Задание 4: Distance-aided Ray Marching

Введение

Начало: 20.04.2018
Конец: 04.05.2018

Автор задания: Фролов В

Цель задания - закрепить на практике основы построения изображений. Основной фокус задания:

- Работа с алгоритмами реалистичной компьютерной графики. Изучение основ фотореалистичной визуализации;
- Изучение алгоритма трассировки неявно заданных поверхностей;
- Изучение основ работы с OpenGL 3 (не обязательная часть).

Обязательная часть

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

Рендеринг не должен занимать больше минуты. 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 – 10 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 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).
- Прогрессивный рендеринг (+2-4, CPU only). Хотя те кто будут делать на GPU тоже могут сделать прогрессивный рендеринг, но для них это будет значительно сложнее и нужно в меньшей степени.
- Хорошая скорость рендеринга (+2-4, многопоточность или использование GPU). Оценивается в зависимости от сложности сцены и реализованных эффектов.

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 “.exe” или “.bat” файл для каждой сцены.

Трассировка лучей

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. 9).

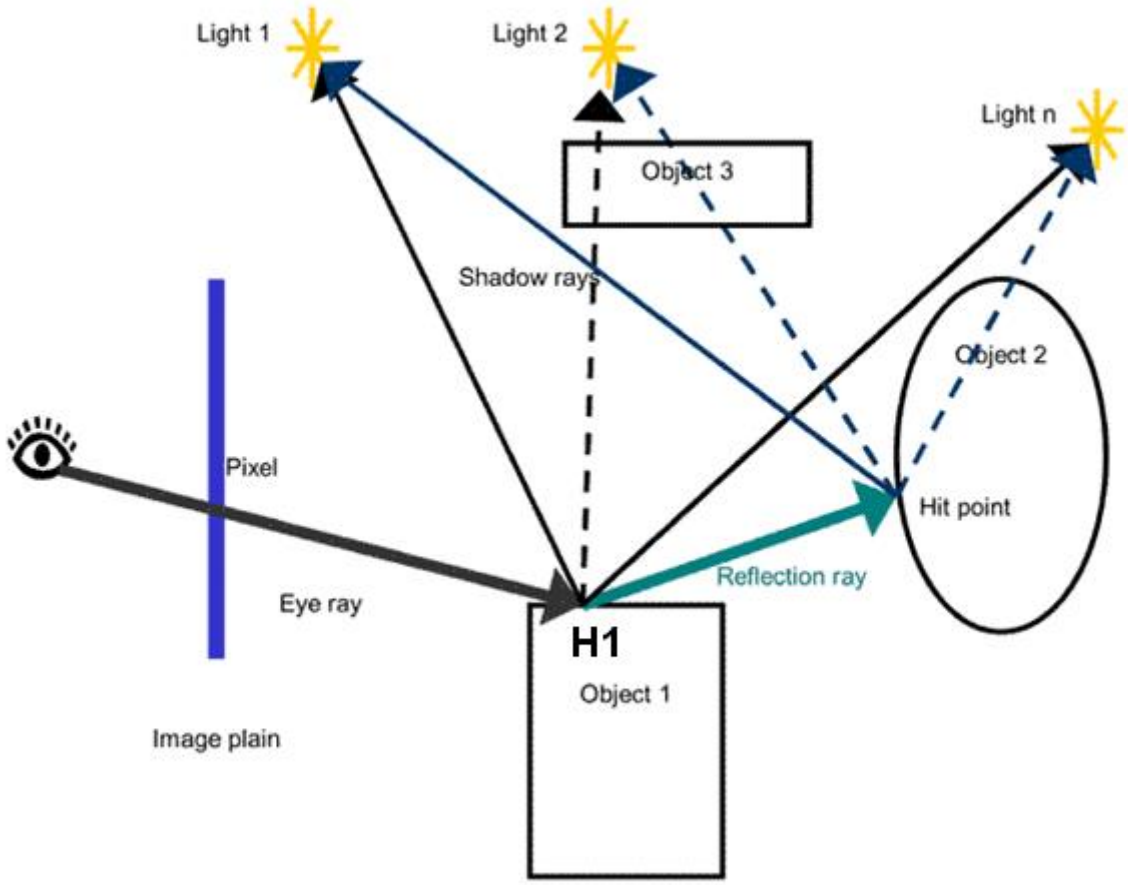


Рисунок 1. Трассировка лучей

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 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). 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.

Иногда теневые лучи бывают цветные. Такие лучи используются, если есть вероятность того, что один объект перекрывается другим прозрачным объектом. В таком случае рассчитывается длина пути теневого луча внутри прозрачного объекта и тень может приобрести какой-либо оттенок (если объект им обладает). 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. 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).

Описание алгоритма Distance-aided Ray Marching

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. Нетрудно заметить, что это число будет представлять собой знаковое (хотя это может зависеть от формулы) расстояние до поверхности.

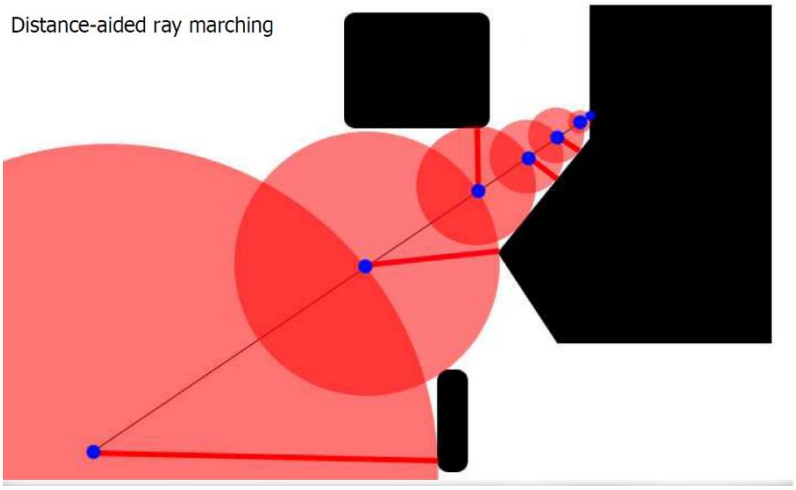


Рисунок 2. Distance-aided ray marching.

Дальше все просто. Мы шагаем по лучу на полученное расстояние и применяем эту процедуру еще раз, до тех пор, пока наше расстояние не станет меньше заданного порога. Следует отметить только, 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. 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));
}
```

Фотореалистичная визуализация

Однако, трассировка лучей в том виде, в котором она была описана не дает фотореалистичное 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 [PBRT] и [RTFGUP] 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 [6]:

$$I\left(\phi_r, \theta_r\right) = \int_{\phi_i} \int_{\theta_i} L\left(\phi_i, \theta_i\right) R\left(\phi_i, \theta_i, \phi_r, \theta_r\right) d\phi_i d\theta_i \quad (1)$$

Рисунок 3. Уравнение светопереноса.

L(φi, θi) – это функция, описывающая общее освещение, падающее в точку x под всеми возможными углами в пределах полусферы. R(φi, θi, φr, θr) – 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.

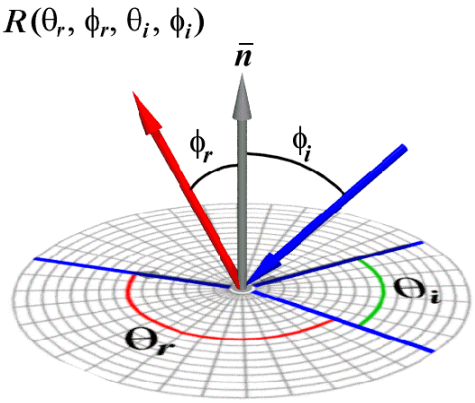


Рисунок 4. Двулучевая Функция Отражательной Способности - ДФО (BRDF).

Вычисляемая интегрированием интенсивность освещения в точке I(φr, θr) 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, 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.

Одна из наиболее часто используемых BRDF, представляющая полностью диффузную поверхность – BRDF Лабмерта

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

Формула (рис. 7) 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 на рис. 7. 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.

Апроксимация Ambient Occlusion для расчета вторичной освещенности

Считать интеграл освещенности точно – довольно дорого. Однако, существует ряд аппроксимаций, позволяющих примерно оценить его значение, не выполняя трассировку лучей по всем направлениям. Во-первых, следует разделить непрямое освещение на “резкоменяющееся” и “плавноменяющееся” – 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 отражения на поверхности.

Для низкочастотной компоненты, обуславливающей диффузные переотражения 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.

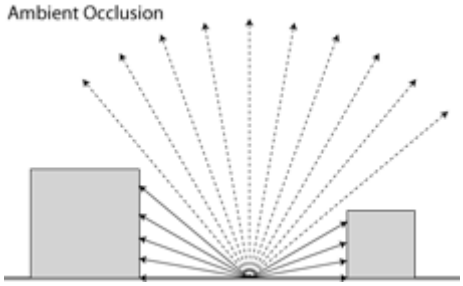


Рисунок 5. Ambient Occlusion algorithm

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.

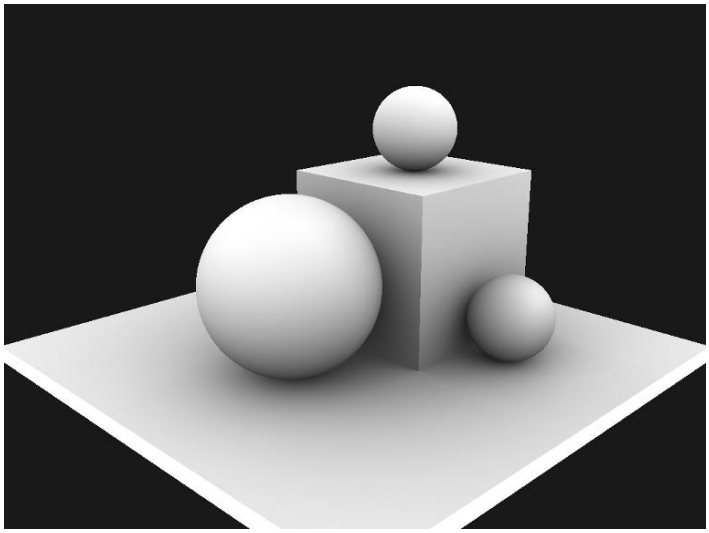


Рисунок 6. Честно посчитанный Ambient Occlusion.

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.

Апроксимация апроксимации Ambient Occlusion для неявно-задаваемых поверхностей

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).

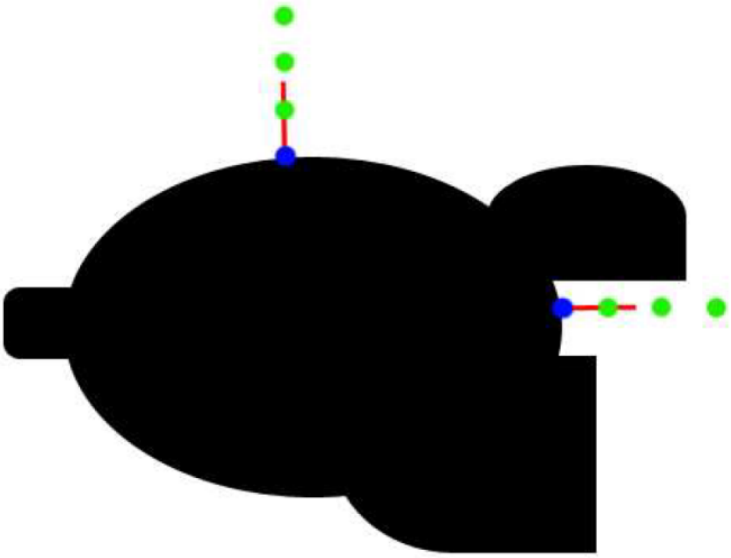


Рисунок 7. Поверхность, в точках которой мы хотим вычислить Ambient Occlusion.

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 знакомая нам функция f(p).

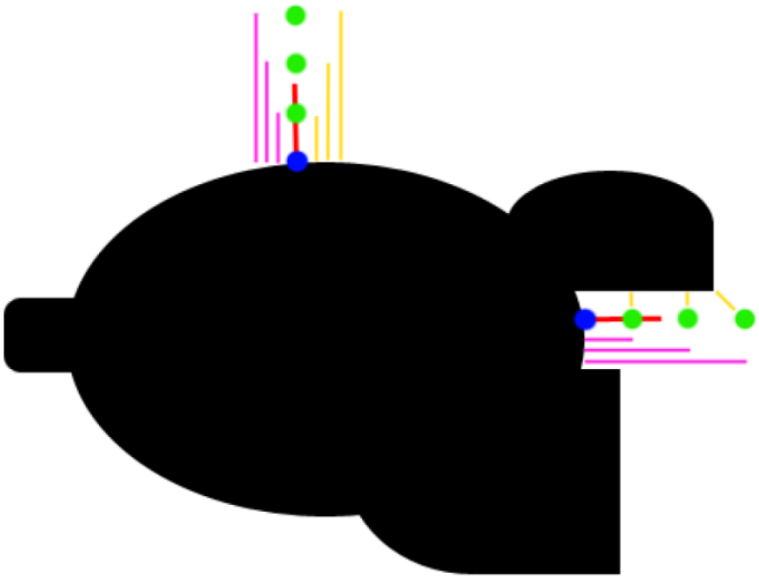


Рисунок 8. Желтые линии показываеют приблизительное расстояние до близлежащей геометриии. Чем оно меньше, тем больше синяя точка закрыта от внешнего освещения.

Ambient Occlusion рассчитанный таким способом, оказывается дешевле чем трассировка первичного луча!

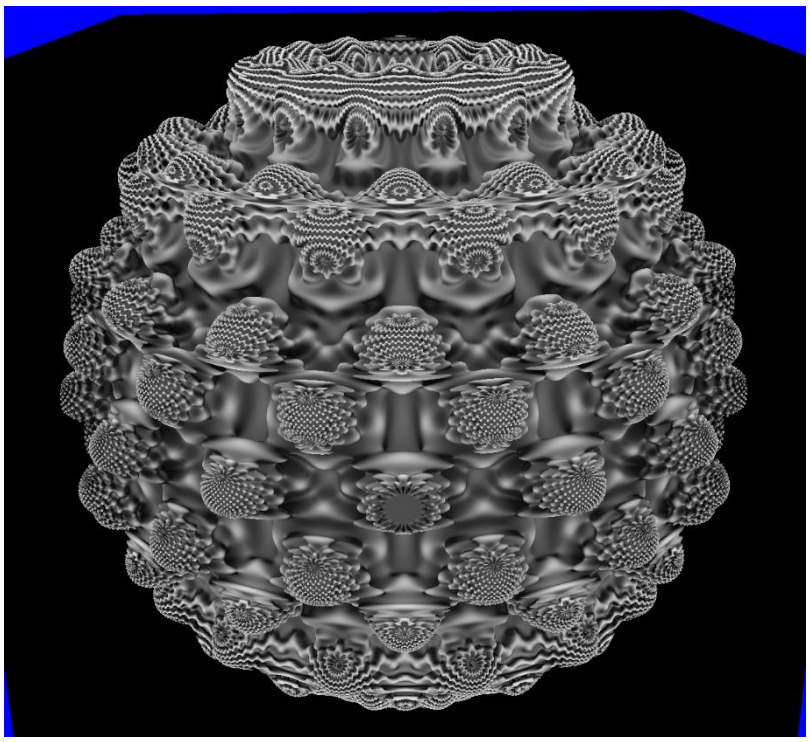


Рисунок 9. Ambient occlusion на фрактальной поверхности. Если его выключить, поверхность станет просто однотонно-белой.

Введение в OpenGL 3

Имеется возможность выполнять данное задание на CPU на любом предусмотренном обычным порядком языке. Вы не обязаны использовать GPU. Однако это желательно.

Для того чтобы облегчить вам реализацию данного задания на GPU и подготовить к следующему, мы предоставляем простой сэмпл на OpenGL 3 и подробно разбираем его реализацию. Также мы предоставляем вам некоторую инфраструктуру для более простого взаимодействия с GL3 и поиска возникающих ошибок.

Создание окна, инициализация констекста, ввод-вывод.

Для того чтобы облегчить вам реализацию данного задания на GPU и подготовить к следующему, мы предоставляем весь код по инициализации контекста, создания окна и поддержке ввода-вывода, загрузки и компиляции шейдеров при помощи библиотеки glus, написанной Norbert Nopper-ом (http://nopper.tv/).

Библиотека glHelper

Основные функции, предоставляемый данной библиотекой это:

- Обработка ошибок OpenGL (все что связано с OGL_CHECK_FOR_ERRORS).
- Работа с шейдерными переменными – функции setUniform, setUniformArray.
- Отрисовка полноэкранного прямоугольника – класс FullScreenQuad.

Как отлаживать GL3

Макрос OGL_CHECK_FOR_ERRORS при обнаружении ошибки выбрасывает исключение std::runtime_error, которое содержит текстовое сообщение с кодом ошибки + строчка и файл, где данная ошибка была зафиксирована. Нужно посмотреть, в каком месте произошла ошибка и какая функция OpenGL ее вызвала. Затем нужно найти эту функцию в документации, где у каждой функции содержится описание того, какие ошибки она может вызывать.

В чем отличие GL3 от GL1 и почему мы избегаем GL2.

Самое основное отличие OpenGL 3 от OpenGL 1 заключается в том, что OpenGL3 – это не библиотека, предназначенная для рисования чего-то конкретного. Правильнее будет сказать, что OpenGL 3 – это библиотека, дающая вам доступ к аппаратным ресурсам графического процессора (GPU) и к его функциональности. Что вы делаете с этими ресурсами – абсолютно ваше дело.

OpenGL2 – это, с другой стороны нечто промежуточное. В нем вы имеете доступ к фиксированной функциональности OpenGL 1, но при этом есть доступ и к программируемой функциональности GL 3 и более поздних версий, доступной через расширения. Вам может показаться, что в таком случае OpenGL 2 – это идеальный вариант, однако это не так. OpenGL2 собрал в себе столько всякой разной функциональности и такое огромное количество состояний (ведь OpenGL – это конечный), различных расширений и багов, что программировать на нем что-то серьезное стало совершенно невозможно. По этой причине, мы не изучаем на нашем курсе OpenGL 2.

В принципе, что касается конкретно этого задания – практически все равно, на чем его делать, на GL2 или GL3. Потому что практически весь код будет в пиксельном шейдере. Так что если у вас нет видеокарты поддерживающей OpenGL 3.0, вы можете сделать это задание на OpenGL 2.0 и вашей любимой видеокарте от интел. Но в этом случае сэмпл отрисовки полноэкранного прямоугольника вам придется написать самим.

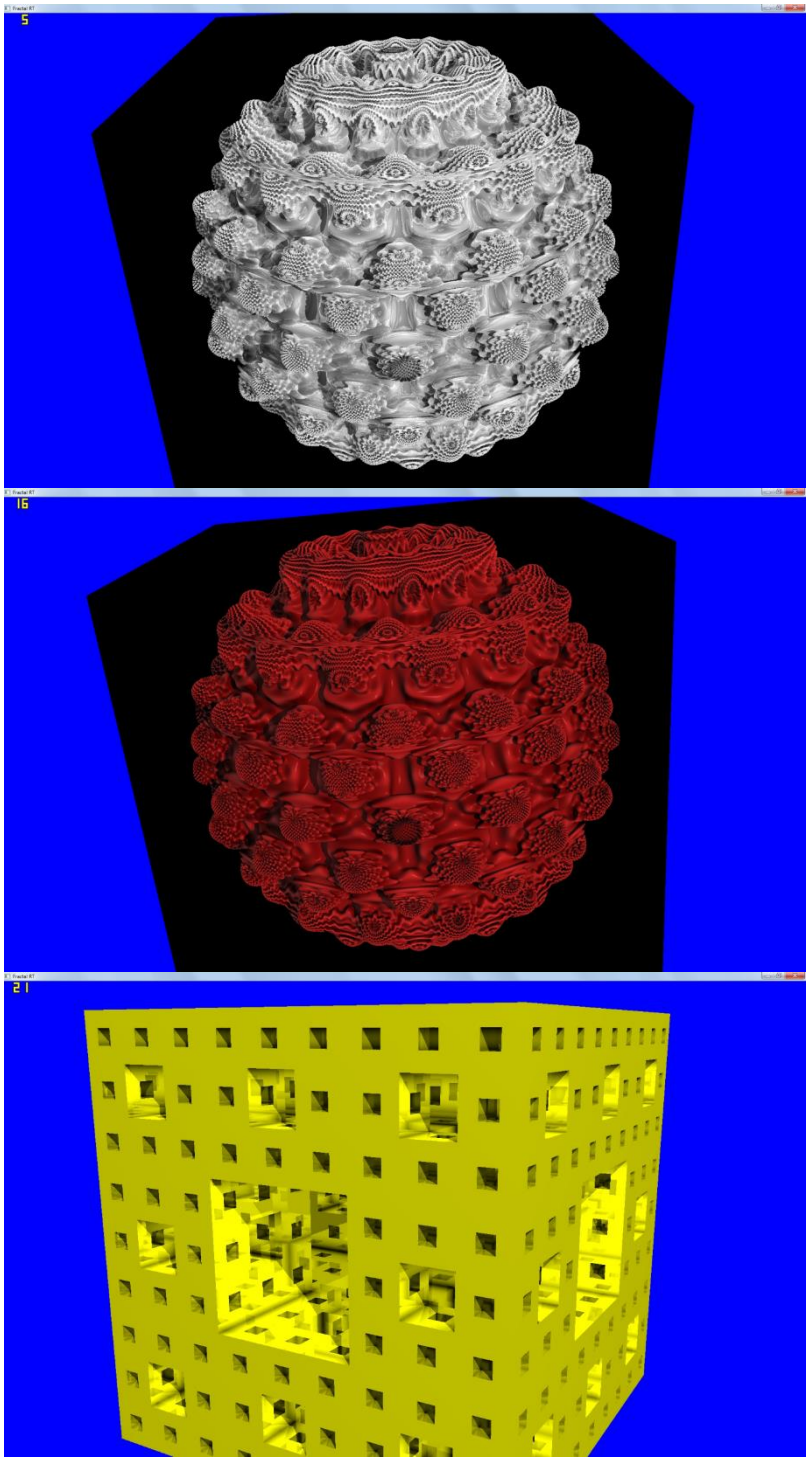
Несколько базовых понятий GL3

Начните изучение GL3 с этого урока http://code.google.com/p/gl33lessons/wiki/Lesson02 Стоит отметить, что вам не нужно беспокоиться о получении указателей на функции OpenGL с помощью wglGetProcAddress, как это по приведенной выше ссылке, в предоставленном вам примере за вас это сделает библиотека glew.

В предоставленном вам примере рисуется полноэкранный прямоугольник. В каждом пикселе экрана выполняется фрагментный шейдер, который шагает вдоль луча внутри кубика с константным шагом (не делайте так в задании, это просто пример!) и аккумулирует желтый цвет.

В теории, для выполнения данного задания, вы можете даже не запускать Visual Studio, а просто написать свой фрагментный шейдер! Но в этом случае в архив все-равно положите исходники предоставляемого вам сэмпла.

Пример выполненного задания:



Данное задание получило бы 15 за базу, 1 за резкие тени, 1 за отражения, 4 за Ambient Occlusion, 4 за 2 различных типа фрактала, 2 балла за использование CSG при построении фрактала с кубом, 4 за скорость визуализации и 1 балл за возможность вращать сцену мышкой (32 балла в сумме).

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

- [RTFGUP] http://www.raytracegroundup.com/
- [PBRT] http://www.pbrt.org/
- http://ray-tracing.ru
- http://code.google.com/p/gl33lessons/