Задание #1: Трассировка лучей с использованием функций расстояний

Автор задания: Фролов В. А.

Аннотация

Цель задания - закрепить на практике основы построения изображений.

Основной фокус задания:

- Работа с алгоритмами реалистичной компьютерной графики.
- Изучение алгоритма трассировки неявно заданных поверхностей;
- Изучение основ работы с OpenGL 3.

Обязательная часть

Необходимо реализовать алгоритм трассировки лучей для сцены, составленной их неявно-задаваемых поверхностей. Обязательно использовать ***минимум 3 различных типа примитива***.

Формулы функций расстояний для примитивов можно посмотреть, например, здесь: http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm. Общее количество примитивов должно быть 5 или более, также обязательно использование минимум 3 различных материалов.

Вместо 3 или более различных примитивов разрешается задать фрактальные поверхности (http://www.subblue.com/projects/mandelbulb). В этом случае требование с 3 материалами не обязательное.

В сцене ***должно быть хотя бы 2 источника света***. При наличии Ambient Occlusion, Environment Light без текстуры – то есть просто условный источник константного цвета считается за отдельный источник света. С текстурой тем более считается.

Рендеринг ***не должен занимать больше 30 секунд***. В случае если вы опасаетесь, что при значительном количестве дополнительно реализованных эффектов скорость упадет ниже допустимого порога, необходимо реализовать прогрессивный (постепенный, уточняющий картинку с каждым проходом) рендеринг.
Для того чтобы реализовать обязательную часть вам достаточно прочитать пункт “Трассировка

лучей” и пункт “описание алгоритма distance-aided ray marching”. Остальное неотребуется только для реализации дополнительных частей задания.

Обязательная часть – **15 баллов**.

Дополнительная часть

- Резкие тени (+1)
- Мягкие тени (+2-3)
- Отражения (+1)
- Преломления (+2-3)
- Анти-алиасинг (+1)
- Туман (+1)
- Ambient Occlusion (+2-4)
- Фрактальные поверхости (+2-6)

  +2 за каждый тип фрактала. Внимание! Типы фрактальный поверхностей должны быть разными! За разные типы засчитываются лишь ***принципиально разные формулы***, а не одна и та же формула с разными числами.

- Визуализация ландшафта или любой другой карты высот при помощи алгоритма Parallax Occlusion Mapping (+4-6) в зависимости от реалистичности получаемой картинки. http://www.iquilezles.org/www/articles/terrainmarching/terrainmarching.htm
- Окружение в виде текстурированной плоскости (+1)
- Окружение в виде текстурированного куб-мапа (+2)
- Сцена сделана реалистично, содержит множество интересных объектов и материалов: (+1-2)
- Использование конструктивной сплошной геометрии (Constructive Solid Geometry, CSG) (+1-2) Формулы комбинирования смотреть по той же ссылке .../distfunctions.htm.
- Интерактивное перемещение по сцене и поворот камеры (как в шутере, по клавишам WASD + мышка или просто поворот мышкой как в предоставляемом сэмпле) (+2).
- Непредусмотренный заданием бонус (+1-2 на усмотрении проверяющего)

Для всех используемых текстур обязательно применять билинейную, трилинейную или анизотропную фильтрацию. Оценки за любые текстурированные объекты, без фильтрации будут снижены.

Разрешается реализовать несколько различных сцен с разными эффектами (в случае если все сложно показать на одной). Можно делать отдельный бинарный файл для каждой сцены, либо скрипты для запуска программы с разными сценами.

#Разрешается выполнять задание в среде shadertoy.

#Разрешается выполнять задание на CPU, на любом языке программирования. Обратите внимание, однако, на правила сдачи.

Сборка и запуск должны быть тривиальными для проверяющего.

Материал для выполнения заданий

Трассировка лучей. Введение.

Алгоритм трассировки лучей в самом общем случае выглядит следующим образом: из виртуального глаза через каждый пиксел изображения испускается луч и находится точка его пересечения с поверхностью сцены (для упрощения изложения мы не рассматриваем объемные эффекты вроде тумана). Лучи, выпущенные из глаза называют первичными. Допустим, первичный луч пересекает некий объект 1 в точке H1 (рис. 1).

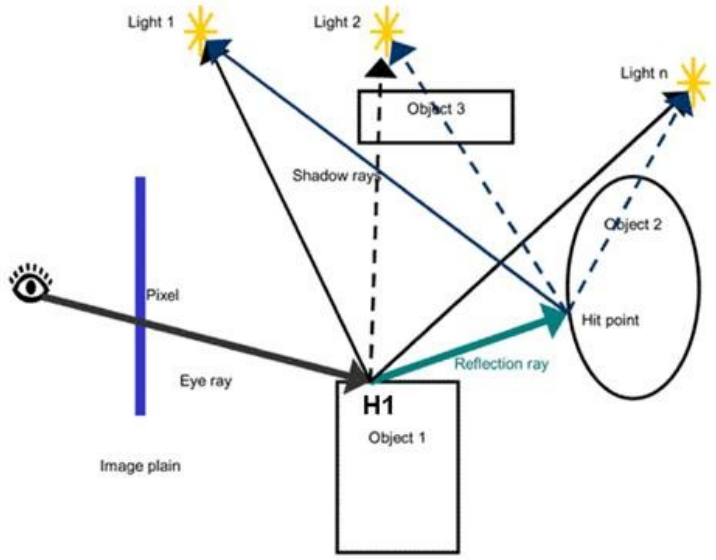

Рисунок 1: Обратная трассировка лучей.

Далее 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. Это позволяет сказать, освещается ли данная точка конкретным источником. Если теневой луч находит пересечение 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 (Фонг, Кук-Торранс и.т.д.). Освещение со всех видимых (из точки H1) источников света 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.

```
// Алгоритм трассировки лучей
//
float3 RayTrace(const 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;

  // затенение, используется локальная модель освещения для источников света
  //
  for(int i=0;i < lights.size();i++)
   if(Visible(hit_point, lights[i]))
     color += Shade(hit_point, hit.normal);

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

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

  return color;
}
```

Листинг 1. Алгоритм обратной трассировки лучей.

Поясним фрагмент программы (листинг 1). Луч представлен двумя векторами. Первый вектор – pos – точка испускание луча. Второй – dir – нормализованное направление луча. Цвет – вектор из трех чисел – синий, красный, зеленый. 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 параметрическому уравнению прямой с условием t>0).

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).

При расчёте освещения по локальной модели внутри функции Shade:

яркость источника нужно делить на квадрат расстояния до него!

Оставим пока что это правило без объяснения.

После, 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 refraction могут быть как монохромными так и цветными. Всё зависит от того, какая используется математическая модель для представления материалов.

Теневые лучи можно сделать цветными. Такие лучи используются, если один объект перекрывается другим прозрачным объектом. В таком случае рассчитывается длина пути теневого луча внутри прозрачного объекта, и тень может приобрести какой-либо оттенок исходя из закона Бугера-Ламберта-Бера. Разумеется, тени, рассчитанные таким образом корректны, только если прозрачный объект, отбрасывающий тень, имеет очень 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. Каустики рассчитываются отдельно с помощью метода фотонных карт или других более продвинутых алгоритмов. Типичный пример каустика – солнечный зайчик от стакана воды, когда через него просвечивает солнце.

Удаление хвостовой рекурсии

В случае, если отраженный (или преломленный) луч всегда один (то есть не может быть так, что имеется одновременно и отраженный и преломленный луч), то рекурсивный вызов в листинге 1, можно заменить на простой цикл.

```
float3 color(0,0,0);
float3 k(1.0f, 1.0f, 1.0f);

for(int i=0;i < MAX_REFLECTION_DEPTH;i++)
{
  Hit hit = RaySceneIntersection(ray);
  if (!hit.exist)
    return color;

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

  // затенение, используется локальная модель освещения для источников света
  //
  for(int i=0;i < lights.size();i++)
   if(Visible(hit_point, lights[i]))
     color += k*Shade(hit_point, hit.normal);

  if (hit.material.reflection <= 0)
    break;

  ray = reflect(ray, hit);
  k *= hit.material.reflection;
}
```

Листинг 2. Алгоритм обратной трассировки лучей без рекурсии.

Это весьма актуально при реализации на GPU, где рекурсия не поддерживается изначально (хотя может быть смоделирована через стэк).

Трассировка с использованием функций расстояний

В данном задании, вам **не нужно** искать пересечение луча и примитива так, как это делают обычно – вычисляя геометрически пересечения прямой и поверхности явно. 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).

Пусть существует функция f(x,y,z), задающая поверхность уравнением вида f(x,y,z) = 0. Для краткости можно будем писать f(p) = 0.

Функции есть тут: http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm

Если в функцию f вместо p подставить, некоторую точку, не лежащую на поверхности, мы получим некое не равное нулю число. По определению самой функции, что это число будет представлять собой знаковое (хотя это может зависеть от формулы) расстояние до поверхности.

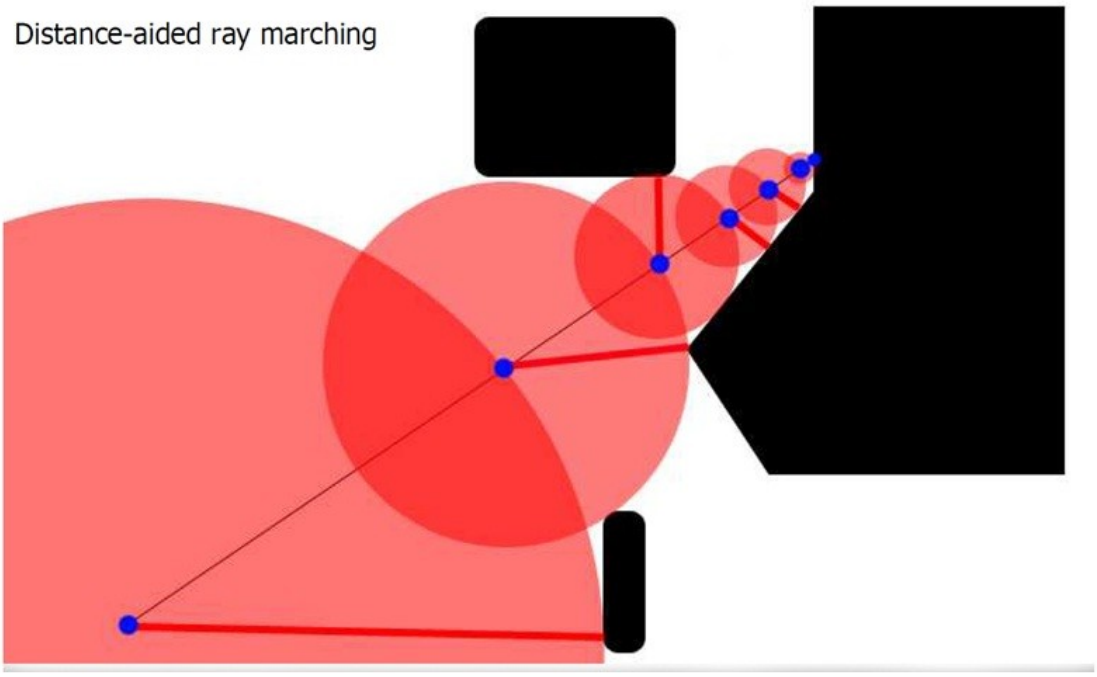

Рисунок 2: Трассировка с использованием функций расстояний.

Дальше мы ***шагаем по лучу на полученное расстояние*** (поэтому такой алгоритм называется ray marching), и применяем эту процедуру еще раз, до тех пор, пока наше расстояние

не станет меньше заданного порога (рис. 2). Следует отметить, что размер шага было бы хорошо ограничивать снизу некоторой константой, для т. к. в противном случае трассировка будет очень медленной вблизи границ объектов.

Еще одна тонкость, для того чтобы не продолжать реймарчинг до бесконечности, вам необходимо в самом начале найти пересечение луча и ограничивающего всю сцену bounding box-a. Как только вы выходите за этот бокс (t > tmax) – остановить реймарчинг. Либо вы просто можете ограничить максимальное число шагов по лучу какой-либо достаточно большой константой.

Вычисление нормали

Нормаль есть не что иное, как градиент к поверхности, заданной нашей функцией f(p) = .
Вычисляем ее численно.

```
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));
}
```

Листинг 3. Вычисление нормали к поверхности.

Фото-реалистичная визуализация. Погружение.

Однако, трассировка лучей в том виде, в котором она была описана не дает фото-реалистичное изображение. Во-первых, материалы реального мира в силу наличия микрорельефа обычно не отражают свет строго в одном направлении (Ламбертовские поверхности, например, равномерно рассеивают свет по всем направлениям). Во-вторых, источники света далеко не всегда имеют точечные размеры (тени от таких источников света получаются мягкие, с плавным переходом свет-тень).

Далее, изображение хранится в памяти как двумерная матрица пикселей, это бесспорно. Однако, считается, что изображение предметов реального мира это не просто набор пикселей. Авторы [Physically Based Rendering] и [Ray Tracing From The Ground Up] склонны рассматривать изображение как непрерывную двумерную функцию. То, что мы видим на экране – приближение этой функции, ее дискретизованное в заданном разрешение представление.

Трассировка лучей в самом общем понимании - это метод, позволяющий восстановить значения этой функции с помощью точечных сэмплов, то есть значений этой функции в точках. Поэтому для того чтобы получить изображение, лишенное алиасинга, необходимо как минимум делать более чем 1 сэмпл на пиксел.

Сосредоточимся теперь на том, как получать значения функции изображения в точках. Каждой точке в двухмерном пространстве экрана соответствует точка на некоторой поверхности (эту точку можно найти с помощью прослеживания пути луча, выпущенного из виртуального глаза через точку экрана в сцену; назовем ее точкой x). Освещенность точке поверхности x вычисляется при помощи интеграла следующего вида:

$$I(\phi_r, \theta_r) = \int\limits_{\phi_i \theta_i} L(\phi_i, \theta_i) R(\phi_i, \theta_i, \phi_r, \theta_r) cos(n, l_{\phi_i, \theta_i}) \mathrm{d}\phi_i \mathrm{d}\theta_i$$

Формула 1. (φi, θi) – направление на источник. (φ_r, θ_r) – направление на наблюдателя.

$L(\varphi i, \theta i)$ – это функция, описывающая общее освещение, падающее в точку x под всеми возможными углами в пределах полусферы. $R(\varphi_i, \theta_i, \varphi_r, \theta_r)$ – BRDF (bidirectional reflectance

distribution function – двунаправленная функция распределения отражения или ДФО). Эта функция полностью описывает свойства взаимодействия конкретной поверхности со светом.

Фактически, ДФО просто связывает по некоторому закону интенсивность и угол падающего света с интенсивностью и углом отраженного или пропущенного прозрачной поверхностью света (рис. 3).

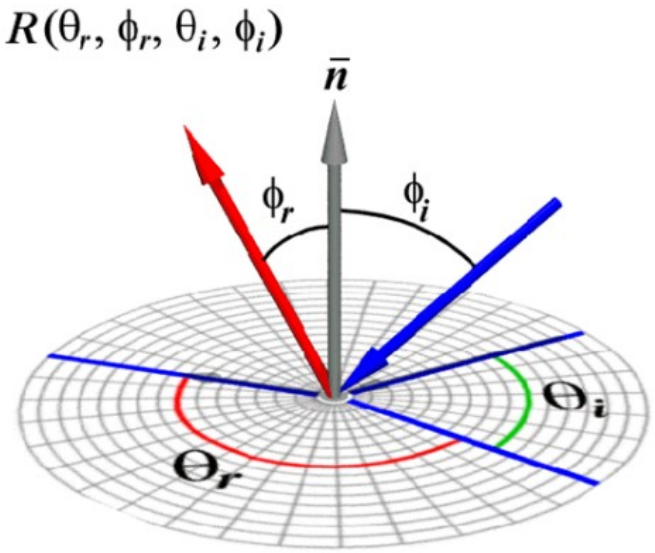

Рисунок 3: BRDF (ДФО).

Вычисляемая интегрированием интенсивность освещения в точке $I(\varphi_r, \theta_r)$ является не числом, а функцией. Другими словами, она дает значения интенсивности света, отражаемой поверхностью под разными углами (собственно пара φ_r, θ_r). 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, обусловленную свойствами отражения поверхности (материала) в точке x. Хотя если мы рассматриваем луч, который трассировали через пиксел из виртуального глаза, то направление конечно одно и задано этим лучом.

Обратите внимание что:

Одна из наиболее часто используемых BRDF, представляющая полностью диффузную поверхность – BRDF Лабмерта. Это константа! Скалярное произведение нормали и направления на источник **зачёркнуто!** Это сделано по той причине, что ***скалярное произведение*** --- это на самом деле ***косинус в интеграле***. А не часть BRDF Ламберта, которое представляет из себя константу (при этом идеально-зеркальное отражение всегда рассматривается как специальный случай, в котором этого косинуса нет). К сожалению, похожая ошибка допущена во многих источниках.

```
float3 EvalDiffuseBRDF(float3 l, float3 norm, float3 color)
{
  return color*max(dot(norm,l),0);
}
```

Листинг 4. Вычисление BRDF Лабмерта.

Итак, интеграл освещённости (формула 1) берется их тех соображений, что освещенность в точке поверхности складывается из света, падающего на поверхность со всех направлений и отражающегося по какому-то определенному закону (какому именно, устанавливает функция 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.

Существуют различные способы вычисления интеграла освещённости. Одни из них более точные, другие менее точные, но более быстрые. Среди точных методов – наиболее популярные это Монте-Карло трассировка лучей, трассировка путей (прямая, обратная, двунаправленная) и фотонные карты.

Локальные модели освещения. Shade и квадратичное затухание.

Считать интеграл освещенности точно – довольно дорого. Самая простая аппроксимация заключается в том, чтобы рассматривать только прямой свет от источников. Непрямой свет принимается за константу (называемую обычно ambient). Это и называется ***локальной моделью***

$$I = emissive + ambient + diffuse*att + specular*att$$

$$att = \frac{1}{k_C + k_L * d + k_Q * d^2}$$

d is the distance from light source to current vertex

k_C is the constant attenuation

k_L is the linear attenuation

k_Q is the quadratic attenuation

Иллюстрация 1: Не физ. корректный способ рассчёта локальной освещённости.

освещения. Расчёт по такой модели обычно называется ***затенением (Shade)***. Популярной моделью учёта яркости является квадратичное затухание с добавлением линейной компоненты и константы (http://in2gpu.com/2014/06/19/lighting-vertex-fragment-shader/):

Обратите внимание, что формула для переменной ***att*** в действительности ***не корректна*** с точки зрения физики (и даже, скорее, с точки зрения банальной математики). Для того чтобы она была корректна, константы k_c и k_l должны быть приравнены ***к нулю***, а константа k_q ***к единице***. То есть затухание ***строго квадратично***. Отчего так? В воздухе свет не затухает – эта формула вообще не учитывает затухание в среде. Тогда откуда берётся квадрат?

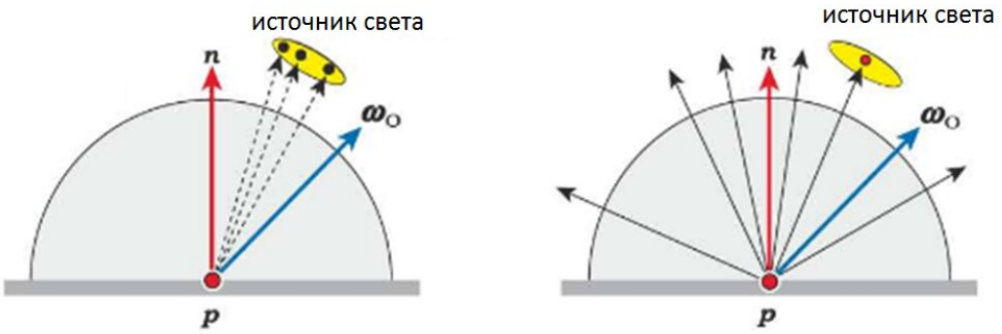

Иллюстрация 2: Расчёт освещённости от площадного источника света.

В действительности он берётся из корректного способа вычисления интеграла освещённости. Предположим, что наш источник света не точечный, а площадный (иллюстрация 2). Если мы будем вычислять интеграл освещённости честно, бросая лучи на полусфере, тогда некоторое небольшое их количество попадёт в источник. А остальные пролетят мимо.

Теперь вопрос: как зависит от расстояния до источника процент лучей, которые будут попадать в источник? Ответ можно получить, если спроецировать источник на полусферу. Тогда мы придём к $1/d^2$. Вот откуда берётся квадратичное затухание. Таким образом, $1/d^2$ --- это чисто геометрический фактор, определяющий какой процент лучей попал бы в источник, если бы мы всё делали правильно. Ведь когда мы считаем освещение по локальной модели, мы все лучи выпускаем точно в источник. Поэтому нам необходим т. н. «регуляризатор» – коэффициент $1/d^2$**,** который позволяет получить правильный результат, ***как если бы*** мы считали интеграл освещённости честно. Зато теперь мы можем корректно рассчитывать освещение даже от точеных источников, несмотря на то что вероятность попадания лучом в точку равна нулю.

Попытка добавить константный или линейный члены затухания связаны с желанием разработчиков получить глобальную освещённости бесплатно, путём некой очень простой аппроксимации. Либо с попыткой моделирования некоторого сфокусированного или направленного источника, у которого тем нее менее хочется сделать затухание по той или иной причине. Вы ***можете использовать любые локальные модели освещения***, если они на ваш взгляд повышают реалистичность изображения. В задании вы ничем не ограничены.

Апроксимация Ambient Occlusion для расчета вторичной освещенности

Как мы уже отмечали, считать интеграл освещенности точно – довольно дорого. Однако, существует ряд аппроксимаций, позволяющих примерно оценить его значение, не выполняя честную трассировку лучей по всем направлениям. Во-первых, следует разделить непрямое освещение на “резко-меняющееся” и “плавно-меняющееся” – соответственно высокочастотную компоненту и низкочастотную компоненты. Высокочастотную составляющую считают, как и раньше, трассировкой лучей. Однако, для нее количество лучей обычно небольшое (если только не рассматриваются мягкие тени и glossy reflections). Например, вам нужен всего 1 отраженный луч чтобы посчитать зеркальные отражения на поверхности.

Для низкочастотной компоненты, обуславливающей диффузные пере-отражения света (которую дольше всего считать), можно применить аппроксимацию, позволяющую получить очень похожий на правду, но не совсем точный результат.

Идея ***Ambient Occlusion*** похожа на трассировку лучей по всем направлениям, но более простая в вычислительном плане. Мы предполагаем, что вторичный, диффузно переотраженный ***свет относительно равномерно распределен в пространстве***. Такое освещение можно заменить неким источником, называемым Environment Light. Он может быть задан панорамой, куб-мапом, или в простейшем случае просто – числом (3 числами – для красного, синего и зеленого). То есть мы говорим, пусть весь наш вторичный свет от относительно-далеко стоящих поверхностей задан некой константой g_environmentLightColor.

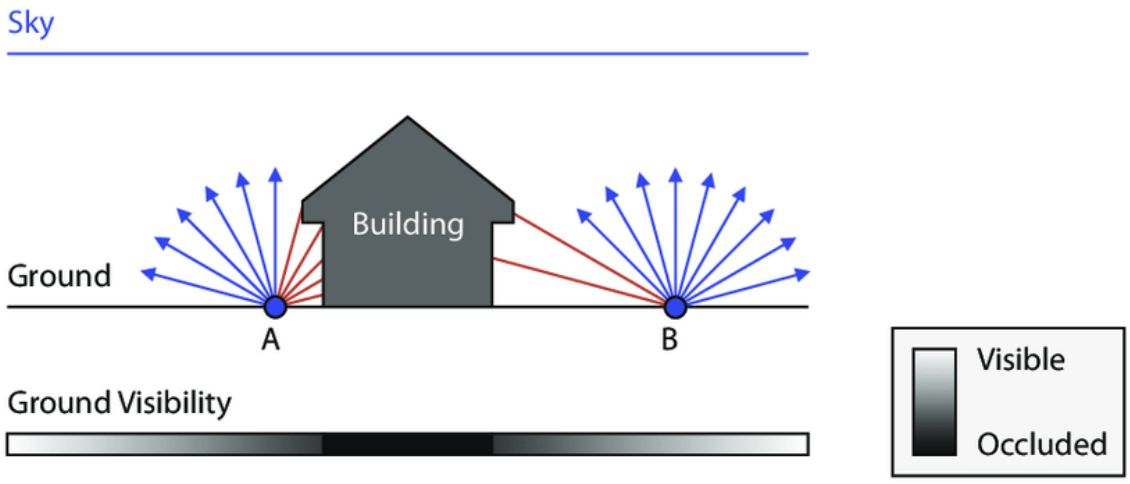

Рисунок 4: Иллюстрация апроксимации Ambient Occlusion.

Тогда, в той точке, где мы хотим посчитать освещение, мы испускаем лучи по всем направлениям, но не трассируем их в сцену, а лишь пытаемся вычислить, какой процент нашего *Environment Light* закрыт близлежащими поверхностями.

То есть, фактически мы просто ограничили длину трассировки луча. Однако, во многих случаях Ambient occlusion можно вычислить, ***не прибегая к трассировке лучей***, заменив ее более простой апроксимацией, ведь нам не нужно в действительности находить пересечения, необходимо лишь знать - какой процент (примерно) нашего воображаемого источника закрыт.

Аппроксимация аппроксимации Ambient Occlusion для неявно-задаваемых поверхностей

// да, именно так, это двойная аппроксимация :)

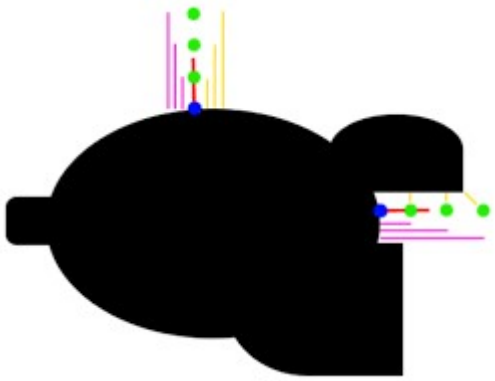

Рисунок 5: Иллюстрация метода аппроксимации вычисления Ambient Occlusion. Желтые линии показываеют приблизительное расстояние до близлежащей геометриии. Чем оно меньше, тем больше синяя точка закрыта от внешнего освещения.

К сожалению, оказывается, что даже Ambient Occlusion честно вычислить довольно трудоемко, особенно если речь идет об интерактивной визуализации. Однако, в случае неявно задаваемых поверхностей, существует упрощение, довольно точно приближающее честный Ambient Occlusion. Идея состоит в том, чтобы взять несколько точек в направлении нормали к поверхности (5-10) …. и, посмотрев в них расстояние до поверхности, оценить, насколько эта точка закрыта близлежащей геометрией. ***Это как раз наш случай***, потому что у нас есть такая функция – это просто левая часть уравнения поверхности f(x,y,z) = 0. То есть это наша функция f(p).

Краткое введение в OpenGL3 и предоставленный шаблон

В предоставленном шаблоне используется OpenGL3 для демонстрации простого рей-марчинга в кубе с константным аккумулированием цвета каждый шаг (чтобы изобразить туман).

Внимание: не делайте так в задании, это просто пример!

Разрешается реализовать задание с использованием сайта shadertoy или целиком на CPU.

В предоставленном шаблоне рисуется полно-экранный прямоугольник (Full Screen Quad). В каждом пикселе выполняется фрагментный шейдер, который и реализует рей-марчинг.

#Задание координат квада:

```
float quadPos[] =
    {
      -1.0f,  1.0f, // v0 - top left corner
      -1.0f, -1.0f, // v1 - bottom left corner
      1.0f,  1.0f,  // v2 - top right corner
      1.0f, -1.0f   // v3 - bottom right corner
    };
```

#Загрузка квада на GPU и создание VAO для него:

```
  GLuint g_vertexBufferObject;  // это буффер (VBO), он хранит вершины квада в памяти GPU
  GLuint g_vertexArrayObject;   // это легковесный объект, который только ссылается на VBO
  {
   …
  }
```

#Передача констант в шейдер (ширина/высота и матрица):

```
 program.SetUniform("g_rayMatrix", rayMatrix);
 program.SetUniform("g_screenWidth" , WIDTH);
 program.SetUniform("g_screenHeight", HEIGHT);
```

#Непосредственная отрисовка квада:

```
glBindVertexArray(g_vertexArrayObject);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
```

Шаблон собирается при помощи CMake следующим довольно стандартным образом:

mkdir build

cd build

cmake ..

make

Ссылки

- http://www.iquilezles.org/www/material/nvscene2008/rwwtt.pdf
- http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm
- http://www.iquilezles.org/www/articles/terrainmarching/terrainmarching.htm
- http://www.subblue.com/projects/mandelbulb
- http://www.raytracegroundup.com/
- http://www.pbrt.org/
- http://ray-tracing.ru
- http://code.google.com/p/gl33lessons/
- https://habr.com/ru/post/437714/ (взрыв, 180 строк кода на C++)