Эффективная и практичная реализация фотонных карт на GPU

Владимир Фролов[1,2], Александр Харламов[2], Владимир Галактионов[1], Константин Востряков[2]
(1) Институт Прикладной Математики им. Келдыша Российской Академии Наук; (2) Nvidia;

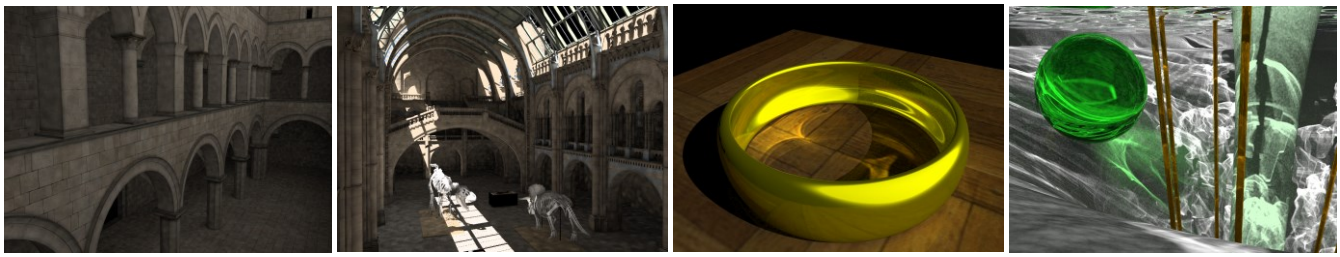

Рисунок 1. Изображения, полученные после 10 проходов (и соответствующие 10 миллионам фотонов) алгоритма прогрессивных фотонных карт. Время рендеринга – от 4 до 8 секунд в разрешении 1024x768 на GTX680.

Abstract

This paper introduce an effective and simple photon mapping implementation for GPU via constructing Multiple Reference (MR) octrees. Although MR-octrees are hierarchical structures, we successfully ignore their hierarchical nature and present an approach with plain construction, compact data layout and stack-less traversal.

Keywords: *Multiple Reference Octree, GPU, Photon Mapping*

Аннотация

В данной статье предлагается эффективная и исключительно простая реализация алгоритма фотонных карт на GPU на основе окто-деревьев с множественными ссылками. Предлагаемый алгоритм построения дерева состоит из 4 шагов, использует линейные структуры данных и только 2 параллельных примитива для построения дерева. Результирующая структура позволяет производить эффективный поиск ближайших фотонов в окто-дереве с заданным радиусом сбора без использования стэка.

Keywords: *Фотонные карты, множественные ссылки.*

1. ВВЕДЕНИЕ

Окто-дерево является достаточно популярной ускоряющей структурой в силу своей регулярности и относительно небольшой глубины. Классическое окто-дерево рекурсивно разбивает трехмерное пространство на 8 частей, сохраняя в узлах (как на рис. 1, слева) ссылки на объекты, которые целиком находятся внутри узла.

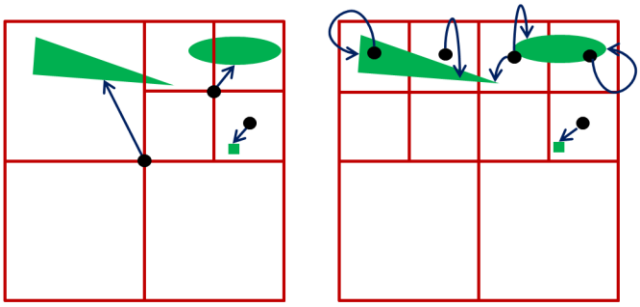

Рисунок 2: Классическое окто-дерево (слева) и окто-дерево с множественными ссылками (справа).

Окто-дерево с множественными ссылками представляет из себя модификацию классического варианта, в котором все ссылки хранятся только в листьях и на каждый объект ссылаются все листья, имеющие с данным объектом геометрическое пересечение (рис. 1, справа). Этот факт позволяет реализовать поиск в дереве без использования рекурсии или стэка, поскольку для поиска ближайших объектов в заданной точке трехмерного пространства достаточно спуститься от корня дерева к листу; каждый лист уже содержит список объектов, которые нужно перебирать последовательно.

2. ОБЗОР СУЩЕСТВУЮЩИХ МЕТОДОВ

2.1 Окто-дерево с множественными ссылками

Традиционный подход к построению окто-деревьев с множественными ссылками наиболее часто используется в технике кэширования освещенности [1]. Алгоритм позволяет добавлять новые элементы в дерево, постепенно достраивая его. При добавлении новые элементы спускаются вниз по дереву во все узлы, с которыми они имеют геометрическое пересечение.

2.2 Построение окто-деревьев на GPU

Алгоритмы построения окто-деревьев на GPU используют факт взаимосвязанности окто-дерева и трехмерной Z-кривой, которая получается в результате вычисления Z-индекса (или кода Мортона) [2] . Зафиксируем некоторую глубину дерева ‘k’. Пусть у нас имеется такой набор точек в трехмерном пространстве, что в каждый узел построенного в будущем окто-дерева попадет не более чем 1 точка. Если вычислить для точек коды Мортона и затем отсортировать точки по этим кодам, мы получим массив, эквивалентный окто-дереву, где в каждом листе хранится ровно по одному элементу (одной точке). Выполняя далее поиск в трехмерном пространстве в произвольной точке (x,y,z), мы вычисляем ее код Мортона и, применив бинарный поиск для полученного кода Мортона (ZInxed(x,y,z)), находим листовой узел окто-дерева, в который попала точка (x,y,z).

В работе [3] было впервые замечено соответствие между Z-кривой и окто-деревом и использовано свойство отбрасывания младших битов Z-индекса для получения идентификаторов старших узлов. Окто-дерево в [3]

представлялось набором массивов $L_1..L_k$, размером 8^i элементов для каждого уровня глубиной i (т.е. несуществующие узлы хранились в памяти на регулярной сетке). Вопрос хранения каких-либо элементов данных в окто-дереве в работе [3] не рассматривается. Таким образом, построенное окто-дерево не может быть использовано непосредственно для пространственного поиска элементов, а может только отвечать на вопрос имеются ли искомые элементы в некотором объеме или нет.

В работе [4] окто-дерево было использовано для построения поверхности из набора точек при помощи алгоритма марширующих кубов. Алгоритм построения использовал серию из параллельных операций уплотнения и сортировки. В отличие от работы [3] Zhou и др. сохраняют только существенные узлы в памяти, а также сохраняют списки точек на всех уровнях дерева. При этом в [4] было замечено, что имея отсортированный массив точек по их Z-индексу в узле, достаточно сохранять лишь 2 смещения, обозначающих начало и конец последовательности точек для данного узла в отсортированном массиве. Причем при переходе от уровня k на уровень k-1 это свойство сохраняется и точки пересортировывать не нужно. Достаточно пересчитать указатели, соответствующие началу и концу последовательности.

Если в рассмотренных ранее работах только отдельные уровни дерева строились параллельно, то в работе [5] был предложен метод, использующий массивный параллелизм GPU при построении всего дерева. Основная идея алгоритма заключается в том, чтобы пронумеровать все узлы дерева при помощи индексов таким образом, чтобы индекс родительского узла всегда являлся префиксом по отношению к индексам его дочерних узлов, если рассматривать индексы как битовые строки. Тогда, отсортированный массив индексов будет эквивалентен дереву. При этом элементы ненулевого размера (треугольники в работе [5]) были использованы только при построении BVH деревьев, поскольку только в этом случае размер элемента не имеет значения, а ограничивающий бокс примитива используется для расчета ограничивающего бокса листа уже после построения дерева и в самом построении участия не принимает. Это возможно благодаря тому, что в BVH деревьях различные узлы дерева могут пересекаться друг с другом своими ограничивающими объемами.

В предложенных ранее подходах построения окто-дерева в качестве примитивов использовались точки – элементы, не имеющие размера. Так как каждая точка при этом условии попадает не более чем в 1 лист, это позволяло использовать тот факт, что искомые элементы данных всегда лежат последовательно в отсортированном по их Z-индексам массиве [4]. Однако, в случае элементов, имеющих размер, это условие нарушается, поскольку один и тот же элемент может попасть в несколько узлов. Для решения этой проблемы мы используем окто-деревья с множественными ссылками и предложенный нами алгоритм построения таких деревьев на GPU.

2.3 Фотонные карты на GPU

Фотонные карты на GPU – довольно популярная тема для исследований. Одна из основных трудностей при реализации фотонных карт – эффективное построение ускоряющих структур для последующего поиска ближайших фотонов. Нам необходим метод, который бы позволял быстро находить ближайшие фотоны в заданном радиусе вокруг заданной точки трехмерного пространства. В силу ограниченного места мы отметим лишь наиболее универсальные и практичные подходы.

В [6] был предложен эффективный алгоритм построения kd-деревьев для поиска k-ближайших фотонов; [7] использовал BVH-дерево и фиксированный радиус сбора. Основным недостатком подходов, использующих классические деревья, является рекурсивный алгоритм обхода дерева при сборе освещенности. Такой алгоритм неэффективен в основном по причине высокой степени дивергентности потоков на GPU и использования локальной памяти.

В работе [8] предложен метод пространственных хэш-таблиц на основе “хэширования кукушки”. Авторы не исследовали метод в применении к алгоритму фотонных карт, и в работе отмечается, что для различных применений могут потребоваться различные хэш-функции.

В работе [9] использовали хэш-таблицы для реализации фотонных карт, причем, для того чтобы исключить коллизии во время поиска, дополнительно производили сортировку пар (ключ, значение) по ключу. При этом поиск ближайших фотонов в точке (x,y,z) производился в ячейке hash(x,y,z) и также в 27 соседних ячейках. [9] отмечает, что для глобальной фотонной карты по сравнению с kd-деревом из работы [6] этот метод не дает ускорения сбора с ростом искомого числа ближайших фотонов (т.е. с ростом радиуса сбора). Это может быть объяснено потерей производительности на ветвлениях из-за наличия вложенного цикла, что необходимо для поиска в 27 соседних ячейках. Мы покажем, что в нашем подходе такой потери нет, поскольку сбор освещенности со всех фотонов состоит целиком из 1 цикла.

В работе [10] представлен эффективный подход на основе лексикографической сортировки координат фотонов. Над отсортированным массивом строилась воксельная ускоряющая структура на основе 3D текстур. Недостатком этого подхода является высокий расход памяти.

В работе [11] предложен упрощенный подход на основе пространственных хэш-таблиц. При возникновении коллизии авторы предлагают стохастически сохранять только 1 фотон на элемент таблицы. Данный подход замедляет трассировку фотонов (в 3-4 раза в соответствии со средним числом коллизий), что может быть нецелесообразно, т.к. при сложных условиях освещения трассировка фотонов будет занимать много больше времени, чем построение ускоряющих структур при помощи любого из известных методов.

В работе [12] впервые высказана идея построения ускоряющей структуры на GPU при помощи сортировки индексов ссылок (а не индексов центров объектов). Размер ячейки в [12] был в 2 раза больше размера самого большого объекта, что гарантирует не более 8 ссылок на объект. Такое ограничение размера ячейки может негативно сказаться на скорости поиска при разных размерах объектов.

Поскольку операция сбора освещенности для 1 порции фотонов может выполняться множество раз, необходимо ускорять сбор освещенности настолько быстро, насколько это возможно. Однако, при этом метод можно считать практичным, если время построение ускоряющей структуры

меньше времени трассировки фотонов и не замедляет ее саму.

3. ПРЕДЛОЖЕННЫЙ ПОДХОД

Мы позиционируем наш метод, как более эффективный по скорости сбора освещенности на GPU, чем методы [9], [10] и [11] и при этом не замедляющий трассировку фотонов. Чтобы избежать рекурсивного обхода и собрать ближайшие фотоны в заданном радиусе, мы используем окто-дерево с множественными ссылками. Принимая во внимании то, что радиус сбора фиксирован, мы можем рассматривать фотоны как сферы с радиусом, равным радиусу сбора освещенности. И строить окто-дерево, содержащее в качестве объектов не точки, а сферы. Этот известный факт используется при построении окто-деревьев с множественными ссылками для кэша освещенности [1].

3.1 Построение окто-дерева

В начале работы наш алгоритм выбирает некоторый фиксированный уровень дерева k, для которого определяется размер его листьев. Уровень k выбирается так, чтобы размер листового узла был меньше или равен среднему размеру объектов. При этом все трехмерное пространство может быть разбито на воксели в виде сетки. Каждый воксел – лист будущего окто-дерева. Однако не все листья будут существовать в действительности, а только те, для которых будут иметься ссылки на объекты. Используя тот факт, что в окто-дереве с множественными ссылками каждый лист ссылается на все объекты, которые имеют с ним пересечение, мы можем сразу сгенерировать массив всех ссылок, не строя пока что само дерево, а вычислив для каждого объекта множество ссылок на него (рис.3).

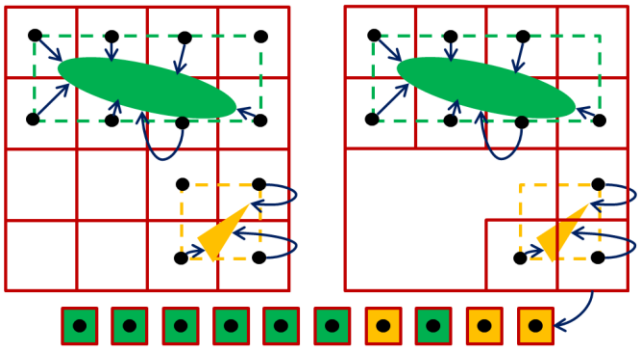

Рисунок 3: Регулярная сетка (слева); существующие ссылки-вокселы (справа); массив сгенерированных ссылок (внизу).

Оказывается, что если отсортировать массив всех ссылок (всех ссылок для всех объектов) по Z-индексу узлов, на которые они указывают, мы уже получим практически готовое окто-дерево. Ниже представлен предлагаемый алгоритм:

```
refs := AppendRefs(Voxelize(objects)); -- 1
refs.sort(compareByZIndex);            -- 2
nodes := AppendUniqueRefs(refs);       -- 3
nodes.sort(compareByZIndex);           -- 4
```

Шаг номер 1 – генерация множества всех ссылок, описанная ранее. Все ссылки добавляются в общий буфер при помощи параллельной операции append. Такая операция может быть реализована через атомарные операции в CUDA или, что более эффективно, через функциональность графических API ‘Transform Feedback’ (OpenGL) и ‘Stream Out’ (DirectX).

Шаг номер 3 представляет из себя выделение множества узлов из множества ссылок и является прямым аналогом шага 4 “remove the duplicates using parallel compaction” из работы [5]. Мы заменили операцию уплотнения (compaction) на последовательность из операций (append, sort), чтобы не выделять дополнительный массив, размер которого равен количеству ссылок. Это позволяет экономить память.

Ниже представлены типы данных, используемые алгоритмом.

```
type Reference is record
  zindex    : integer; -- Morton code
  dataIndex : integer; -- offset to user data
end record;
type Node is record
  zindex    : integer; -- Morton code
  refStart  : integer; -- first ref
  refEnd    : integer; -- last ref
end record;
```

Наша реализация использует способ хранения данных SOA (Structure Of Arrays), благодаря чему мы имеем возможность освобождать неиспользуемые массивы после построения дерева. Так мы поступили с массивом для хранения Z-индексов ссылок и в результирующей структуре ссылка представлена, таким образом, одним 32 битовым числом.

3.2 Сбор фотонов

Алгоритм сбора ближайших фотонов не использует стек и является прямым аналогом выборки из кэша освещенности на основе окто-деревьев с множественными ссылками [1]. Его основное преимущество заключается в том, что после спуска от корня к листу дерева искомые фотоны итерируются единственным циклом, перебирающим список всех ссылок на фотоны.

4. РЕЗУЛЬТАТЫ

В зависимости от радиуса сбора (и как следствие различного числа попадающих в сферу фотонов) наш алгоритм (назовем его “фотонные карты на основе окто-деревьев со множественными ссылками”) ускоряет сбор освещенности в 2-5 раз по сравнению с классическим окто-деревом (рис. 4, таблица 1). В отличие от [9] наш алгоритм продолжает значительно выигрывать у рекурсивного решения при увеличении числа собираемых фотонов. Поскольку в работе [11] не было представлено анализа скорости сбора освещенности, мы проводим аналогию с работой [9] и заключаем, что наш подход более эффективен на GPU из-за наличия только одного цикла итерирования фотонов в отличие от [9], [10] и [11], где необходим еще 1 вложенный цикл.

Мы сравнивали время построения нашего дерева с обычным окто-деревом из работы [5]. Так как известно, что алгоритм [5] ограничен производительностью сортировки, за наилучшее время построения обычного окто-дерева по методу [5] мы взяли время сортировки 1 миллиона точек.

Полученное время построения ожидаемо, так как в рассматриваемом нами типе окто-дерева на 1 фотон может сгенерироваться от 10 до 20 ссылок; поэтому время построения окто-дерева с множественными ссылками в 10-20 раз больше времени построения обычного окто-дерева на GPU на основе подхода [5]. Однако мы считаем, что данное замедление не имеет эффекта на конечное время рендеринга, поскольку дерево строится 1 раз после этапа трассировки фотонов. Причем трассировка фотонов, даже с учетом оптимизированной реализации трассировки лучей, сравнимой по производительности с [13], занимает в 5-10 раз больше времени, чем построение дерева (таблица 2, колонка “ph-trace”).

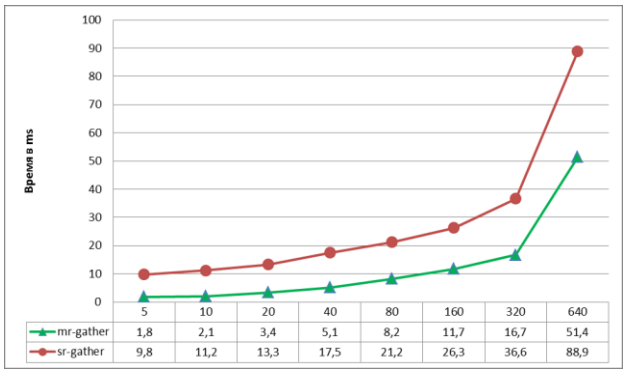

Рисунок 4. Зависимость времени сбора в миллисекундах (ось ‘y’) для 1 миллиона первичных лучей на сцене “Спонза” от среднего числе фотонов участвующих в сборе (ось ‘x’). Увеличение этого числа эквивалентно увеличению радиуса сбора или суммарному количеству обрабатываемых фотонов.

Сцена	mr-build	sr-build	mr-gather	sr-gather
Спонза	46 мс	3.1 мс	3.1 мс	14 мс
Музей	52 мс	3.2 мс	4.4 мс	16 мс
Кольцо	24 мс	3.0 мс	5.6 мс	9.9 мс
Вода	26 мс	3.0 мс	8.3 мс	18 мс

Таблица 1. Сравнение скорости построения окто-дерева с множественными ссылками (mr-build) со скоростью построения обычного окто-дерева (sr-build). Соответствующее время сбора для 1 миллиона первичных лучей по нашему методу (mr-gather) и с использованием обычного окто-дерева (sr-gather).

Сцена	ссылки	память	ph-trace	ph-trace (opt)
Спонза	10M	115MB	155 мс	22 мс
Музей	11M	128MB	470 мс	26 мс
Кольцо	2.0M	23MB	174 мс	27 мс
Вода	2.4M	25MB	300 мс	16 мс

Таблица 2. Время трассировки фотонов. Количество ссылок в окто-дереве (ссылки), занимаемая алгоритмом построения память (память), реальное время трассировки фотонов (ph-trace) и гипотетически оптимальное время трассировки фотонов (ph-trace (opt)).

Последняя колонка в таблице 2 получена как реальное время трассировки фотонов “ph-trace”, умноженное на отношение числа аккумулированных фотонов к числу фотонов, выпущенных из источника света. Это позволяет нам оценить время трассировки фотонов при применении такой гипотетической оптимизации, при которой все фотоны испускаются из источника света в правильных направлениях и всегда сохраняются в фотонной карте. Таким образом, мы можем отметить, что наше время построения окто-дерева сравнимо с недостижимым на практике теоретическим минимумом времени трассировки фотонов (таблица 2, колонка “ph-trace (opt)”), и поэтому мы делаем вывод о том, что для фотонных карт наш алгоритм имеет достаточную производительность построения дерева. Все замеры производились в разрешении 1024x768 на GTX680. Программная реализация выполнена с использованием технологии CUDA и библиотеки thrust.

5. ЛИТЕРАТУРА

[1] Křivánek, J., Gautron, P., Ward, G., Jensen, H. W., Christensen, P. H., and Tabellion, E. 2008. *Practical global illumination with irradiance caching*. In ACM SIGGRAPH 2008 Classes (Los Angeles, California, August 11 - 15, 2008). SIGGRAPH '08. ACM, New York, NY, 1-20.

[2] Morton, G. M. (1966), *A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing*, Technical Report, Ottawa, Canada: IBM Ltd.

[3] Prekshu A., Rhushabh G., Sharat C., Srinivas A. Fast, Parallel, GPU Based space filling curves and octrees. 2008. Department of Computer Science & Engineering, IIT Bombay.

[4] Zhou, K., Gong, M., Huang, X., and Guo, B. 2010. *Data-parallel octrees for surface reconstruction.* IEEE Transactions on Visualization and Computer Graphics.

[5] Tero Karras. *Maximizing parallelism in the construction of BVHs, octrees, and k-d trees.* In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics* (EGGH-HPG'12), Eurographics Association, 2012, Aire-la-Ville, Switzerland, Switzerland, 33-37.

[6] Kun Zhou, Qiming Hou, Rui Wang, Baining Guo. Real-Time KD-Tree Construction on Graphics Hardware. *ACM Transactions on Graphics (SIGGRAPH Asia 2008)*

[7] Fabianowski D., Dingliana J. *Interactive Global Photon Mapping* In Computer Graphics Forum 28(4), [EGSR 2009], pages 1151-1159, July 2009.

[8] Alcantara, D. A., Sharf, A., Abbasinejad, F., Sengupta, S., Mitzenmacher, M., Owens, J. D., Amenta, N. 2009. Realtime parallel hashing on the gpu. In SIGGRAPH Asia ’09: ACM SIGGRAPH Asia 2009 papers, ACM, New York, NY, USA, 1–9.

[9] Fleisz M. Photon Mapping on the GPU. Master of Science Computer Science School of Informatics University of Edinburgh 2009.

[10] Д.К. Боголепов., В.Е. Турлапов. Моделирование каустик в реальном времени на основе комбинированных возможностей OpenCL и шейдеров. Вестник Нижегородского университета им. Н.И. Лобачевского, 2011, No 3 (2), с. 180–186

[11] Hachisuka T., and Jensen H. W. 2010. *Parallel progressive photon mapping on GPUs.* In ACM SIGGRAPH ASIA 2010 Sketches (SA '10). ACM, New York, NY, USA, Article 54, 1 pages.

[12] Scott Le Grand. “Broad-Phase Collision Detection with CUDA”. In *GPU Gems 3*, Addison-Wesley, 2008, pp. 697–721.

[13] Aila, T. and Laine, S. 2009. *Understanding the efficiency of ray traversal on GPUs*. In Proceedings of the Conference on High Performance Graphics 2009 (New Orleans, Louisiana, August 01 - 03, 2009). S. N.

6. БЛАГОДАРНОСТИ

Работа поддержана грантами РФФИ № 12-01-31027 мол_а, № 12-01-00560 и стипендией президента РФ № СП-4053.2013.5.