ПОСТРОЕНИЕ SAH BVH ДЕРЕВЬЕВ ДЛЯ ТРАССИРОВКИ ЛУЧЕЙ НА GPU В РЕАЛЬНОМ ВРЕМЕНИ

Д.П. Сопин[1], Д.К. Боголепов[1], Д.Я. Ульянов[2], В.Е. Турлапов[1]

[1]Нижегородский госуниверситет им. Н.И. Лобачевского

[2]Нижегородский государственный технический университет им. Р.Е. Алексеева

Введение

Первые реализации трассировки лучей на ГПУ [1-3] допускали обработку сцен только со *статической* геометрией. Данное ограничение позволяло строить ускоряющую структуру на этапе предварительной обработки сцены и использовать ее на ГПУ для ускорения визуализации. Обширные исследования показали, что на стандартном потребительском оборудовании трассировка лучей в *реальном времени* возможна для всех основных ускоряющих структур: регулярные и иерархические сетки, kd-деревья и деревья ограничивающих объемов (BVH). Хотя технология нашла применение во многих задачах, она остается малопригодной в приложениях с *динамической* геометрией таких, как компьютерные игры, тренажеры и системы виртуальной реальности.

Значительное внимание современных исследований связано не только с методами трассировки, но и с алгоритмами для быстрого построения эффектных ускоряющих структур, что обеспечило бы поддержку динамической геометрии. При этом в корне меняется постановка задачи, поскольку теперь необходимо принимать во внимание не только эффективность структуры данных на этапе *визуализации*, но и учитывать время ее *построения*, которое часто является основным ограничением скорости работы. Недавние исследования показали, что поддержка динамических сцен возможна при использовании всех основных ускоряющих структур [4]. Однако в ряде случаев на анимацию наложены ограничения, в частности, допускаются только *иерархические* движения примитивов или *деформируемые* сцены.

В настоящей работе для ускорения визуализации были выбраны деревья BVH, обладающие рядом положительных особенностей. Во-первых, данная структура обеспечивает наиболее «плотную» аппроксимацию геометрии сцены при минимальном числе узлов – требуется минимальное число шагов построения и обхода дерева. Во-вторых, в процессе построения дерева используются только *центроиды* треугольников, поэтому исключается случай пересечения треугольником плоскости разбиения – примитив всегда принадлежит ровно одному потомку. Другое полезное свойство деревьев BVH состоит в возможности их *обновления* вместо полного *перестроения* и используется в ряде работ, где анимация возможна лишь с некоторыми ограничениями. Тем не менее, обработка сцен с анимацией общего вида возможна только при наличии быстрых алгоритмов построения BVH с нуля.

Высокая производительность трассировки лучей напрямую зависит от *качества* дерева, наилучшим критерием которого является *эвристика площадей поверхности* (SAH). Данная эвристика была впервые предложена в работе [5] и определяется следующим образом: на каждом шаге рекурсивного построения дерева при разбиении набора треугольников T на две части L и R вычисляется стоимость такого разбиения:

$$SAH(T \to (L,R)) = K_T + K_I \left[\frac{SA(L)}{SA(T)} N_L + \frac{SA(R)}{SA(T)} N_R \right]. \quad (1)$$

Здесь $SA(X)$ – площадь ограничивающего параллелепипеда узла X, N_X – число треугольников в X, а K_I и K_T – параметры реализации, определяющие стоимость теста пересечения и обхода дерева. Эффективность визуализации достигается путем минимизации стоимости разбиений при построении структуры данных.

Первая попытка *интерактивного* формирования SAH BVH деревьев была предпринята в работе [6], в которой автор адаптировал дискретный подход к вычислению SAH (*binned* SAH), предложенный изначально для kd-деревьев, и эффективно реализовал ее на многоядерных ЦПУ. Впоследствии данная реализация была доработана для архитектуры Intel MIC, где были получены лучшие на данный момент временные оценки [8]. Недавняя работа [7] показала, что построение SAH BVH дерева на ГПУ также возможно, однако указанные временные оценки оказываются хуже аналогичных оценок даже для устаревших ЦПУ [6], несмотря на использование более производительной аппаратуры. Таким образом, на данный момент отсутствуют эффективные реализации построения SAH BVH деревьев на ГПУ, которые обеспечили бы возможность визуализации произвольных динамических сцен.

1. Постановка задачи

Целью работы является разработка эффективных алгоритмов построения SAH BVH деревьев на ГПУ в *реальном* времени. В работе предполагается, что все объекты сцены представлены набором треугольников (*triangle soup*), при этом допускаются произвольные изменения геометрии во время визуализации.

С концептуальной точки зрения алгоритм построения дерева BVH аналогичен тем, что описаны в работах [6-8]. Однако, в отличие от указанных работ, мы не использовали простые эвристики, подобные «Median Splits» и «LBVH», поскольку они ведут к снижению качества генерируемой структуры данных и, следовательно, к худшей производительности на этапе трассировки. Предлагается ряд методик для эффектного отображения общего алгоритма на архитектуру современных ГПУ, что позволило получить 10-кратное ускорение по сравнению с лучшей известной реализацией для ГПУ [7].

2. Базовый алгоритм построения SAH BVH

Процесс построения дерева представлен в виде совокупности *заданий*, последовательное исполнение которых реализуется через концепцию *очереди*:

1) В очередь заданий добавляется корневой узел дерева, который содержит все треугольники сцены.
2) Первый узел очереди становится текущим.
3) Текущий узел разбивается на 16 равных вокселей по каждой из трех осей. Для каждого вокселя определяется число треугольников и их ограничивающий объем.
4) На основе эвристики площадей поверхности (SAH) вычисляется оптимальная плоскость разбиения из набора дискретных положений.
5) Текущий узел разбивается на два новых узла, которые содержат геометрические примитивы, расположенные слева и справа от выбранной плоскости разбиения.
6) Для каждого нового узла число треугольников сравнивается с заданным пороговым значением (мы использовали 4). Если число треугольников превышает указанный порог, то соответствующий узел добавляется в очередь заданий.
7) Текущий узел удаляется из очереди. Если очередь не пуста, то выполняется переход к шагу 2).

3. Адаптация алгоритма для ГПУ

Основная задача состояла в адаптации рассмотренного общего алгоритма для архитектуры современных графических процессоров. Очевидный способ такой адаптации состоит в отображении одного задания на одну *группу работ* (здесь и далее используется терминология стандарта OpenCL). В этом случае повторим результаты работы [7]: низкая производительность на верхних уровнях дерева, поскольку в вычислениях будет задействована только часть доступных ядер, и падение производительности на нижних уровнях дерева, связанное с ростом накладных расходов на поддержку большого числа небольших заданий.

Следующий, относительно простой подход был предложен в [8] и реализован для архитектуры Intel MIC: для больших узлов (с числом примитивов больше заданного порога) используются ресурсы всего процессора, в то время как остальные узлы обрабатываются согласно предыдущей схеме – по одному на группу работ. Данный подход оказался эффективным для архитектуры Intel MIC, но при отображении на архитектуру ГПУ появляется ряд проблем. Как и в случае с предыдущим подходом, возникает падение производительности на нижних уровнях и появляется «промежуточный слой» – часть уровней, на которых одна задача не способна загрузить весь графический процессор, но количество задач мало для того, чтобы отобразить их на группы работ.

3.1. Общая схема

Для эффективного использования ГПУ предлагается улучшенный алгоритм построения SAH BVH дерева, который в значительной степени свободен от указанных проблем. Общая схема работы состоит в следующем:

1) На каждом шаге построения дерева выполняются все доступные задания вне зависимости от их размера. В результате получаем возможность наиболее полно использовать ресурсы ГПУ, а также снизить накладные расходы, связанные с передачей заданий с центрального процессора.
2) Для обработки узлов, содержащих достаточно большое число геометрических примитивов (мы использовали порог 256), используется несколько групп работ на узел (мы использовали $N/512$, где N – число геометрических примитивов в узле). В сочетании с оптимизацией из пункта 1) такой подход обеспечивает использование ресурсов, близкое к оптимальному.
3) Построение дерева разбивается на 3 этапа: обработка больших узлов (более 32K примитивов), обработка средних узлов (от 256 до 32K примитивов) и обработка малых узлов (менее 256 примитивов). Такой подход дает возможность использовать специальные модификации алгоритма на каждом уровне, в результате чего удается сократить накладные расходы и оптимизировать использование ресурсов ГПУ.

В целом, на ЦПУ выполняется только отправка заданий и генерация новых заданий. Данные стадии обладают наименьшей трудоемкостью, поэтому их портирование на ГПУ не является оправданным.

3.2. Генерация заданий для ГПУ

Как указано ранее, для обработки одного узла используется несколько групп работ, при этом одновременно обрабатывается несколько узлов. Такой подход позволяет обеспечить высокую загрузку ГПУ, однако приводит к отсутствию простого способа определения объема работы для каждой конкретной группы работ. Поэтому в качестве дополнительных параметров для каждой группы работ мы передаем два массива величин – номер узла, обрабатываемого данной группой, и номер данной группы в узле.

3.3. Подсчет примитивов

Вторым этапом построения уровня дерева является подсчет числа треугольников в каждом вокселе узла и ограничивающих объемов (AABB) этих треугольников для каждой группы работ (это корректно, так как каждая группа обрабатывает примитивы из одного узла дерева). Вычисления выполняются для всех трех координатных осей – как показали эксперименты, на определенных сценах это дает существенный (до 10 раз) рост производительности. Этап можно условно разделить на две части: формирование данных о вокселях узла на основе информации о треугольниках и применение к этим данным классического алгоритма параллельной редукции.

Для реализации первой части был использован подход, схожий с используемыми в работах [7] и [8]: примитивы каждой группы работ делятся на части по 16 элементов (в соответствии с числом вокселей) и каждая из указанных частей обрабатывается 16 последовательными потоками (*halfwarp* в терминологии NVIDIA CUDA). Данный алгоритм можно описать следующим псевдокодом:

```
for i in 1 to 16 do
  if bin(triangle[i]) = threadId
    bin[threadId].append(triangle[i])
```

Использование 16 проходов на 16 элементов позволяет оптимально использовать SIMD архитектуру ГПУ и способствует эффективной работе с памятью (устраняет конфликты банков и обеспечивает *coalescing*).

3.4. Редукция

Для реализации данного этапа использовался классический алгоритм параллельной редукции. Однако в процессе его адаптации было сделано одно существенное изменение – не используем переменное число итераций алгоритма. При обработке «средних» уровней дерева достаточно всего одной итерации, при обработке «больших» уровней использовали дополнительное ядро «окончательной редукции», редуцирующее все имеющиеся комплекты вокселей в один (для «маленьких» уровней этот этап не требуется вовсе). Потери производительности для такого подхода не существенны, так как основная масса вычислительной нагрузки приходится на этапы подсчета и редукции. В результате получили более простой алгоритм и уменьшили объем обмена данных между ЦПУ и ГПУ.

3.5. Поиск оптимального разбиения

Это одна из наименее ресурсоемких частей алгоритма. В ней на основе имеющихся данных о вокселях с помощью SAH вычисляется оптимальная плоскость разбиения из набора дискретных положений. Этап реализован на ГПУ для уменьшения объема трафика между ЦПУ и ГПУ.

3.6. Переупорядочивание

Последним ресурсоемким этапом построения является переупорядочивание элементов узлов в соответствии с найденными разбиениями (элементы слева от плоскости разбиения перемещаются в начало массива, элементы справа – в конец). В качестве основы для решения данной подзадачи использовался алгоритм *поразрядной сортировки*, эффективно реализованный для ГПУ в работе [11]. Традиционно данный алгоритм выполняется в *несколько* проходов, что влечет за собой дополнительные расходы. В нашем алгоритме вместо «глобального» префиксного суммирования использованы *атомарные операции* на глобальной памяти. В результате дополнительные шаги вычисления префиксной суммы на глобальном уровне заменены одной атомарной на

группу работ на локальном уровне, что позволило сократить требуемую для алгоритма память и количество вычислений.

4. Результаты

В вычислительных экспериментах использовалась классическая трассировка лучей Уиттеда. Для обхода дерева BVH использовался классический алгоритм на основе полного стека (в *частной* памяти каждого потока).

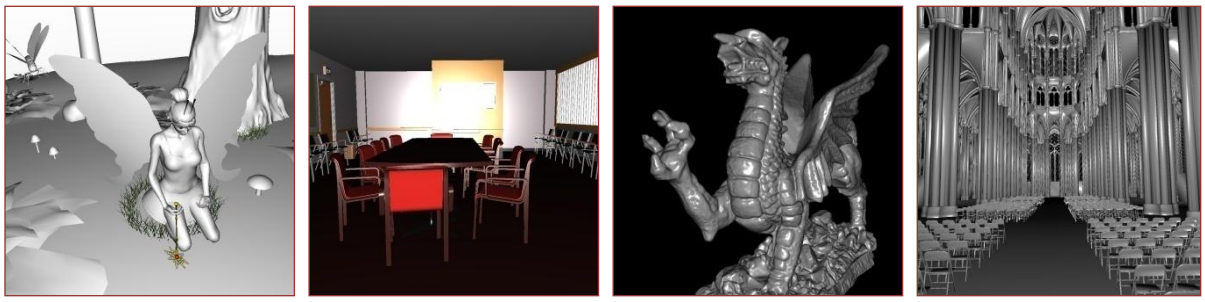

Рис. 1. Изображения тестовых сцен

Оценки производительности изложенного алгоритма выполнялась на различных сценах размера от 10K до 3M треугольников, которые часто используются исследователями для сравнения алгоритмов визуализации (рис. 2). Тесты проводились на компьютере с ГПУ NVIDIA GeForce GTX 480 под управлением операционной системы Linux (с версией видеодрайвера 270.41.06). В таблице представлено сравнение полученных результатов с другими известными работами.

Scene		*Lauterbach* [7]		*Wald* [8]		*Ours*	
Fairy Forest	174K	488 ms	21.7 FPS	31 ms	29 FPS	40 ms	25 FPS
Conference	284K	477 ms	24.5 FPS	42 ms	46 FPS	68 ms	36 FPS
Welsh-dragon	2.2M	–	–	–	–	362 ms	19.6 FPS
Cathedral	3.2M	–	–	–	–	697 ms	14 FPS

Заключение

Исследовалась задача построения SAH BVH деревьев в реальном времени с использованием ГПУ. Предложенные методики адаптации общего алгоритма позволили улучшить результаты, более чем в 10 раз по сравнению с лучшей известной реализацией [7] (с учетом разницы в использованной аппаратуре – в 5 раз). Более того, полученные временные оценки сопоставимы с оценками для «компромиссных» структур (обеспечивающих быстрое построение, но менее эффективных при трассировке – Hybrid BVH, Two-level Grids), полученными в работах [7], [13]. Текущая реализация позволяет выполнять визуализацию произвольных *динамических* сцен до 800K треугольников и *статических* сцен до 5M треугольников.

Литература

1. Timothy J. Purcell. Ray Tracing on a Stream Processor. Ph.D. dissertation, Stanford University, March 2004.
2. Foley T., Sugerman J. KD-Tree Acceleration Structures for a GPU Raytracer. In Proceedings of the ACM SIGGRAPH/ Eurographics conf. on Graphics hardware, P. 15–22, 2005.

3. J. Gunther, S. Popov, H. Seidel, P. Slusallek. Real-time Ray Tracing on GPU with BVH-based Packet Traversal. In Proc. of the IEEE Symposium on Interactive Ray Tracing, 2007.
4. Wald I., Mark W., Gunther J., et al. State of the Art in Ray Tracing Animated Scenes. Computer Graphics Forum 28(6), 2009, P. 1691–1722.
5. Goldsmith J. and Salmon J. Automatic Creation of Object Hierarchies for Ray Tracing. IEEE Computer Graphics and Applications, 1987. Vol. 7, No. 5, P. 14–20.
6. Wald I. On fast Construction of SAH-based Bounding Volume Hierarchies. In Proceedings of the Eurographics Symposium on Interactive Ray Tracing, 2007, P. 33–40.
7. Lauterbach C., et al. Fast BVH Construction on GPUs. Computer Graphics Forum, 2009. 28, 2. P. 375–384,.
8. Wald I. Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture // IEEE Transactions on Visualization and Computer Graphics, 2010.
9. Thrane N., Simonsen L. A Comparison of Acceleration Structures for GPU Assisted Ray Tracing. Master's thesis University of Aarhus. 2005.
10. Zhou K., et al. Real-time KD-tree construction on graphics hardware. ACM Trans. Graph. 27, 5, 1–11, 2008.
11. Satish N., Harris M., Garland M. Designing efficient sorting algorithms for manycore GPUs.
12. Harris M. Parallel Prefix Sum (Scan) with CUDA.
13. J. Kalojanov, et al. Two-level Grids for Ray Tracing on GPUs // Proc. of the Eurographics conf., 2011.