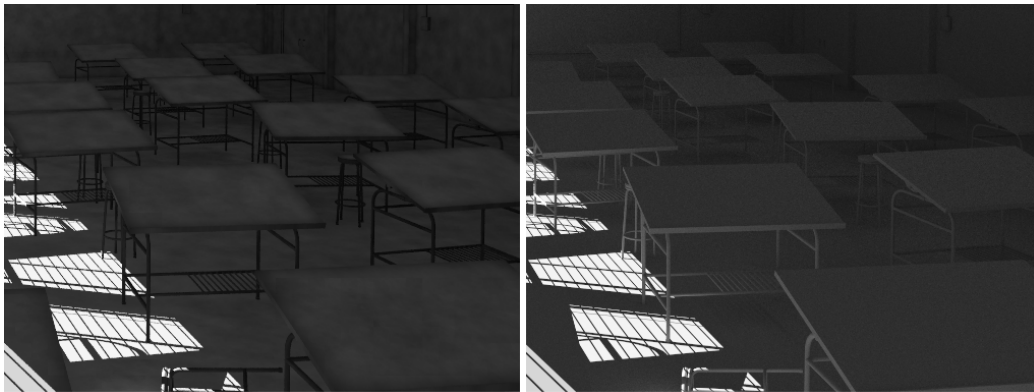


Рис. 52: Сбор освещенности с 10М фотонов (слева). Финальный сбор с 1М фотонов (справа).



Рис. 53: Сбор освещенности с 10М фотонов (слева). Финальный сбор с 1М фотонов (справа).

0.8.2 Прогрессивные фотонные карты

Для фотонных карт в классической реализации серьезной проблемой является объем потребляемой памяти. Для того, чтобы получить изображение высокого качества может потребоваться до нескольких миллиардов фотонов. Хранить все фотоны в памяти в этом случае не представляется разумным, поскольку даже если хранить фотоны в соответствии с компактным представлением в 16 байт, один миллиард фотонов займет $16 * 10^9 \approx 15Gb$.

Для того чтобы иметь возможность обрабатывать произвольное число фотонов имея фиксированный объем памяти был разработан метод прогрессивных фотонных карт [25]. Основная идея метода прогрессивных фотонных карт заключается в том, чтобы повторять все 4 шага (испускание фотонов из источника света, трассировка, построение фотонной карты и сбор освещенности) несколько раз для порций фотонов фиксированного размера (например, по 1 миллиону фотонов). При этом, при обработке каждой следующей порции фотонов радиус сбора и аккумулируемый поток уменьшаются в соответствии с формулой 30. Такой метод называется Прогрессивными фотонными картами (Progressive Photon Mapping, PPM [25]). Если выбрана стратегия с фиксированным радиусом сбора, то для каждой следующей порции фотонов радиус сбора уменьшается в соответствии с соотношением 31. Такой алгоритм называется Стохастическими Прогрессивными Фотонными Картами (Stochastic Progressive Photon Mapping, SPPM [26], [27]). Параметр α варьируется в пределах от 0 до 1. Рекомендуемое значение - больше или равно $2/3$.

$$\frac{r_{i+1}^2}{r_i^2} = \frac{N_i + \alpha M_i}{N_i + M_i} \quad (30)$$

$$\frac{r_{i+1}^2}{r_i^2} = \frac{i + \alpha}{i + 1} \quad (31)$$

В соотношении 30 N_i - число уже собранных фотонов. M_i - число вновь добавленных фотонов (на очередном проходе).

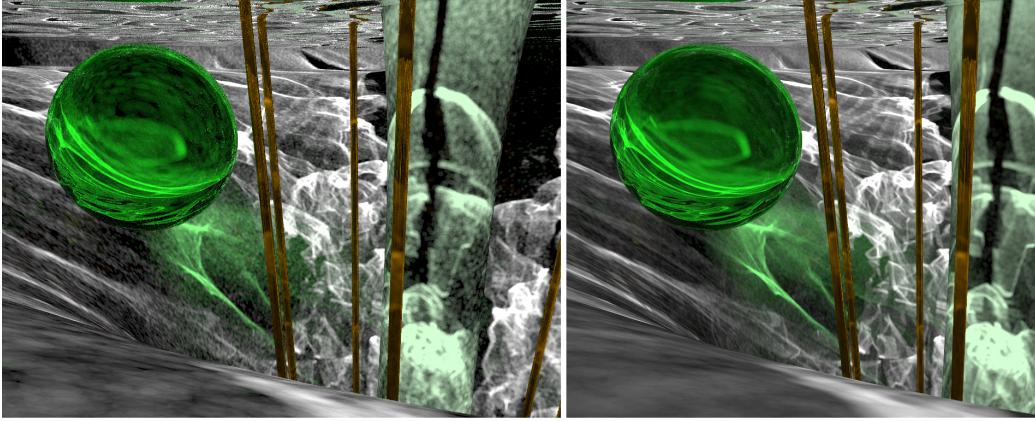


Рис. 54: Стохастические прогрессивные фотонные карты. 1М фотонов слева. 10М (10 проходов) - справа.

Метод стохастических прогрессивных карт значительно проще в реализации, чем обыкновенные прогрессивные фотонные карты, поскольку он не фиксирует точки сбора (что необходимо делать в методе прогрессивных фотонных карт) [27]. При этом, результаты для различных порций фотонов могут быть скомбинированы простым усреднением, как и в обыкновенной трассировке путей (с.м. раздел 'прогрессивное вычисление интеграла'). Алгоритм стохастических прогрессивных фотонных карт выглядит следующим образом:

1. Заранее выделяем некоторый объем памяти для хранения достаточно большой порции фотонов (например 1 миллион фотонов).
2. Трассируем фотоны из источника света до тех пор пока в фотонной карте не накопится указанное число фотонов. Важно отметить при этом, что при расчете доли световой энергии, которую переносит один фотон, необходимо в качестве суммарного числа фотонов

использовать количество выпущенных из источника света (а не количество аккумулированных) фотонов.

3. Строим ускоряющие структуры для быстрого поиска ближайших фотонов.
4. Трассируем пути из виртуальной камеры. В точках пересечения с диффузными поверхностями собираем освещенность по формуле 29. При этом используем фиксированный радиус сбора r_i . Цвет, как и при обыкновенной трассировке путей накапливается в пикселях.
5. Уменьшаем радиус в соответствии с формулой 31. Переходим к шагу 2.

0.8.3 Некоторые важные оптимизации

Три фотонные карты В целях эффективности (повышения точности) в [21] предлагается хранить 3 отдельные фотонные карты:

1. Глобальная фотонная карта. Данная карта содержит фотоны, прошедшие путями вида $L(S|D|G|V)^+D$ и используется для вычисления диффузного освещения.
2. Каустическая фотонная карта. Данная карта содержит фотоны, прошедшие путями вида LS^+D и используется для вычисления каустиков.
3. Объемная фотонная карта. Данная карта содержит фотоны, прошедшие путями вида $L(S|D|G|V)^+V$ и используется для вычисления объемных эффектов.

Разделение фотонных карт позволяет обрабатывать каустики и диффузное освещение (а также объемное рассеивание) при помощи различных порций фотонов. Это особенно важно при использовании финального сбора, поскольку в этом случае для диффузного освещения с финальным сбором нужно гораздо меньше фотонов чем для каустиков.

Определение видимости. Рассмотрим изображение сцены классной комнаты на рис. 55. Для расчета освещения на этой сцене фотоны выпускались снаружи. При этом, лишь небольшая часть фотонов проникла внутрь комнаты через окно. Остальные фотоны сохраняются с обратной стороны стен и потолка, приводя к неэффективному расходованию вычислительных ресурсов. Для того чтобы не сохранять фотоны на снаружи

комнаты, необходимо перед тем, как производить трассировку фотонов, пометить некоторым образом поверхности, для которых фотоны сохраняются в фотонной карте.

Простейший способ заключается в трассировке некоторого небольшого числа путей для каждого пиксела из виртуальной камеры и прямой пометки пересекаемых треугольников. При этом, каждую сторону треугольника необходимо пометать отдельно. Чтобы идентифицировать сторону треугольника, можно использовать векторное произведение для вычисления нормали. Поскольку вершины треугольника хранятся в памяти в заданном порядке, такой способ однозначно позволяет идентифицировать сторону по нормали. Недостаток данного метода заключается в том, что некоторые тонкие примитивы (треугольники) могут быть пропущены алгоритмом. Хотя, если используется финальный сбор, это не имеет значения поскольку такие примитивы вносят почти нулевой в результирующее значение интеграла. Т.е. чем тоньше треугольник, тем больше вероятность не попасть в него в процессе трассировки путей из виртуальной камеры. Но также уменьшается вероятность попасть в данный треугольник в процессе сбора освещенности.

Еще один способ заключается в трассировке так называемых частиц видимости [28]. Частицы видимости представляют из себя фотоны, испускаемые из камеры. Таким образом, сначала строится фотонная карта видимости, в которой все фотоны трассируются из виртуальной камеры, а затем, используя карту видимости, строится фотонная карта освещенности.

Карты проекций. При оптимизированной реализации построения ускоряющей структуры и сбора освещенности, одним из наиболее ресурсоемких этапов в фотонных картах становится трассировка фотонов. Причина низкой скорости трассировки заключается не том, что медленно происходит сам процесс трассировки одиночного фотона, а в том, на сложных сценах значительная часть испускаемых из источника света фотонов погибает в процессе трассировки поскольку фотоны не достигают видимых областей. Это особенно верно при реализации определения видимости, рассмотренном выше и при трассировке каустических фотонов. На сцене класса (рис. 55) только $\frac{1}{7}$ часть фотонов выпущенных из источника света попала внутрь комнаты. На классической сцене 'cornell box' (рис. 29) только $\frac{1}{10}$ часть каустических фотонов, выпущенных из источника света, попала в зеркальный или стеклянный шары, образуя каустики. Русская рулетка дополнительно усугубляет ситуацию, стохастически убивая фотоны.



Рис. 55: Комната, освещенная солнцем через окно. Лишь $\frac{1}{7}$ часть выпущенных из источника света фотонов попадает внутрь комнаты.

Карты проекций [21] - это метод, позволяющий не выпускать фотоны в те области, где они гарантированно погибают, не сохраняясь ни на одной поверхности. Идея этого метода заключается в том, чтобы спроецировать сцену на источник света (например при помощи растеризации в кубическую текстурную карту) и пометить на этой проекции 'мертвые области'. После чего в эти мертвые области фотоны не трассируются совсем, погибая еще при рождении на источнике света.

0.8.4 Объемные фотонные карты

При помощи алгоритма фотонных карт можно достаточно эффективно вычислять эффекты основанные на объемном рассеянии света. В этом случае фотоны сохраняются прямо в объеме, а сбор освещенности происходит при помощи 'марширования по лучу' (ray marching).

Алгоритм марширования по лучу:

```
float3 RayMarching(Ray ray)
{
    float tmin, tmax;
    if(!RayBoxIntersection(ray, boxMin, boxMax, &tmin, &tmax))
        return Black;

    float dt = 0.05f;
```

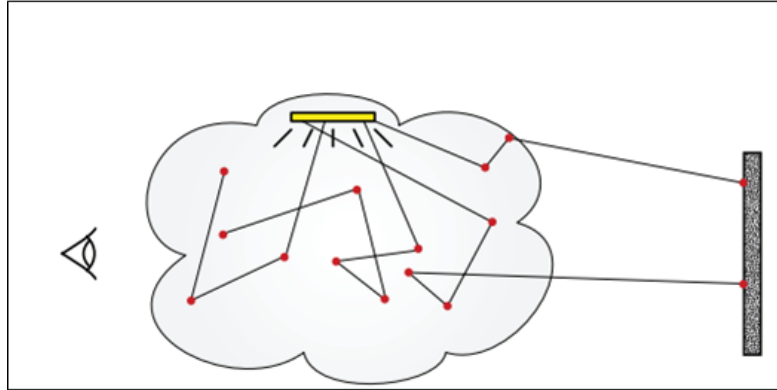


Рис. 56: Трассировка фотонов внутри объема [27].

```
float t = tmin;

float alpha = 1.0f;
float3 color = float3(0,0,0);

while(t < tmax && alpha > epsilon)
{
    float3 p = ray.pos + ray.dir*t;
    float a = GetOpacity(p);
    color += a*alpha*GatherPhotons(p);
    alpha *= (1.0f-a);
    t += dt;
}

return color;
}
```

Listing 32: Марширование луча внутри объема.

$$\frac{r_{i+1}^3}{r_i^3} = \frac{i + \alpha}{i + 1} \quad (32)$$

Для стохастических прогрессивных объемных фотонных карт соотношение 31 модифицируется в 32 с учетом того что процесс происходит в объеме.

0.8.5 Резюме по фотонным картам

Фотонные карты - метод дающий смещенную оценку (смещенный метод). Прогрессивные фотонные карты и стохастические прогрессивные

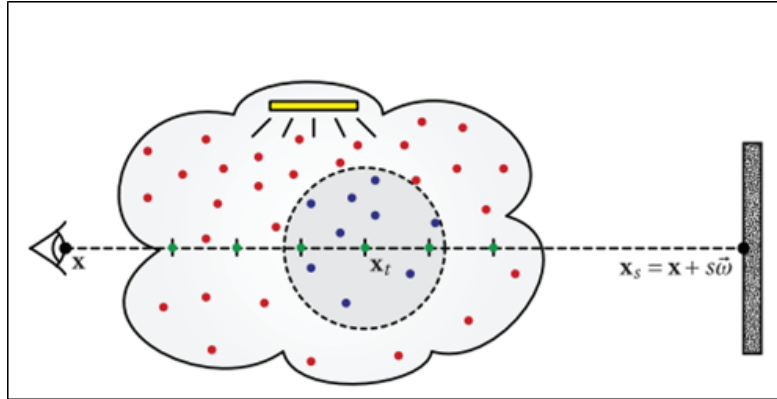


Рис. 57: Сбор освещенности при помощи марширования вдоль луча [27].

фотонные карты - также методы смещенные, но при этом консистентные (сходимость по вероятности) [25], [25]). На практике смещение проявляется на изображении в виде цветных пятен (как и в кэше освещенности). В фотонных картах цветные пятна достаточно долго не удаляются с изображения в процессе расчета и изображение лишенное пятен может потребовать обработки нескольких миллиардов фотонов. При таком большом числе фотонов гораздо более простая в реализации прямая и/или обратная трассировка путей даст более точное и приемлимое для человеческого глаза решение.

Финальный сбор позволяет снизить число необходимых фотонов примерно на порядок. Однако, он может быть использован только для вычисления диффузного вторичного освещения и его корость сравнима со скоростью обыкновенной трассировки путей. Для эффективной реализации финального сбора рекомендуется использовать карты светимости [23] или октанные текстуры [24]. Последний метод предпочтительнее, поскольку для него не нужно реализовывать отображение поверхностей трехмерных объектов на двумерную текстуру (что в общем случае является нетривиальной задачей).

Тем не менее, фотонные карты наряду с методом соединения вершин [18] являются одним из самых эффективных алгоритмов для вычисления путей вида S^+DS^+ (каустики, видимые через стекло или зеркало). Такие пути, как правило, значительно хуже вычисляются трассировкой путей.

0.9 Кэш освещенности (Irradiance Cache)

Кэш Освещенности (Irradiance Cache, IC) – это не способ вычисления интеграла освещенности. Это лишь способ ускорения вычисления этого интеграла на множестве точек. Основная идея заключается в том, что вторичное освещение разделяется на две компоненты – низкочастотную (диффузную) и высокочастотную (отражающую). Низкочастотная компонента на изображении меняется плавно, поэтому ее можно вычислить каким-либо из методов лишь в очень небольшом числе точек, а в остальных – интерполировать [29]. Рисунки 1 и 2 демонстрируют пример использования кэша освещенности.

Высокочастотная компонента обычно обусловлена резкими максимумами BRDF, поэтому она может быть вычислена сэмплированием с помощью относительно небольшого числа (что конечно не всегда так) лучей только в максимумах BRDF (importance sampling).

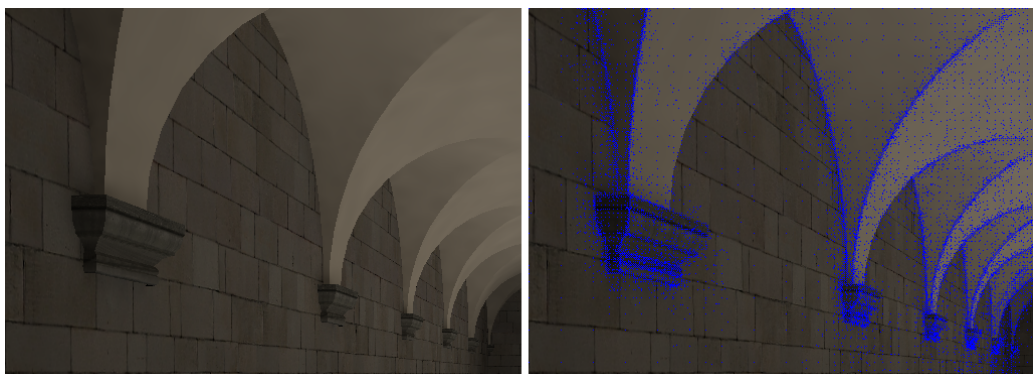


Рис. 58: Освещение вычисленное при помощи кэша освещенности (слева). Визуализированные точки кэша (справа).

Различают кэш излучения в трехмерном пространстве сцены (world space) и кэш в экранной плоскости (screen space). Реализация кэша в экранной плоскости проще и эффективнее. Интерполяция также производится в пространстве экрана. Однако, кэш излучения в пространстве экрана может быть использован по понятным причинам только для первично видимых поверхностей.

Как правило, кэш освещенности вычисляется на лету. Каждый раз при вычислении вторичного диффузного освещения делается попытка выборки из кэша. Если значение может быть выбрано так, что ошибка меньше допустимой величины, оно получается как результат интерполяции ближайших точек кэша. Если ошибка больше допустимой величины,

значение честно вычисляется с помощью трассировки лучей по всем направлениям (или финального сбора из фотонной карты) и полученная точка сохраняется в КЭШе [29] (листинг 33).

```
float3 IrradianceCaching(float3 p, float3 n)
{
    if (InterpolationIsPossible(p,n))
    {
        return IrradianceCacheLookUp(p,n);
    }
    else
    {
        float3 I = EvalIntegral(p,n);
        float R = CalcValidityRadius(p,n);
        InsertRecordInCache(I,R,p,n);
        return I;
    }
}
```

Listing 33: Применение кэша освещенности.

Алгоритм кэширования вторичного освещения содержит 2 узких места - расчет собственно вторичного освещения в точках кэша и финальный алгоритм интерполяции освещения во всех остальных точках.

0.9.1 Вычисление вторичного освещения

Стандартным способом вычисления вторичного освещения в точках кэша является монте-карло трассировка лучей или финальный сбор в сочетании с фотонными картами. Однако в [29] предлагается вычислять вторичную освещенность в точках с помощью растеризации в кубическую текстурную карту. Данная операция может производиться очень быстро т.к. имеет аппаратную поддержку даже в довольно старых видеокартах (обычно используется для симуляции отражений на объектах).

Несмотря на преимущество по скорости, недостаток этого алгоритма в том, что он не может учитывать зеркальных отскоков света и может быть в ряде случаев медленнее трассировки лучей на больших сценах. Причина этого заключается в том, что при растеризации вся геометрия обрабатывается графическим процессором. Она проходит весь графический конвейер: преобразование вершин, возможно тесселяцию и геометрические преобразования над примитивами, интерполяция атрибутов вершин и вычисление значений освещенности в пикселах. Допустим, геометрия – это некоторое здание, разделенное на комнаты. Всё здание содержит 2 миллиона вершин. Пусть виртуальный наблюдатель находится в одной из комнат. Несмотря на то, что комната может составлять 10-20% всей

геометрии, при вычислении освещения в точке каждый раз потребуется пропускать через графический конвейер все 2 миллиона вершин. Тот факт, что 90% сцены может быть закрыто стеной или каким-то близко-расположенным объектом никак не используется. В то же время, скорость трассировки лучей при правильной реализации зависит от размера входных данных сублинейно (обычно логарифмически), а не линейно, как в случае растеризации.

Отдельного внимания заслуживает вопрос точности вычисления освещенности в точках кэша. Допустим для вычисления освещенности при помощи трассировки путей в пикселях на некоторой сцене будет достаточно около 1000 путей на пиксел. Тогда для вычисления освещенности в точках кэша потребуется может потребоваться большая точность (порядка 4000 путей). Недостаточная точность вычисления для попиксельной трассировки путей приводит к шуму на изображении, который в некоторой степени фильтруется глазом и мало заметен. В случае кэша освещенности, недостаточная точность приводит к появлению цветных пятен, которые гораздо более заметны для человеческого глаза, чем шум.

0.9.2 Вычисление радиуса валидности

Одним весьма неочевидным моментом при реализации кэша освещенности является вычисление радиуса валидности точки кэша (переменная R , листинг 33). Подразумевается, что каждая точка иррадианс кэша может быть использована для интерполяции только в некоторой определенной области. Есть по крайней мере 2 причины, по которым радиус валидности точки нужно ограничивать. Первая причина - стремление избежать артефактов. Вторая - скорость выборки из кэша. Если слишком много точек будут затрагиваться при каждой выборке, интерполяция станет медленной.

Обычно область валидности ограничивают сферой с центром в точке иррадианс кэша и некоторым радиусом, называемым радиусом валидности (хотя есть работы где для этой цели используются эллипсоиды). В [3] для вычисления радиуса валидности было использовано 5 различных эвристик. Каждая из них имеет недостатки, но все вместе они дают удовлетворительный результат.

Из самых простых - используется эвристика расстояния до ближайших поверхностей. Эта эвристика вычисляется исключительно из геометрических соображений. При оценке освещенности, для всех трассируемых лучей запоминается расстояние, на котором они ударились о поверхность и из этих расстояний берется наименьшее. Однако при использовании этого подхода, в процессе нахождения минимума важно не принимать в

расчет лучи, имеющие маленький угол с тангент-плоскостью (рис. 59).

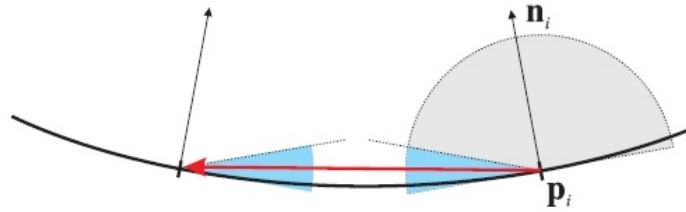


Рис. 59: Точки расположенные на вогнутой поверхности. Если принимать в расчет лучи, идущие перпендикулярно нормали, радиус валидности получится слишком маленьким.

В противном случае на вогнутых криволинейных поверхностях радиус валидности точки всегда будет равным нулю. Альтернативный метод реализации эвристики расстояния до поверхностей заключается в том, чтобы не искать минимальное расстояние а вычислить так называемое "среднее гармоническое расстояние" (формула 33).

$$R_i^{HMD} = \frac{N}{\sum_{i=1}^N \frac{1}{r_i}} \quad (33)$$

Число N в формуле 33 - это количество лучей используемых при сэмпловании полусферы. r_i - расстояние от точки кэша до поверхности в которую ударился соответствующий луч. Формула 33 для среднего гармонического расстояния характерна тем, что чем меньше расстояние r_i тем больше оно влияет на финальный результат. Дополнительно размер радиуса валидности рекомендуется ограничивать снизу размером 2-4 пикселей (спроецированных на сцену в world space) и сверху размером 20-64 пикселей. Конкретные числа могут варьироваться.

0.9.3 Структуры данных и интерполяция

Для хранения точек иррадианс кэша обычно используется специальное октодерево со множественными ссылками (multiple reference octree) [29]. Точки кэша излучения в таком октодереве хранятся только в листьях, но каждый лист хранит список всех сфер, которые он пересекает. Основной плюс такого октодерева в том, что возможно осуществить простой рекурсивный поиск от корня к листу с последующим перебором всего лишь нескольких точек кэша освещенности (листинг 34).

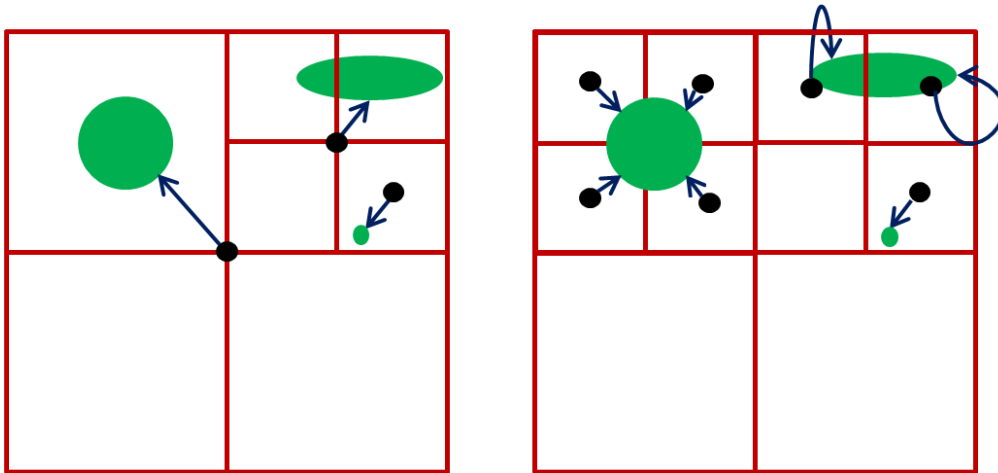


Рис. 60: Октодерево с одиночными ссылками (слева) и окто-дерево с множественными ссылками (справа). Стрелки обозначают ссылки.

Интерполяция. В заданной точке трехмерного пространства, при помощи поиска в октодереве мы должны найти все сферы (с центрами в точках иррадианс кэша и радиусами равными радиусам валидности записей кэша), которые пересекают данную точку. Затем, пройдя в цикле по всем точкам, нужно собрать с них освещение. Одна из практических формул была предложена Tabellion-ом и Lamorlette в [23] и представлена ниже в виде алгоритма. Переменные lookPos и lookNorm - это позиция и нормаль в той точке, в которой мы хотим сделать выборку из кэша освещенности. pointPos и pointNorm соответственно - позиция и нормаль для i-ой записи иррадианс кэша, validityRadius - радиус валидности этой записи.

```
float4 IrradianceCacheLookUp(float3 lookPos, float3 lookNorm)
{
    float3 nodePos = g_octreeCenter;
    float nodeSize = g_octreeBoxSize;
    OctreeNode node = GetOctreeNode(0); // 0 is root node
    float3 nextNodePos;

    // root to leaf strackless traversal
    //
    while(!node.Leaf())
    {
        float3 diff = lookPos - nodePos;
        int childOffset = GetOctIndex(diff);
        node = GetOctreeNode(node.GetOffsetToChild() + childOffset);
    }
}
```

```

        nextNodePos.x = diff.x > 0.0f ? 1.0f : -1.0f;
        nextNodePos.y = diff.y > 0.0f ? 1.0f : -1.0f;
        nextNodePos.z = diff.z > 0.0f ? 1.0f : -1.0f;

        nodePos += 0.25f*nodeSize*nextNodePos;
        nodeSize *= 0.5f;
    }

    // interpolation
    //
    float3 E = make_float3(0,0,0);
    float summWt = 0.0f;

    for(int i = 0; i < node.GetNumPoints(); i++)
    {
        float3 pointPos = node.records(i).pos;
        float3 pointNorm = node.records(i).norm;
        float validityRadius = node.records(i).r;

        // perr = ||p1-p2||/(Ri)
        // nerr = (1-sqrt(n1*n2))/(sqrt(1-cos(10)))
        //
        float perr = length(lookPos-pointPos)/validityRadius;
        float nerr = 9.0124*sqrtf(1.f - dot(lookNorm, pointNorm));
        float err = fmaxf(perr, nerr);

        if (err < 1.0f)
        {
            float wt = (1.0f - err);
            summWt += wt;
            E += wt*color;
        }
    }

    E *= (1.0f/summWt);
    float errorEstimate = 1.0f/(summWt + epsilon);

    return float4(E, errorEstimate);
}

```

Listing 34: Поиск в окто-дереве множественными ссылками и интерполяция в кэше освещенности.

После обхода всех точек в цикле, полученную освещенность E нужно делить на сумму весов $summWt$, чтобы получить правильное значение, учтя вклады от каждой точки соответственно их весам. Функция `GetOctIndex` вычисляет индекс узла на основании 'линейного' представления окто-деревя в памяти (листинг 35). Функция `'GetOffsetToChild'` возвращает

смещение в массиве узлов окто-дерева до первого потомка. Следующие 7 потомков лежат сразу за первым в соответствии с листингом 35 и рисунком 61.

```
int GetOctIndex(float3 pos)
{
    int x1 = (int)(pos.x < 0);
    int y1 = (int)(pos.y < 0);
    int z1 = (int)(pos.z < 0);

    return x1 + (y1 << 1) + (z1 << 2);
}
```

Listing 35: Вычисления индекса узла окто-дерева.

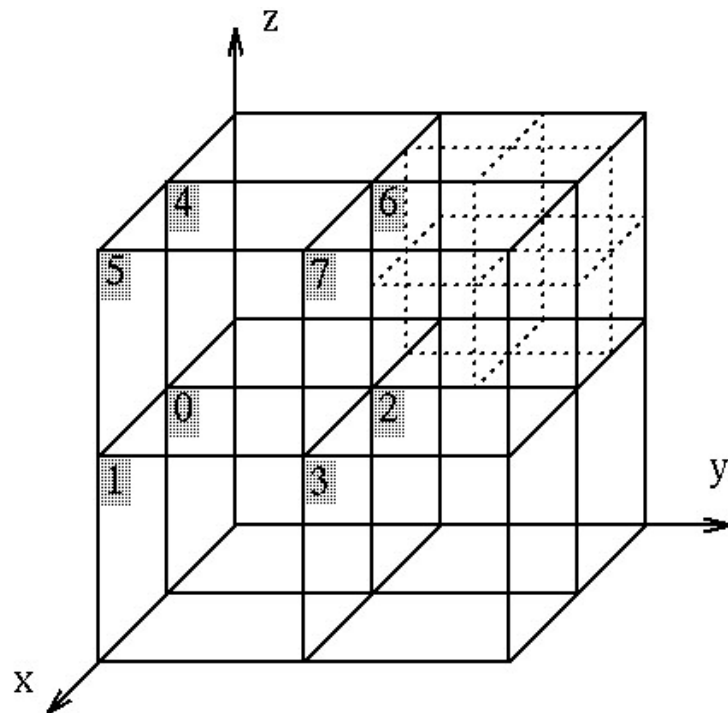


Рис. 61: Нумерация узлов окто-дерева в соответствие с функцией GetOctIndex из листинга 35.

0.9.4 Резюме по кэшу освещенности

Кэширование вторичного освещения - весьма эффективная техника ускорения вычисления глобально освещения, позволяющая снизить количество необходимых вычислений практически на 3 порядка (для больших

разрешений). Допустим имеется изображение размером 1024×1024 пиксела. Если вычислять вторичное освещение в каждой точке, то потребуются суммарно протрассировать порядка 10^9 лучей только для вычисления вторичной освещенности. При использовании кэша освещенности количество точек, в которых необходимо вычислить вторичную освещенность сокращается с миллиона до нескольких тысяч. Соответственно, число лучей падает с 10^9 до $10^6 - 10^7$.

Однако, кэширование освещенности имеет определенные недостатки:

1. Кэш освещенности хорошо работает только на сценах с преимущественно гладкими поверхностями. На сценах с большим количеством геометрических деталей (например карты нормалей для имитации микрорельефа) кэш освещенности не даст преимущества и может даже замедлить процесс рендеринга.
2. Кэш освещенности достаточно сложно сделать параллельным.
3. Кэш освещенности дает мерцание при анимации. Если стохастическая трассировка путей дает шум, который эффективно удаляется при помощи размытия в движении, то пятна вызванные кэшированием освещенности удалить гораздо труднее.
4. Кэш освещенности все же снижает точность решения и является смещенным методом. Чем ниже точность, тем больше выигрыш в скорости, но тем более заметны артефакты.

Литература

- [1] Whitted Turner. An improved illumination model for shaded display // Commun. ACM. New York, NY, USA, 1980. Т. 23, № 6. С. 343–349. URL: <http://doi.acm.org/10.1145/358876.358882>.
- [2] Pharr Matt, Humphreys Greg. Physically Based Rendering, Second Edition: From Theory To Implementation. 2nd изд. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.
- [3] Jakob Wenzel. Mitsuba Renderer. 2013. URL: <http://www.mitsuba-renderer.org/>.
- [4] Quinlez Icigo. Computer graphics, Fractals and Demoscene. 2013. URL: <http://www.iquilezles.org/www/index.htm>.
- [5] Fujimoto A., Tanaka Takayuki, Iwata K. Tutorial: computer graphics; image synthesis / под ред. Kenneth I. Joy, Charles W. Grant, Nelson L. Max [и др.]. New York, NY, USA: Computer Science Press, Inc., 1988. С. 148–159. URL: <http://dl.acm.org/citation.cfm?id=95075.95111>.
- [6] MacDonald David J., Booth Kellogg S. Heuristics for ray tracing using space subdivision // Vis. Comput. Secaucus, NJ, USA, 1990. Т. 6, № 3. С. 153–166. URL: <http://dx.doi.org/10.1007/BF01911006>.
- [7] Ernst Manfred, Greiner Gunther. Early Split Clipping for Bounding Volume Hierarchies // Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing. RT '07. Washington, DC, USA: IEEE Computer Society, 2007. С. 73–78. URL: <http://dx.doi.org/10.1109/RT.2007.4342593>.
- [8] Ize Thiago, Wald Ingo, Parker Steven G. Ray tracing with the BSP tree // Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing. 2008.

- [9] Aila Timo, Laine Samuli. Understanding the efficiency of ray traversal on GPUs // Proceedings of the Conference on High Performance Graphics 2009. HPG '09. New York, NY, USA: ACM, 2009. C. 145–149. URL: <http://doi.acm.org/10.1145/1572769.1572792>.
- [10] Nicodemus Fred E. Directional Reflectance and Emissivity of an Opaque Surface // Applied Optics. 2009. T. 4, № 7. C. 767–773.
- [11] Kajiya James T. The rendering equation // SIGGRAPH Comput. Graph. New York, NY, USA, 1986. T. 20, № 4. C. 143–150. URL: <http://doi.acm.org/10.1145/15886.15902>.
- [12] Соболев И.М. Численные методы Монте-Карло. 1 изд. М.: Наука., 1973.
- [13] Suffern Kevin. Ray Tracing from the Ground Up. Natick, MA, USA: A. K. Peters, Ltd., 2007.
- [14] Shirley Peter, Wang Changyaw. Distribution Ray Tracing: Theory and Practice // In Proceedings of the Third Eurographics Workshop on Rendering. 1992. C. 33–43.
- [15] Veach Eric. Robust monte carlo methods for light transport simulation. Ph.D. thesis. Stanford, CA, USA: Stanford University, 1998. AAI9837162.
- [16] Heckbert Paul S. Adaptive radiosity textures for bidirectional ray tracing // SIGGRAPH Comput. Graph. New York, NY, USA, 1990. T. 24, № 4. C. 145–154. URL: <http://doi.acm.org/10.1145/97880.97895>.
- [17] Боголепов Денис. Методы глобального освещения для интерактивного синтеза изображений сложных сцен на графических процессорах. Ph.D. thesis. НГГУ им. Лобачевского, Нижний Новгород, Россия: Нижний Новгород, 2013.
- [18] Georgiev Iliyan, Křivánek Jaroslav, Slusallek Philipp. Bidirectional light transport with vertex merging // SIGGRAPH Asia 2011 Sketches. SA '11. New York, NY, USA: ACM, 2011. C. 27:1–27:2. URL: <http://doi.acm.org/10.1145/2077378.2077412>.
- [19] Veach Eric, Guibas Leonidas J. Metropolis light transport // Proceedings of the 24th annual conference on Computer graphics and interactive techniques. SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997. C. 65–76. URL: <http://dx.doi.org/10.1145/258734.258775>.

- [20] Cline David, Egbert Parris. A Practical Introduction to Metropolis Light Transport: Tech. Rep.: : Brigham Young University, 2005.
- [21] Jensen Henrik Wann, Christensen Per. High quality rendering using ray tracing and photon mapping // ACM SIGGRAPH 2007 courses. SIGGRAPH '07. New York, NY, USA: ACM, 2007. URL: <http://doi.acm.org/10.1145/1281500.1281593>.
- [22] Wald Ingo, Gunther Johannes, Slusallek Philipp. Balancing Considered Harmful – Faster Photon Mapping using the Voxel Volume Heuristic // Computer Graphics Forum. 2004. T. 22, № 3. С. 595–603. Proceedings of Eurographics.
- [23] Tabellion Eric, Lamorlette Arnauld. An approximate global illumination system for computer generated films // ACM SIGGRAPH 2004 Papers. SIGGRAPH '04. New York, NY, USA: ACM, 2004. С. 469–476. URL: <http://doi.acm.org/10.1145/1186562.1015748>.
- [24] Востряков Константин. Глобальное освещение с помощью октанных текстур // Материалы 16-ой международной конференции Графи-кон'2006. GraphiCon'06. Новосибирск, Россия: Графикон, 2006.
- [25] Hachisuka Toshiya, Ogaki Shinji, Jensen Henrik Wann. Progressive photon mapping // ACM Trans. Graph. New York, NY, USA, 2008. Т. 27, № 5. С. 130:1–130:8. URL: <http://doi.acm.org/10.1145/1409060.1409083>.
- [26] Hachisuka Toshiya, Jensen Henrik Wann. Stochastic progressive photon mapping // ACM Trans. Graph. New York, NY, USA, 2009. Т. 28, № 5. С. 141:1–141:8. URL: <http://doi.acm.org/10.1145/1618452.1618487>.
- [27] State of the Art in Photon Density Estimation / Toshiya Hachisuka, Wojciech Jarosz, Guillaume Bouchard [и др.] // ACM SIGGRAPH 2012 Courses. SIGGRAPH '12. New York, NY, USA: ACM, 2012. С. 6:1–6:469. URL: <http://doi.acm.org/10.1145/2343483.2343489>.
- [28] Suykens Frank, Willems Yves D. Density Control for Photon Maps // Proceedings of the Eurographics Workshop on Rendering Techniques 2000. London, UK, UK: Springer-Verlag, 2000. С. 23–34. URL: <http://dl.acm.org/citation.cfm?id=647652.732120>.
- [29] Krivánek Jaroslav, Gautron Pascal. Practical Global Illumination with Irradiance Caching (Synthesis Lectures in Computer Graphics and Animation). Morgan and Claypool Publishers, 2009.