УДК 004.424.5.032.24

Г.В. Воронцов, А.П. Преображенский, О.Н. Чопоров

# БЫСТРОЕ ПОСТРОЕНИЕ BVH ДЕРЕВА НА GPGPU

*Воронежский институт высоких технологий*
*Воронежский государственный технический университет*

*В данной статье проводится анализ возможностей разработки алгоритма построения BVH-дерева. Этот алгоритм должен позволять быструю перестройку дерева при изменении параметров задачи. Предлагается анализировать линейное BVH-дерево, вначале выбирается порядок, в котором листовые узлы обозначаются в дереве, затем генерируются внутренние узлы по этому порядку. Рассматривается кривая Z-порядка, вычисляются Мортон-коды для каждого листового узла. На следующем шаге осуществляется сортировка получившихся Мортон-кодов. В алгоритме сортировки двоичной последовательности используется параллельный алгоритм Radix Sort. Для анализа последовательности листовых узлов, необходимо разбить ее на подинтервалы, размеры которых определяются в рамках бинарного поиска. Чтобы повысить значение занятости видеокарты, предлагается указать внутренние узлы определенным образом, и тогда, появляется возможность узнать, к какому диапазону листовых узлов относится любой выбранный внутренний узел. Для тестирования алгоритма быстрого построения BVH-дерева на GPGPU была использована технология DirectCompute и язык программирования шейдеров - HLSL. Приведена зависимость времени выполнения каждой от количества примитивов при вычислении ограничивающих объемов листовых узлов и их кодов Мортона, формировании иерархии дерева, а также вычислении ограничивающих объемов внутренних узлов.*

**Ключевые слова:** BHV-дерево, бинарный поиск, видеокарта, листовой узел, коды Мортона.

**Введение.** BVH-дерево (англ. Bounding Volume Hierarchy Tree) применяется во многих практических апектах компьютерной графики: от обнаружения физических столкновений до поиска точки пересечения в алгоритмах трассировки, будь то трассировка лучей, или трассировка пути, или более сложный алгоритм [1]. Описание трехмерной сцены с помощью бинарного дерева очень хорошо помогает оптимизировать поиск требуемых объектов на сцене. Но для использования дерева в каждом рендер-кадре для динамических объектов в сцене, необходимо не только быстро искать нужные данные, но и быстро строить или перестраивать данное дерево. В случае с постоянным перестроением дерева, может возникнуть такая ситуация, что дерево станет вырожденным, и любой поиск по нему станет неоптимальным в плане затраченного времени. В этой связи, необходимо использовать такой алгоритм, который позволит, без каких-либо серьезных затрат ресурсов, строить дерево каждый раз заново. В данной работе даются предложения по формированию такого алгоритма.

**Описание методики построения алгоритма построения дерева.** С точки зрения практики представляет интерес реализация параллельного

алгоритма построения BVH-дерева, например, на видеопроцессоре. Для этого можно использовать технология GPGPU, например, DirectCompute или CUDA. И, из всех видов BVH-дерева для такого алгоритма наиболее хорошо подойдет так называемое линейное BVH-дерево (LBVH), так как основной особенностью данного вида дерева является то, что, зная порядок листовых узлов, внутренние узлы можно рассматривать как их линейный диапазон [4]. А сама идея алгоритма состоит в том, чтобы вначале выбрать порядок, в котором листовые узлы (причем, каждый листовой узел соответствует одному примитиву, например, треугольнику) обозначаются в дереве, а затем уже генерировать внутренние узлы по этому порядку. Так как использование BVH-дерева предполагает оптимизацию поиска нужных примитивов, то объекты, которые находятся рядом друг с другом, должны и в дереве быть расположены также. Поэтому, необходимо выбрать порядок таким образом, чтобы не нарушить эту концепцию, иначе бинарный поиск по дереву будет не оптимален. Для этого, можно выбрать сортировку по так называемой кривой заполнения пространства (англ. Space-filling curve) [5]. Выберем кривую Z-порядка [6], которая также еще называется кривая Мортон-порядка, и которая представляется в виде карты многомерных данных в одном измерении, но при этом идет сохранение расположения точек данных. В нашем случае, точка данных является трехмерной точкой – центром Bounding Box.

Данные в карте кривой Z-порядка представляются в виде так называемых Мортон-кодов – двоичных последовательностей, поэтому первым шагом алгоритма построения BVH-дерева будет вычисление Мортон-кодов для каждого листового узла (который, как мы помним, хранит в себе один примитив). Для этого, нам необходимо найти центр тяжести примитива (центр Bounding Box), и для него уже вычислить код: сначала берем дробную часть каждой координаты центра (который представлен трехмерным вектором - $x, y, z$) и расширяем ее, вставляя по два нуля после каждого бита (алгоритм 2), и в итоге получаем 30-битную последовательность. Далее, мы должны чередовать биты всех трех координат, чтобы получить одну двоичную последовательность для всей точки (Рис. 1). Следующим же шагом является сортировка получившихся Мортон-кодов, чтобы получить кривую Z-порядка, так как она предполагает, что все коды должны быть в порядке возрастания.

Так как Мортон-код является двоичной последовательностью, то наилучшим выбором из алгоритмов сортировок, является поразрядная сортировка (Рис. 2). В этой связи, мы должны взять параллельный алгоритм Radix Sort, чтобы время выполнения данного шага было минимальное. После того, как отсортировали все коды, мы получаем Z-кривую, на основе которой мы можем построить BVH-дерево.

1: **for each** primitive with index $i \in [0, n-1]$ **in parallel do**
2: $b_{min} \leftarrow (0,0,0)$
3: $b_{max} \leftarrow (0,0,0)$
4: **for** $p \in [0,3]$ $where\ p\ is\ position\ vertex\ in\ primitive$ **do**
5: $b_{min} \leftarrow minUnion(b_{min}, p)$
6: $b_{max} \leftarrow maxUnion(b_{max}, p)$
7: **end for**
8: $center \leftarrow \frac{b_{max}+b_{min}}{2}$
9: $center_x \leftarrow \min(\max(center_x * 1024, 0), 1023)$
10: $center_y \leftarrow \min(\max(center_y * 1024, 0), 1023)$
11: $center_z \leftarrow \min(\max(center_z * 1024, 0), 1023)$
12: $xx \leftarrow expandBits(center_x)$
13: $yy \leftarrow expandBits(center_y)$
14: $zz \leftarrow expandBits(center_z)$
15: $mortonCode \leftarrow xx * 4 + yy * 2 + zz$
16: **end for**

Рисунок 1 – Реализация алгоритма, связанного с вычислением кода Мортона для каждого примитива

1: unsigned integer expandBits(unsigned integer $v$)
2: $v \leftarrow (v * 0x00010001)\ \&\ 0xFF0000FF$
3: $v \leftarrow (v * 0x00000101)\ \&\ 0x0F00F00F$
3: $v \leftarrow (v * 0x00000005)\ \&\ 0x49249229$
4: $result \leftarrow v$

Рисунок 2 – Реализация алгоритма, связанного с расширением числа с помощью вставки двух нулей после каждого бита

Как уже было сказано, линейное BVH-дерево удобно использовать в том случае, если знаем порядок листовых узлов. Порядок мы вычислили (Z-кривая), теперь можно рассматривать внутренние узлы. Предположим, что у нас есть $N$ листовых узлов, тогда корневой узел дерева, по логике, содержит все из них, то есть, он охватывает диапазон $[0, N-1]$. Левый дочерний узел корня охватывает диапазон $[0, \gamma]$, а правый - $[\gamma+1, N-1]$. То есть, мы можем, спускаясь сверху вниз, и разбивая на вот такие диапазоны нашу последовательность листовых узлов для каждого внутреннего узла, построить дерево. Причем, данный алгоритм выполняется до того момента, пока мы не столкнемся с таким диапазоном, который содержит только один элемент из последовательности, то есть, пока мы не придем к листовому узлу.

Но, возникает вопрос, а каким образом разбивать диапазоны? То есть, как выбрать величину $\gamma$? Исходя из концепции LBVH, разбивать диапазоны необходимо в соответствии с самым высоким битом, который отличается между кодами Мортона в заданном диапазоне [3]. То есть, необходимо разбивать диапазон так, чтобы старший бит был равен нулю

для всех элементов левого дочернего узла, и единице – для правого дочернего узла (Рис. 3).

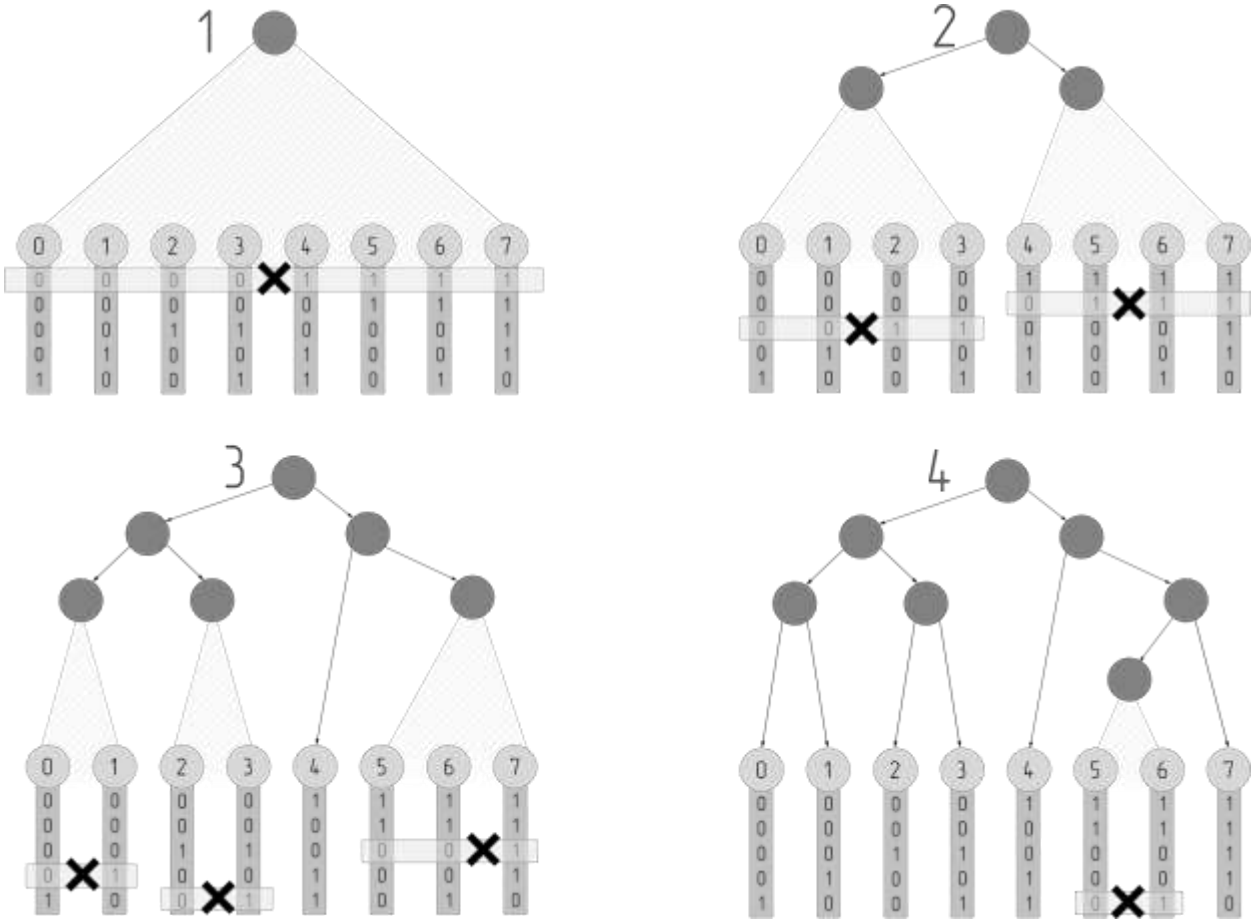


Рисунок 3 – Построение BVH-дерева

Если смотреть на данный процесс с точки зрения Z-кривой, то получается, что разбиение элементов кривой по самому высокому биту соответствует их расположению по обе стороны плоскости, которая выровнена по оси в трехмерном пространстве.

Чтобы найти самые высокие биты, которые отличаются между кодами в заданном диапазоне, логичным будет использование бинарного поиска. Если более подробно рассматривать, то нам необходимо выбрать наилучшее предположение для позиции в последовательности кода Мортона и попытаться продвинуть ее по экспоненциально убывающим шагам. При этом на каждом шаге мы должны проверить, будет ли новая позиция нарушать условие для левого потомка, если не нарушает, то принимаем ее, иначе – не принимаем. Сам цикл проверки новой позиции выполняется до тех пор, пока не закончатся шаги (Рис.4).

1: **for each** internal node with index $i \in [0, n-1]$ **in parallel do**
2: $code_{first} \leftarrow L_{first}$
3: $code_{last} \leftarrow L_{last}$
4: **if** $code_{first} < code_{last}$ **then**
5: $split \leftarrow first$
6: break
7: **end if**
8: $prefix_{common} \leftarrow \delta(code_{first}, code_{last})$
9: $split \leftarrow first$
10: $step \leftarrow last - first$
11: **while** $step > 1$ **do**
12: $step \leftarrow \frac{step+1}{2}$
13: $split_{new} \leftarrow split + step$
14: **if** $split_{new} < last$ **then**
15: $code_{split} \leftarrow L_{split_{new}}$
16: $prefix_{split} \leftarrow \delta(code_{first}, code_{split})$
17: **if** $prefix_{split} > prefix_{common}$ **then**
18: $split \leftarrow split_{new}$
19: **end if**
20: **end if**
21: **end while**
22: **end for**

Рисунок 4 – Реализация алгоритма, предназначенного для разбиения диапазонов в BVH-дереве

Как видно из рис. 3, каждый новый диапазон зависит от предыдущего. Можно, конечно, выполнять алгоритм в такой реализации, при этом создавая новые потоки для каждого из потомков, но тогда, получится, что вначале у нас будет выполняться только один поток – на корневой узел, потом два – на его левого и правого потомка, потом четыре и т.д [3]. Недостаток такого алгоритма заключается в том, что такой параметр видеокарты как ее занятость будет очень и очень низким. Для первого уровня дерева значение занятости будет составлять не более 0,006%, на втором – 0,013%, что крайне мало. То есть, мощности GPU будут не задействованы, и только на последних уровнях значение занятости достигнет приемлемого уровня. Для realtime построения BVH-дерева крайне важной является скорость выполнения алгоритма, и поэтому, рассмотрим другую возможность. Основная идея нашего предложения состоит в том, чтобы указать внутренние узлы определенным образом, и тогда, появляется возможность узнать, к какому диапазону листовых узлов относится любой выбранный внутренний узел. Такая концепция позволяет выполнить алгоритм построения иерархии дерева для каждого узла независимо от других, что позволит сразу выделить максимальное количество потоков, и полностью использовать доступные мощности видеопроцессора.

Используя тот факт, что любое двоичное дерево, а BVH дерево является именно таким, с $N$ листовыми узлами всегда имеет ровно $N-1$ внутренних узлов, мы можем сразу выделить $N-1$ потоков, и начать параллельно генерировать иерархию дерева. Основная сложность состоит в том, чтобы правильно выбрать диапазон, который принадлежит текущему внутреннему узлу. Как уже было ранее сказано, необходимо вначале каким-либо образом пронумеровать внутренние узлы. Корень будет иметь индекс 0, а дочерние элементы каждого узла расположены по обе стороны от его разделенного положения. Из-за свойств отсортированных кодов Мортона [6], данная схема нумерации никогда не приведет к дубликатам или пробелам.

Теперь остается только вычислить диапазон выбранного узла (Рис. 5), и для этого, сначала, мы должны определить направление диапазона текущего внутреннего узла, смотря на соседние коды Мортона [2]. Для примера, обозначим направление через $d$, а текущий код через $ki$ и $d=1$ будет указывать на диапазон, который начинается с $ki$, и $d=-1$ будет указывать на диапазон, который заканчивается на $ki$. Так как каждый внутренний узел содержит диапазон, состоящий минимум из двух кодов Мортона, то мы знаем, что $ki$ и $ki+d$ должны принадлежать текущему узлу. Мы также знаем, что $ki-d$ принадлежит соседнему узлу, так как потомки узла всегда расположены рядом друг с другом в иерархии дерева.

1: **for each** internal node with index $i \in [0, n-1]$ **in parallel do**
2: $d \leftarrow sign(\delta(i, i+1) - \delta(i, i-1))$
3: $\delta_{min} \leftarrow \delta(i, i-d)$
4: $l_{max} \leftarrow 2$
5: **while** $\delta(i, i + l_{max} * d) > \delta_{min}$ **do**
6: $l_{max} \leftarrow l_{max} * 2$
7: $l \leftarrow 0$
8: **for** $t \leftarrow \{l_{max}/2, l_{max}/4, \dots, 1\}$ **do**
9: **if** $\delta(i, i + (l+t) * d) > \delta_{min}$ **then**
10: $l \leftarrow l + t$
11: $j \leftarrow i + l * d$
12: $\delta_{node} \leftarrow \delta(i, j)$
13: $s \leftarrow 0$
14: **for** $t \leftarrow \{l/2, l/4, \dots, 1\}$ **do**
15: **if** $\delta(i, i + (s+t) * d) > \delta_{node}$ **then**
16: $s \leftarrow s + t$
17: $\gamma \leftarrow i + s * d + \min(d, 0)$
18: **if** $\min(i, j) = \gamma$ **then** $left \leftarrow L_\gamma$ **else** $left \leftarrow I_\gamma$
19: **if** $\max(i, j) = \gamma + 1$ **then** $right \leftarrow L_{\gamma+1}$ **else** $right \leftarrow I_{\gamma+1}$
20: $I_i \leftarrow (left, right)$
21: **end for**

Рисунок 5 – Реализация алгоритма, предназначенного для вычисления диапазона выбранного узла.

Теперь, коды, которые принадлежат текущему узлу, имеют общий префикс. Это означает, что нижняя границы длины префикса задается выражением $\delta_{min} = \delta(i, i-d)$, так, что $\delta(i,j) > \delta_{min}$ для любого $kj$, который принадлежит текущему узлу. Мы сможем выполнить данное условие, если будем сравнивать $\delta(i, i-1)$ с $\delta(i, i+1)$ и выбирая $d$ так, чтобы $\delta(i, i+d)$ соответствовало большему индексу.

Тоже самое мы делаем и для поиска другого конца диапазона, проводя поиск наибольшего $l$, который удовлетворяет условию $\delta(i, i+l_d) > \delta_{min}$. Сначала мы определяем степень верхней границы $l_{max} > l$, начиная с двойки и увеличивая значения экспоненциально, пока она больше не будет удовлетворять условию $\delta(i, i + l_{max} * d) > \delta_{min}$. Как только мы вычислили верхнюю границу, мы находим $l$, используя бинарный поиск в диапазоне $[0, l_{max} - 1]$. То есть, мы должны рассмотреть каждый бит $l$, начиная с самого высокого, и установить его в единицу, если новое значение не будет удовлетворять условию $\delta(i, i + (l+t) * d) > \delta_{min}$, а другой конец диапазона задается $j = i + l_d$.

Значение $\delta(i,j)$ показывает нам длину префикса, который соответствует текущему внутреннему узлу, которую обозначим через $\delta_{node}$. Теперь мы можем найти позицию $\gamma$, которая делит диапазон, выполнил двоичный поиск для наибольшего $s \in [0, l-1]$, и удовлетворяющего условию $\delta(i, i + s_d) > \delta_{node}$. Если направление $d = 1$, то тогда $\gamma = i + s_d$, так как это самый высокий индекс, который принадлежит левому потомку. Если направление $d = -1$, то мы должны уменьшить значение. Получается, что потомки текущего узла охватывают диапазоны $[\min(i,j), \gamma]$ и $[\gamma + 1, \max(i,j)]$, о чем, в принципе, мы говорили в начале статьи. И в итоге, мы сравниваем начало и конец диапазона каждого потомка, чтобы увидеть, является ли он листовым или внутренним узлом

После выполнения всех предыдущих алгоритмов у нас выстраивается бинарное дерево, но оно еще не является BVH-деревом, так как ограничивающие объемы имеют только листовые узлы, а по определению BVH Tree, каждый узел, внутренний или листовой, должен иметь ограничивающий объем (в нашем случае это Bounding Box). Причем, Bounding Box для каждого узла зависит от своих потомков, и распараллелить процесс вычисления объема достаточно сложно. Поэтому, можно воспользоваться подходом, который позволит параллельно, и достаточно быстро, посчитать ограничивающие объемы для каждого узла. Мы уже знаем Bounding Box для каждого листового узла, поэтому суть данного подхода заключается в том, чтобы выделить $N-1$ потоков, где $N$ – количество листов в дереве, и начать восхождение к корню дерева от

листов, рассчитывая для каждого внутреннего узла объем. Чтобы избежать дублирования вычислений, можно использовать атомарный флаг для каждого узла, который покажет, был ли для него вычислен ограничивающий объем.

**Полученные результаты.** Для тестирования алгоритма быстрого построения BVH-дерева на GPGPU была использована технология DirectCompute и язык программирования шейдеров – HLSL. Алгоритм был разделен на три части (шейдера), в которые не входит сортировка кодов Мортона:

- Вычисление ограничивающих объемов листовых узлов и их кодов Мортона
- Формирование иерархии дерева
- Вычисление ограничивающих объемов внутренних узлов

Также, были сгенерированы массивы из треугольников разной длины, чтобы можно было понять, насколько быстро строится дерево. Результаты тестирования показаны на Рис. 6.

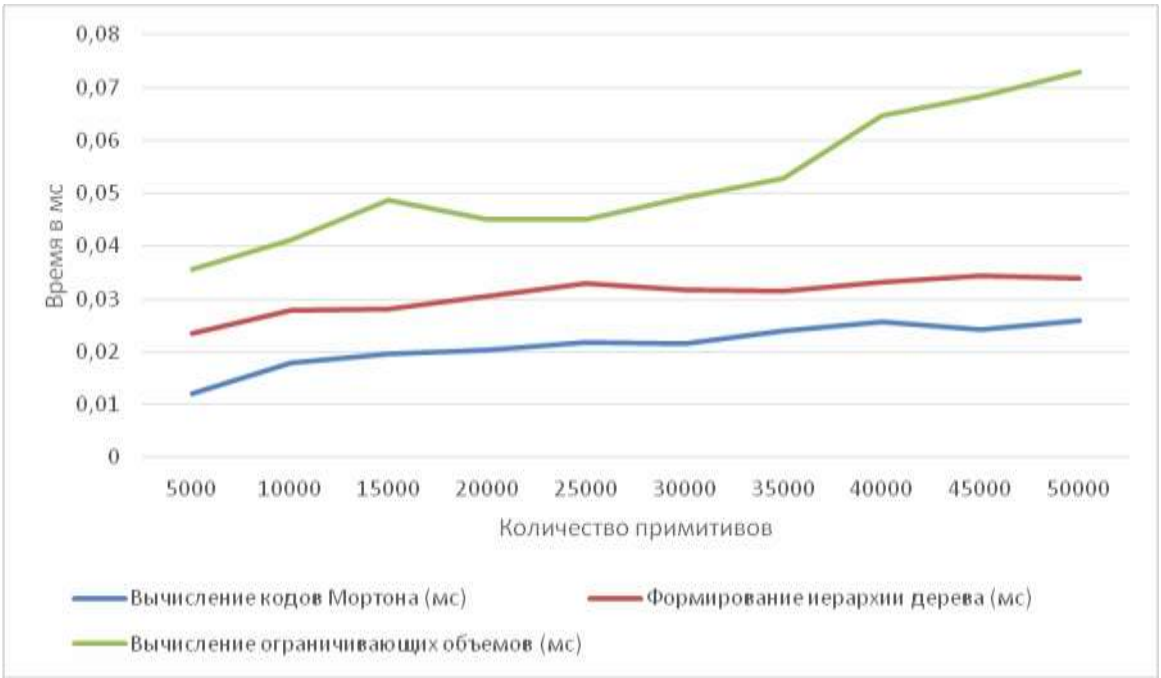


Рисунок 6 – Зависимость времени выполнения каждой из трех частей алгоритма от количества примитивов по предлагаемому алгоритму

Также сравним скорость построения SAH BVH-дерева, которое строится рекурсивно на CPU и наш алгоритм через Мортон-коды на GPU, учитывая время на сортировку (Рис.7 и Рис.8). Как видно из графика, SAH BVH-дерево строится намного дольше, чем наш алгоритм, и дело не только в том, что алгоритм выполняется на CPU, но еще и в том, что на каждом шаге выполняется минимум три сортировки, и в среднем 512

проверки (шага) для каждой из осей пространства, в алгоритме Morton BVH же сортировка выполняется только один раз.

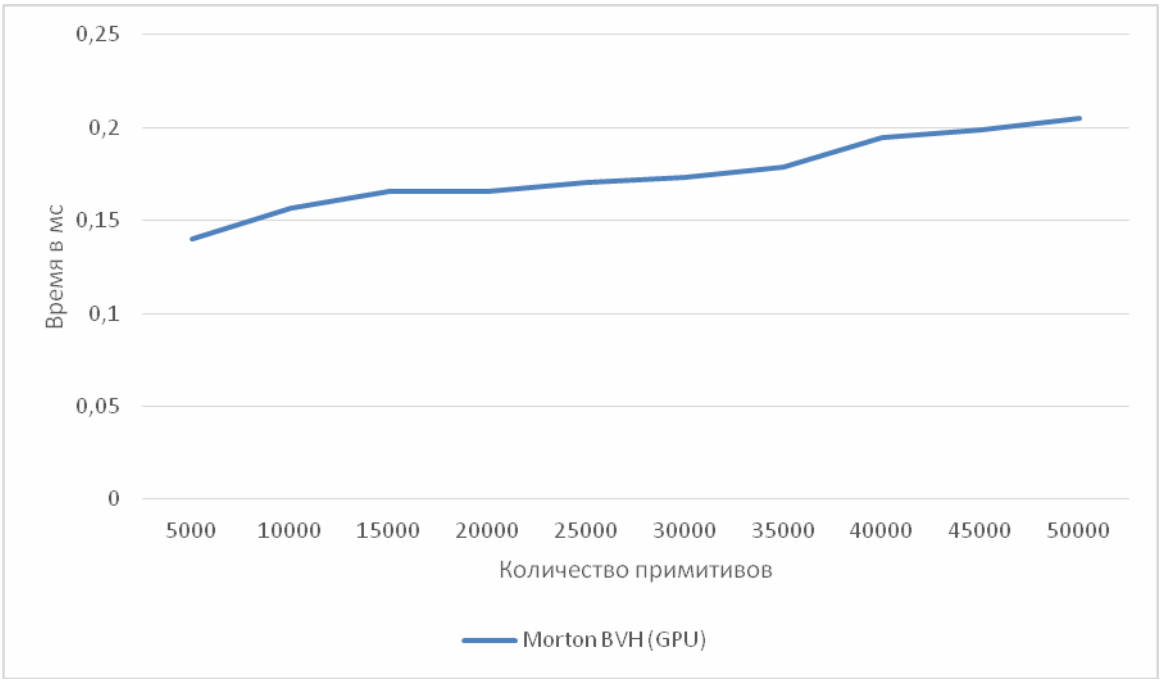


Рисунок 7 – Зависимость времени выполнения алгоритма от количества примитивов для Morton BVH (с Radix sort)

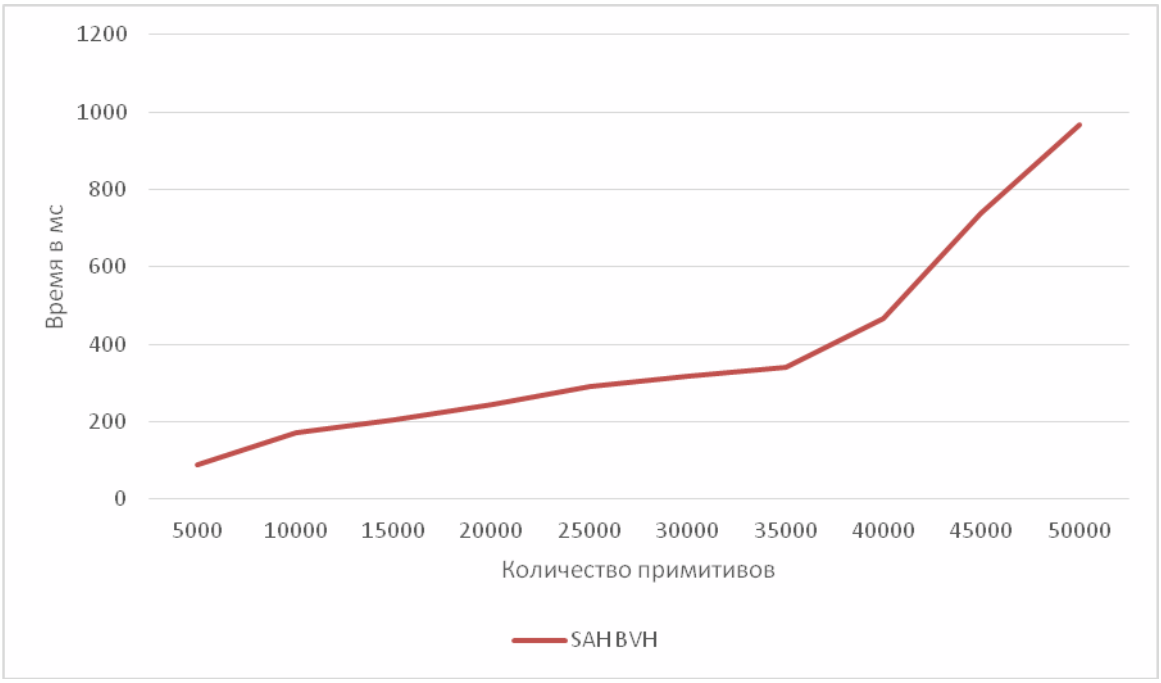


Рисунок 8 – Зависимость времени выполнения алгоритма от количества примитивов SAH BVH

Все тесты проводились на системе с видеокартой 10-го поколения Nvidia GeForce 1060 6Gb GTX и CPU Intel i7.7700.

**Выводы.** Новизна данного подхода генерации BVH-дерева состоит в том, что используется Z-кривая для получения правильного порядка примитивов в трехмерной сцене, в отличие от других алгоритмов, где на каждом шаге необходимо сортировать все примитивы по трем осям. Также, данный подход позволяет строить дерево полностью параллельно, что очень сильно уменьшает время выполнения алгоритма. Но, стоит отметить, что на практике, хоть дерево быстро строится, и выполнять алгоритм можно хоть каждый кадр, но результаты поиска по нему оказались несколько хуже, чем если бы дерево строили по SAH. Оказалось, что коды Мортона не дают оптимального разделения на узлы, из-за чего при поиске очень много времени уходит на лишние вычисление. Поэтому, данный алгоритм стоит использовать, например, для трассировки или поиска столкновений, только для динамических объектов, а для статической геометрии – лучше подойдет дерево, построенное с использованием SAH.

## ЛИТЕРАТУРА

1. Boundin Volume Hierarchy [электронный ресурс]: https://en.wikipedia.org/wiki/Bounding_volume_hierarchy
2. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees, Tero Karras, NVIDIA Research, 2012 [электронный ресурс]: https://devblogs.nvidia.com/wp-content/uploads/2012/11/karras2012hpg_paper.pdf
3. Simpler and Faster HLBVH with Work Queues, Kirill Garanzha, Jacopo Pantaleoni, David McAllister, NVIDIA [электронный ресурс]: http://www.highperformancegraphics.org/previous/www_2011/media/Papers/HPG2011_Papers_Garanzha.pdf
4. Fast BVH Construction on GPUs, C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, D. Manocha [электронный ресурс]: http://luebke.us/publications/eg09.pdf
5. Space filling curve [электронный ресурс]: https://en.wikipedia.org/wiki/Space-filling_curve
6. Z-order curve [электронный ресурс]: https://en.wikipedia.org/wiki/Z-order_curve

G. V. Vorontsov, A. P. Preobrazhenskiy, O. N. Choporov

# THE ALGORITHMS OF PARALLEL RADIX SORTING ON GPGPU

*Voronezh Institute of high technologies*
*Voronezh state technical University*

*This paper analyzes the possibilities of developing the BVH tree construction algorithm. This algorithm should allow to quickly rebuild the tree when you change the parameters of the problem. It is proposed to analyze the linear BVH tree, first select the order in which the leaf nodes are indicated in the tree, then the internal nodes are generated in this order. The z-order curve is selected and the Morton codes for each leaf node are calculated. The next step is to sort the resulting Morton codes. The binary sequence sorting algorithm uses a parallel Radix Sort algorithm. To analyze the sequence of leaf nodes, it is necessary to divide it into subintervals, the sizes of which are determined by binary search. In order to increase the occupancy of the graphics card, you are prompted to specify the internal nodes in a certain way, and then you have the opportunity to find out what range of leaf nodes any selected internal node belongs to. DirectCompute technology and Shader programming language-HLSL were used to test the algorithm of fast BVH tree construction on GPGPU. Given the dependence of running time of each of the number of primitives in the computation of the bounding volumes of the leaf nodes and their codes of Morton, forming a tree hierarchy, the computation of the bounding volumes of internal nodes.*

**Keywords:** BHV-tree, binary search, video card, leaf node, Morton codes.

## REFERENCES

1. Boundin Volume Hierarchy [электронный ресурс]: https://en.wikipedia.org/wiki/Bounding_volume_hierarchy
2. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees, Tero Karras, NVIDIA Research, 2012 [электронный ресурс]: https://devblogs.nvidia.com/wp-content/uploads/2012/11/karras2012hpg_paper.pdf
3. Simpler and Faster HLBVH with Work Queues, Kirill Garanzha, Jacopo Pantaleoni, David McAllister, NVIDIA [электронный ресурс]: http://www.highperformancegraphics.org/previous/www_2011/media/Papers/HPG2011_Papers_Garanzha.pdf
4. Fast BVH Construction on GPUs, C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, D. Manocha [электронный ресурс]: http://luebke.us/publications/eg09.pdf
5. Space filling curve [электронный ресурс]: https://en.wikipedia.org/wiki/Space-filling_curve
6. Z-order curve [электронный ресурс]: https://en.wikipedia.org/wiki/Z-order_curve