

Data Layout and Compact Representation of BVH Trees for Raytracing

Андрей Трифонов

27 сентября 2022 г.

References

Data Layout:

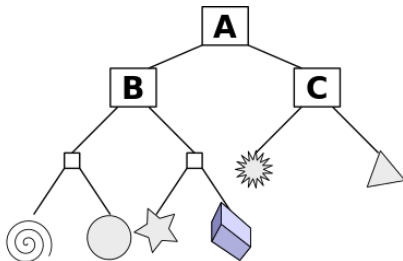
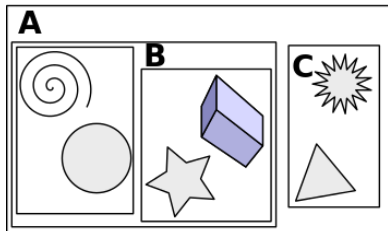
- *"Analysis of Cache Behavior and Performance of Different BVH Memory Layouts for Tracing Incoherent Rays"* by D. Wodniok, A. Schulz, S. Widmer and M. Goese
- *"Bounding Volume Hierarchy Optimization through Agglomerative Treelet Restructuring"* by Leonardo R. Domingues and Helio Pedrini

Компактное Представление:

- *"Ray Tracing with the Single Slab Hierarchy"* by Martin Eisemann, Christian Woizischke and Marcus Magnor
- *"The Minimal Bounding Volume Hierarchy"* by Pablo Bauszat, Martin Eisemann and Marcus Magnor

Что такое BVH?

Bounding Volume Hierarchy (BVH) - иерархия ограничивающих объемов (BV).



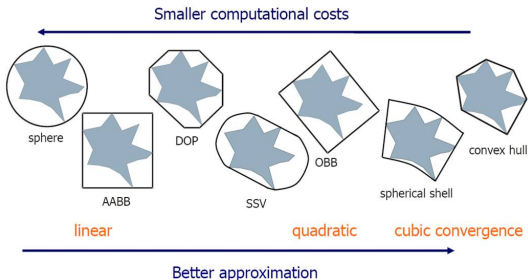
Все геометрические объекты, образующие листовые узлы дерева, заключены в эти BV.

Затем эти узлы группируются в небольшие наборы и заключаются в более крупные BV.

Типы BV

Типы ограничивающих объемов:

- Sphere tree
- AABB tree (Axis Aligned Bounding Box)
- OBB tree (Oriented Bounding Box)
- k-DOP (Discrete Oriented Polytope)
- SSV (Swept Sphere Volume)



BVH для трассировки лучей

Зачем?

BVH часто используются в трассировке лучей для устранения потенциальных кандидатов на пересечение в сцене путем пропуска геометрических объектов, расположенных в ограничивающих объемах, которые не пересекаются текущим лучом.

Путем организации BV в BVH временная сложность (количество выполненных тестов) может быть уменьшена с линейной до логарифмической от количества примитивов (треугольников).

В контексте ускоряющей структуры для рейтрейсинга используются:

- AABB (Axis Aligned Bounding Box)
- OBB (Oriented Bounding Box) (*намного реже*)

Для оценки эффективности BVH используются метрики:

- Количество обойдённых узлов (NodesCount – NC)
- Количество обойдённых листьев (LeavesCount – LC)
- Кол-во арифметических операций проверки пересечения AABB (или др.) с лучом (АОС)
- Количество тестов луч-треугольник (или кол-во арифметических операций, TC)
- Количество длинных прыжков в пределах 1 буфера (LongJumpCount – LJC)
- Объём памяти, прочитанный (BusLoadInBytes – BLB)
- Leaf Count Variability (LCV)
- Время на определённом CPU
- Время на определённом GPU (не для всех)
- Общий объём памяти на всё дерево

Функция стоимости BVH дерева

Traversal cost c_T

Средняя стоимость пересечения луча с BV

Intersection cost c_I

Средняя стоимость пересечения луча с примитивом сцены

Conditional probability $P(N_c|N)$

Условная вероятность пересечения дочернего узла N_c при пересеченном узле N

Cost function

$$c(N) = \begin{cases} c_T + \sum_{N_c} P(N_c|N)c(N_c) \\ c_I|N| \text{ (if } N \text{ is leaf)} \end{cases}$$

Расположение данных BVH (Data Layout)

Cache locality CPU & GPU

Локальность кеша относится к вероятности того, что последовательные операции будут находиться в кеше и, следовательно, будут выполняться быстрее.

GPU Texture memory

Этот аппаратный блок предоставляет свой механизм кэширования глобальной памяти GPU.

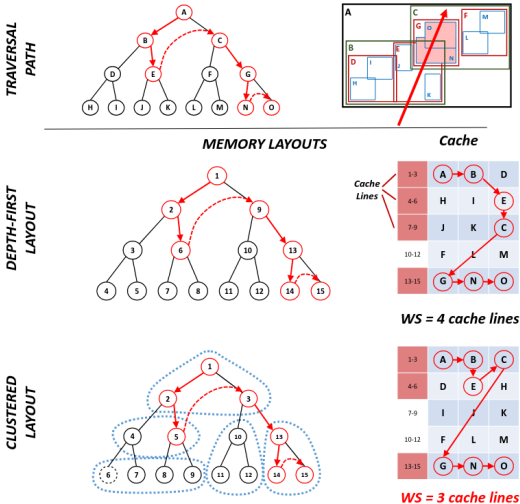
- максимизации 2D пространственной локальности
- аппаратная обработка адресов, выходящих за границы

Data Layout

Изменения порядка узлов и их внутреннего представления может улучшить **cache locality** и тем самым уменьшить **traversal cost**.

Clustered Data Layout

Расположение данных BVH



Treelets. Маленькие узлы-деревья

Идея

Объединить близкую друг к другу геометрию в **узлы-деревья** - *treelets*, чтобы повысить *cache locality*, так как следующие когерентные лучи будут попадать в загруженные в *texture memory* трилеты

Поскольку **время выполнения и требования к памяти** быстро растут вместе с размерами дерева трилета, целесообразно использовать только *маленькие древовидные структуры*

Перестановкой *treelets* можно уменьшить общую *SAH cost*
Treelet BVH (TRBVH) дерева

Создание трилетов

Treelets

```
DQ = {root} // deferred queue
MQ = {} // merge queue
p = *threshold* // [0; 1]
while (DQ != {})
    MQ = DQ.pop; ct = {};
    while (MQ != {})
        N = MQ.pop; ct += N
        for (N_c : N)
            if (SAH(N, N_c) > p)
                if (DFS)
                    MQ.push_front(N_c)
                else // BFS
                    MQ.push_back(N_c)
    TRBVH += {ct}
```

Agglomerative Treelet Restructuring

```
for (internal node i : BVH)
    treelet = FormTreelet(i)
    clusters = treeletLeaves
    while (length(clusters) > 1)
        distances = {}
        for ((x, y) : pairs(clusters))
            d = Dissimilarity(x, y)
            distances = (d, x, y)
        (m, n) = FindMinDistance(distances)
        o = MergeClusters(m, n)
        clusters.remove(m)
        clusters.remove(n)
        clusters.add(o)
```

Метрика расстояния

Agglomerative Treelet Restructuring

На каждой итерации пара узлов которые ближе друг к другу с использованием данной метрики, будут объединены.

Метрика

Поскольку цель состоит в том, чтобы минимизировать общую стоимость SAN дерева, за расстояние между двумя кластерами будем считать площадь поверхности ограничивающей рамки, содержащей их

Это дорогая операция расстояния, поэтому есть смысл кэшировать расстояния между парами кластеров заранее

Треугольная матрица расстояний

Agglomerative Treelet Restructuring

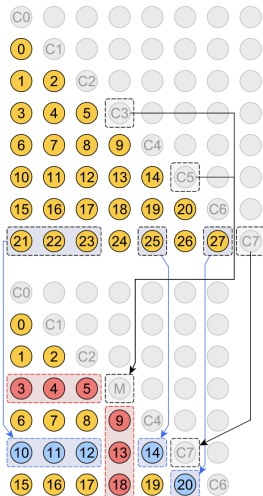
Удобно кэшировать расстояния между кластерами в треугольной матрице.

Рассмотрим изменения в матрице после объединения кластеров 3 и 5

Строка и столбец по индексу массива

$$r = 1 + \left\lfloor \frac{\sqrt{8i+1}-1}{2} \right\rfloor$$
$$c = i + \frac{r(r-1)}{2}$$

- Красные - вычислить заново
- Синие - скопировать

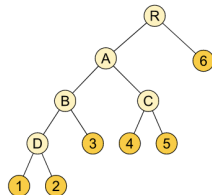
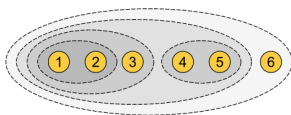
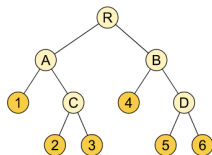


Объединение кластеров

Agglomerative Treelet Restructuring

Agglomerative clustering

На каждом шаге два кластера, находящиеся ближе друг к другу (по матрице расстояний) объединяются. Соединяем объединенные кластеры внутренним узлом (изменяя указатели)



Аллоцировать память на новые узлы не нужно.

SAH cost проверка

Agglomerative Treelet Restructuring

Все предлагаемые модификации **сохраняются в списке**, а не применяются сразу

Каждая запись списка содержит:

- индекс внутреннего узла, который необходимо изменить
- индексы двух его дочерних узлов

После обработки всего трилета *SAH cost* новой топологии сравнивается с первоначальной стоимостью трилета

Изменения вступят в силу только в том случае, если стоимость уменьшилась

Post-processing

Agglomerative Treelet Restructuring

Практика показывает, что *SAH-cost* оптимизированного дерева можно еще уменьшить, разрушив (начать хранить примитивы напрямую) некоторые поддеревья

Стоимость *разрушенного* дерева

$$c = c_t A(n) N(n)$$

n - subtree root

$A()$ - площадь поверхности

$N()$ - кол-во примитивов (треугольников) в дереве

Если полученная стоимость меньше *SAH-cost* поддеревца, то принимаем решение об замене поддеревца листом с $N(n)$ треугольников

Параметры

Agglomerative Treelet Restructuring

- размер трилета (кол-во узлов на трилет)
Оптимальное значение: 9
- количество итераций (полные перестроения всей TRBVH)
Оптимальное значение: 2
- γ (сколько листьев должно быть в трилете)
Оптимальное значение: `treelet_size`, и удваивать на каждой итерации

Соотношение скорость качество

γ определяет может ли узел быть использован в качестве корня трилета. Чем больше γ тем быстрее скорость конструкции. Уменьшение γ приводит к улучшению качества.

Обход дерева снизу вверх

Agglomerative Treelet Restructuring

Каждый тред начинается с обработки листа.

Avoiding race conditions (для бинарного дерева)

По умолчанию 2 треда будут достигать каждый узел. Введем правило, что за родительский узел берется только второй тред, а первый остается неактивным.

Получается при обходе кол-во неактивных тредов в варпе быстро растет.

Память нужная для алгоритма

Agglomerative Treelet Restructuring

В *shared memory* располагаются 3 массива:

- *leaf indices*
- *internal node indices*
- *leaf surface areas*

Leaf surface areas используются только при формировании трилета, так что этот массив можно использовать для других целей при престроении трилета.

Когда кластеры объединяются, можно использовать последний неиспользуемый узел из *internal node indices* для хранения информации в новом кластере.

Занимаемая память

Agglomerative Treelet Restructuring

52 byte of global memory на узел:

- указатели на родителя и на двух дочерних узлов
- индекс
- площадь поверхности
- *AABB (32 byte)*

Прочие траты:

- атомарные счетчики (*4 byte / node*)
- кол-во треугольников под узлом (*4 byte / node*)
- *SAH-cost* каждого узла (*4 byte / node*)
- агломеративное расписание (необязательно) (*256 byte / treelet size of 32*)
- матрица расстояний (при размере трилета > 21)
(*number of cluster combinations * 4bytes / warp*)

Результаты

Agglomerative Treelet Restructuring

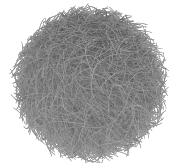
Sponza (262K)

Method	Mrays/s	Time	SAH	Memory	Relative
LBVH	56.17	7.91 ms	114.42	7 MB	69.23%
TRBVH	81.13	37.86 ms	75.75	50 MB	100.00%
ATRBVH	85.99	26.43 ms	75.42	12 MB	105.99%
ATRBVH*	77.80	30.36 ms	75.88	12 MB	95.90%

Hairball (2.9M)

Method	Mrays/s	Time	SAH	Memory	Relative
LBVH	15.33	65.71 ms	541.90	77 MB	92.41%
TRBVH	16.59	374.22 ms	478.08	520 MB	100.00%
ATRBVH	16.44	255.24 ms	475.72	121 MB	99.10%
ATRBVH*	16.44	289.69 ms	477.46	121 MB	99.10%

Method	Performance (%)	Time (%)
LBVH	80.8	20.3
TRBVH	100	100
ATRBVH	99.7	69.5
ATRBVH*	99.9	78.4



Плюсы и минусы

Agglomerative Treelet Restructuring

Достоинства:

- 30% быстрее построение ATRBVH, по сравнению с базовым методом TRBVH (при схожем качестве)
- Очень параллелизируемый
- Настраиваемый и гибкий
- Кол-во необходимой временной памяти сведено к минимуму

Недостатки:

- Параметры не настраиваются динамически

Incremental Traversal

Это техника для повышения эффективности рейтрейсинга путем дешевой арифметики с уменьшенной точностью.

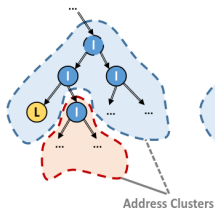
Идея заключается в **последовательном приближении начала луча к пересекаемому узлу**. Это и позволяет уменьшить точность для теста пересеканости луча с плоскостью.

Также вычислительные затраты можно уменьшить используя тесты пересечений с родительскими узлами (*parent plane sharing*)

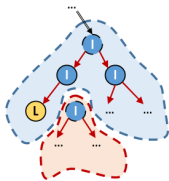
Алгоритм двухуровневой кластеризации

- 0 Input: обычное BVH (DFS) **T** с размером указателей на дочерние узлы размером **n** бит
- 1 Из корня дерева **T** создаем адресные кластеры **AC** по *COLBVH*
- 2 Для кадого дочернего **AC** создаем *glue node* **G**, "склеивая" с родительским узлом, и повторяем 1
- 3 Для каждого **AC** мы рекурсивно создаем кэш-кластеры **CC** по *COLBVH*

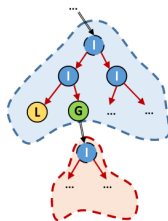
1) Forming Address Clusters



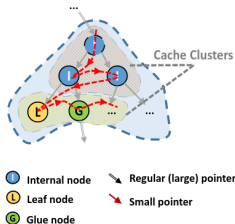
2a) Low-Res. Child pointers



2b) Glue Node Insertion



3) Cache-Aware Reordering



Кластеры адресов. Функция BuildAC

Двухуровневая кластеризация

```
BuildAC(dstOffset, srcRoot) -> offset_t
  maxN = 2^ptr_bits; AC = {}
  child_nodes = {src_root}; child_ACs = {}
  while ((size(AC + children) < maxN)
    && !children.is_empty)
    node = pop_max_SA(child_nodes)
    AC << node
    if (is_internal(node))
      children << node.left << node.right
  for (node : child_nodes)
    if (is_internal(node))
      child_ASs << node << make_glue(node)
    else
      AC << node
```

...

Кластеры адресов. Функция BuildAC (продолжение)

Двухуровневая кластеризация

```
dst_offset += BUILDCCS(dst_offset, AC)
for (root_node : roots(child_ACs))
    dst_offset = align(dst_offset, cache_layout)
    update_parent_glue_node(root_node)
    dst_offset = BuildAC(dst_offset, root_node)
return dst_offset
```

update_parent_glue_node()

Офсеты корней дочерних кластеров неизвестны на момент создания склеивающего узла, а значит должны быть занесены в *glue nodes* позже

Чтобы шаг кэш-кластеринга имел смысл, нужно чтобы корень каждого **АС** был выровнен по размеру кэш-линии

Кэш-кластеры. Функция BuildCCs

Двухуровневая кластеризация

```
BuildCCs(dst_offset, AC) -> offset_t
    maxN = cache_line_size / node_size
    CC_roots = get_root(AC); deferred_CCs = {}
    while (!CC_roots.is_empty)
        child_nodes << CCRoots; CC = {}
        while (size(CC) < maxN && !child_nodes.empty)
            node = pop_max_SA(child_nodes)
            CC << node
            if (is_internal(node))
                childNodes << node.left << node.right
        CC_roots << {childNodes + CCRoots} // DFS
        if (size(CC) == maxN)
            dst_offset = WriteCluster(dst_offset, CC)
        else deferred_CCs << CC
    for (CC : deferred_CCs) // complete but not full
        dst_offset = WriteCluster(dst_offset, CC)
    return dst_offset
```

Кэш-кластеры. Функция WriteCluster

Двухуровневая кластеризация

Функция *WriteCluster* записывает кэш-кластер в **BVH**

```
WriteCluster(dst_offset, CC) -> offset_t  
    update_child_ptr(get_parent(CC[0]));  
    for (i in [0; size(CC)])  
        dstBVH[dst_offset] = CC[i];  
        dst_offset++  
    return dst_offset
```

update_child_ptr()

Записать указатели на дочерние узлы, адрес которых мы не знали раньше

Дальнейшие оптимизации

Двухуровневая кластеризация

- **Padding:** Кэш-кластер, следующий за неплодным кэш-кластером, стоит хранить с отступом от него – так, чтобы начало **СС** совпадало с началом кэш-линии
Нужно следить (ограничивать паддинг), чтобы узлы не выходили за пределы small pointers
- **Cluster merging:** Можно объединять малютенькие собранные кластеры, объединяя их узлы и сортируя по SA. Чтобы более большие узлы были загружены в кэш-линию

Компактное представление BVH

Существует несколько подходов, которые пытаются **минимизировать использование памяти**, используя один или несколько из следующих методов:

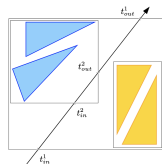
- **Сокращение информации**, которая хранится для каждого BV
- **Снижение точности** данных, которые хранятся в BV
- Удаление дочерних и примитивных указателей **неявным индексированием**
- Повышение коэффициента ветвления (**числа потомков на узел**), чтобы уменьшить общее количество узлов в BVH
- **Сжатие** данных иерархии

Идеи сокращения информации BVH

Enclosing Property of BVH

Ограничивающее свойство BVH обеспечивает, что луч всегда:

- пересекает сначала (или одновременно) узел предок, прежде чем он сможет пересечь дочерние узлы
- всегда будет покидать дочерние узлы раньше или одновременно узла-предка



Еще одно полезное свойство, касающееся эффективности

BV дочерних элементов узла имеют общие ограничивающие плоскости со своим родителем.

Структура узла

Двухуровневая кластеризация

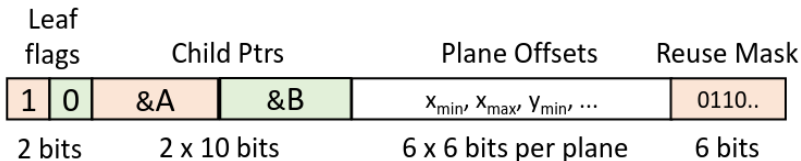
Для имплементации двухуровневой кластеризации было необходимо уменьшить размер узла:

- **Сокращение информации:** *повторное использование плоскостей родительских узлов*

Для пары дочерних узлов храним 6 плоскостей и *reuse mask*

- **Снижение точности:** *квантизация узла*

Кол-во квантизированных битов определяет компромисс между объемом памяти и качеством BVH



Плюсы и минусы

Двухуровневая кластеризация



Single Slab Hierarchy

The single slab hierarchy (SSH)

Иерархия одиночных пластин - полное k -арное дерево BVH, которое хранится в массиве и индексируется как heap. Каждый узел либо листовый, либо имеет k дочерних. Это способ разделения BV надвое одной пластиной, в результате чего получается k -ичное дерево.

Принцип, используемый для этого компактного представления BVH:

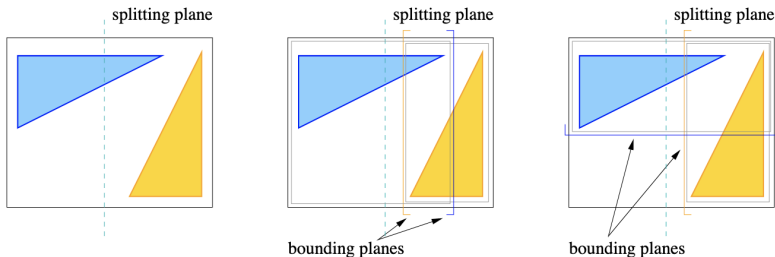
- **Сокращение информации**, которая хранится для каждого BV

Bounding Plane Adjustment

Single Slab Hierarchy

Чтобы не оставлять пустот в BVH, для каждого дочернего узла мы подбираем свою ось вдоль которой разместим splitting plane (single slab).

Значит, помимо координаты по оси нашей splitting plane, для каждого узла нужно хранить выбранную ось (как флаг).



Структура Узла

Single Slab Hierarchy

```
#pragma align(8)
struct SSHNode {
    float plane;
    union {
        int firstChildNodeID; //inner nodes
        int firstTriangleID; //leaf nodes
        // bit 0..1: x/y/z/leaf
        // bit 2: left/right interior
        // bit 3..4: traversal axis
    }
};
```

Заняв 5 битов флагами у нас остается 27 битов (от int32), чтобы сохранить индекс. Таким образом можно закодировать до 134 мил. примитивов.

Построение

Single Slab Hierarchy

```
void createSSH(TriangleList tris, AABB parent,
               SSHNode& node) {
    AABB bounds(tris); // bounding box
    float bestSurface = HUGE_VAL;
    int bestSide = 0;
    for (Side side : make_sides(bounds)) {
        AABB temp(parent);
        temp.side = side;
        if (surface(temp) < bestSurface) {
            bestSurface = surface(temp);
            bestSide = side;
        }
    }
    ...
}
```

Построение (продолжение)

Single Slab Hierarchy

```
...
node.boundingPlane = boudns.bestSide;

if (tris.size() < n) {
    createLeafNode(); return; }

TriangleList leftTris, rightTris;
subdivide(tris, leftTris, rightTris); // SAH
parent.bestSide = boudns.bestSide;

int childID = node.firstChildNodeID;
createSSH(leftTris, parent, childID);
createSSH(rightTris, parent, childID + 1);
}
```

Обход SSH

Single Slab Hierarchy

```
bool intersectSSH(Ray& ray, mask reverse[3],
                  SSHNode* node, float& t_hit,
                  float& t_near, float& t_far) {

    // 1. Calculate the distance to its BP
    int axis = node->getSlabAxis();
    float t = (node->plane - ray.origin[axis])
              / ray.dir[axis];
    // 2. Compare it to the active ray segment
    if (node->geometryIsLeft()) {
        t_near = (reverse[axis] | (t <= t_near))
                ? t_near : t;
        t_far  = (reverse[axis] & (t < t_far))
                ? t : t_far;
    }
}
```

...

Обход SSH (продолжение)

Single Slab Hierarchy

```
...
} else {
    t_near = (reverse[axis] & (t > t_near))
           ? t_near : t;
    t_far  = (reverse[axis] | (t >= t_far))
           ? t : t_far;
}

if ((t_near > t_far) || (t_near > t_hit)) {
    return false;
}

if (node->isLeaf()) { ... } else { ... };
}
```

Extension to dynamic scenes

Single Slab Hierarchy

Skinned meshes

Можно применить простой bottom-up подход, используя тривиальные min/max операции для адаптации границ узлов. Структура иерархии остается неизменной. Этого достаточно для большинства когерентных анимаций.

Arbitrary movements

Можно параллельно перестраивать SSH. Как только построение будет завершено - сделать замену.

Плюсы и минусы

Single Slab Hierarchy

Достоинства:

- Сжатие 75%
- Подходит для интерактивного рейтрейсинга
- Возможность адаптировать под динамические сцены
- Более легкий алгоритм траверса (в 6 раз быстрее чем пересекать с AABV)
- Не изменена структура самого дерева - изменено только представление узлов
- Сравним по скорости с обычным AABV BVH







Недостатки:

- Немного дольше строить
- В два раза больше узлов проходит по сравнению с AABV BVH
- В среднем на один треугольник на луч пересекается больше

Результаты

Single Slab Hierarchy

- N_T : avg of traversed nodes per ray
- N_I : avg of ray-object intersections per ray
- T_C : total time needed for **construction**
- T_R : total time needed for **traversal**
- $s(T_R)$: speedup achieved by the SSH with respect to the BVH
- Mem : memory usage of the AS in megabytes, excluding the triangle data

	Method	N_T	N_I	T_C	T_R	$s(T_R)$	Mem
	Scene - Power Plant - 12,748,510 triangles, 640 × 480						
	BVH	43.55	4.24	28.83s	3.31s	1.0×	778.11MB
	SSH	80.23	5.34	28.9s	2.73s	1.22×	194.53MB
	Scene - Happy Buddha - 1,087,716 triangles, 640 × 480						
	BVH	6.45	0.63	1.93s	0.54s	1.0×	66.39MB
	SSH	14.14	1.53	1.98s	0.59s	0.91×	16.60MB
	Scene - Dragon - 871,414 triangles, 640 × 480						
	BVH	10.47	0.95	1.55s	0.84s	1.0×	53.19MB
	SSH	23.21	2.11	1.57s	0.89s	0.94×	13.30MB
	Scene - Fairy 1st frame - 174,117 triangles, 640 × 480						
	BVH	26.52	2.53	0.30s	2.06s	1.0×	10.63MB
	SSH	45.51	4.23	0.31s	1.70s	1.21×	2.66MB
	Scene - Toys - 11,141 triangles, 640 × 480						
	BVH	29.57	1.31	0.02s	2.08s	1.0×	0.68MB
	SSH	46.85	3.24	0.02s	1.57s	1.33×	0.17MB
	Scene - Bunny - 69,451 triangles, 640 × 480						
	BVH	8.77	0.55	0.09s	0.66s	1.0×	4.24MB
	SSH	18.01	1.31	0.10s	0.65s	1.03×	1.06MB

The Minimal Bounding Volume Hierarchy

The Minimal Bounding Volume Hierarchy (MVH)

Минимальная иерархия ограничивающих объемов - полное k -арное дерево BVH, которое хранится в массиве и индексируется как heap. Каждый узел либо листовой, либо имеет k дочерних.

Принципы, используемые для этого компактного представления BVH:

- Удаление дочерних и примитивных указателей **неявным индексированием**
- **Сокращение информации**, которая хранится для каждого BV

Неявное индексирование

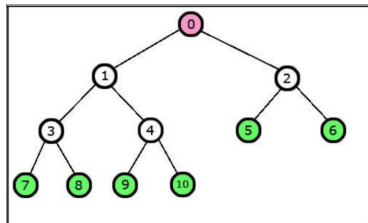
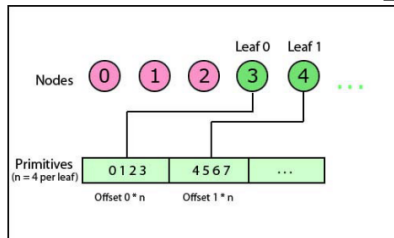
Minimal Bounding Volume Hierarchy

Индексы дочерних узлов i -ого узла между $i * k + 1$ и $i * k + k$

В статье-источнике $k = 2$, то есть дерево - бинарное.

Обозначим кол-во примитивов на узел n . И пусть листовые узлы начинаются с индекса l .

$primID = (i - l) * n$ - будет индекс первого примитива в массиве примитивов для i -ого узла (если он листовой, то $i > l$).

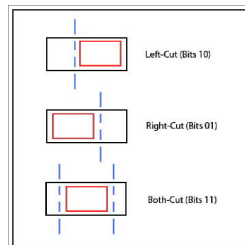


Сокращение хранимых данных для узла MVH

Minimal Bounding Volume Hierarchy

Выделим для узла всего 2 бита. Получим 4 возможных варианта для получившегося дочернего узла:

- No-Cut: if no surface reduction is possible for this node
- Left-Cut: if the minimum slab is increased
- Right-Cut: if the maximum slab is decreased
- Both-Cut: if the Left-Cut and Right-Cut is used



Каждый раз мы используем фиксированный reduction factor ζ
Лучшие результаты с $\zeta \in [0.35, 0.30]$

Построение MVH. Алгоритм

Minimal Bounding Volume Hierarchy

- 1 Дублируем последний примитив пока P не станет $P \% n = 0$
- 2 Сколько узлов всего? Для $k = 2$ кол-во узлов $2(P/n) - 1$
- 3 Сколько узлов в каждом поддереве?
 - Присваиваем всем листьям их кол-во примитивов
 - Суммируем bottom-up, пока не дойдем до корня
- 4 Пусть всего N узлов. Тогда аллоцируем массив длины $2N$ из *int32*
- 5 MVH строится с использованием object median split, где **процесс разделения** использует предварительно вычисленные количества примитивов для разделения списка объектов на две части

Процесс разделения BV

Minimal Bounding Volume Hierarchy

- 1 Отсекаем от рамки родителя способами 01, 10, 11 и сравниваем с реальным BV для текущего списка примитивов
- 2 Если один из способов нам подходит - записываем в массив эти 2 бита.
- 3 Как только дошли до листа - заносим примитивы в массив примитивов под соответствующими индексами

На каждом этапе разделения выбирается ось, вдоль которой располагаются самые длинные грани приближенного AABB

Глобальные параметры

- Root's BB
- Reduction factor ζ

Обход MVH

Minimal Bounding Volume Hierarchy

Алгоритм обхода такой же, как и для Single Slab, однако строить splitting plane приходится на самом этапе обхода.

```
float minTab[4] = { 0.0f, zeta, 0.0f, zeta };
float maxTab[4] = { 0.0f, 0.0f, zeta, zeta };
bool intersect(Ray& r, float& tHit, int* mvhArray,
               int node, AABB& parent,
               float tNear, float tFar) {
    // reconstruct AABB
    vec3 size = parent.max - parent.min;
    char axis = GetAxisOfMaximumExtent(size);
    char bitID = GetNodeCut(node);
    parent.min[axis] += size[axis]*minTab[bitID];
    parent.max[axis] -= size[axis]*maxTab[bitID];

    ...
}
```

Обход MVH (продолжение)

Minimal Bounding Volume Hierarchy

```
...  
// intersection  
float tmin, tmax;  
parent.intersect(r, tmin, tmax, axis);  
float tNear = max(min(tmin, tmax), tNear);  
float tFar = min(max(tmin, tmax), tFar);  
return ((tNear <= tFar) && (tNear <= tHit));  
}  
  
char GetNodeCut(int node, int* mvhArray) {  
    int iIndex = node >> 4;  
    int iShift = (node & 15) << 1;  
    return ((mvhArray[iIndex] >> iShift) & 0x3);  
}
```

Двухуровневая MVH. TLAS и BLAS

Minimal Bounding Volume Hierarchy

Идея

- Половина всех затрат памяти приходится на последний уровень дерева BVH (вдвое больше узлов)
- Большинство лучей пересекают верхние уровни дерева - надо сделать их качественными

Top level acceleration structure (TLAS)

Верхний уровень в несжатом BVH формате, оптимизированный с помощью затратного по времени SAH

Bottom level acceleration structure (BLAS)

Нижний уровень состоит из разных MVH, на которые ссылается верхний уровень как на листья.

Результаты

Minimal Bounding Volume Hierarchy

Scene	Tris	BVH	MVH	Ratio BVH:MVH	2-level MVH	Ratio BVH:2-level MVH
Bones	4,204	70.75KB	0.51KB	100:1	26.59KB	3:1
Sponza	67,462	797.94KB	8.23KB	97:1	41.02KB	20:1
Office	385,376	1,636.50KB	47.04KB	35:1	72.71KB	23:1
Fairy	174,117	1,957KB	21.25KB	92:1	50.41KB	39:1
Cars	549,662	6,051.63KB	67.09KB	90:1	170.20KB	36:1
Dragon	871.306	10.44MB	0.11MB	101:1	142.08KB	75:1
Buddha	1,087,716	13.39MB	0.13MB	101:1	168.50KB	81:1

	ts	tt	rt	ts	tt	rt	ζ	Loss	ts	tt	rt	ζ	Loss
Bones	2.5	1.3	0.10s	67.7	84.3	2.70s	0.3	1:27	3.2	2.7	0.16s	0.3	1:1.6
Sponza	39.2	16.7	0.98s	3,828	415	160.7s	0.3	1:164	159	196.6	7.35s	0.3	1:7.5
Office	26.1	63.3	1.42s	1,992	262	328.2s	0.4	1:231	110	153.6	5.11s	0.4	1:3.6
Fairy	21.0	10.5	0.56s	12,690	3,740	519.0s	0.35	1:926	72.4	83.9	3.46s	0.35	1:6.1
Cars	44.5	15.4	1.06s	4,461	1,943	221.0s	0.4	1:208	148.9	186.4	7.07s	0.4	1:6.5
Dragon	19.3	6.6	0.51s	2,081	1,676	66.8s	0.35	1:131	154.1	188.0	8.04s	0.35	1:15.7
Buddha	2.5	0.76	0.11s	311	262	10.37s	0.35	1:94	23.3	28.4	1.27s	0.35	1:11.5
Bones	19.1	219	0.02s	316.4	369.9	0.06s	0.3	1:3	24.6	256.6	0.02s	0.3	1:1.09
Sponza	190.8	22	0.08s	1,542	3,393	1.87s	0.3	1:25	784.6	2,102	0.13s	0.3	1:1.71
Office	179.2	2,961	0.10s	25,002	7,895	3.26s	0.4	1:33	702.5	6,464	0.17s	0.4	1:1.74
Fairy	185.6	2,337	0.08s	49,250	8,114	3.83s	0.35	1:46	683.9	5,527	0.17s	0.35	1:1.98
Cars	332.9	2,671	0.12s	20,458	14,350	2.36s	0.4	1:20	1,067	6,996	0.23s	0.4	1:1.96
Dragon	391.0	4,399	0.15s	11,244	19,142	1.55s	0.35	1:11	1,777	15,468	0.40s	0.35	1:2.71
Buddha	281.3	1,191	0.10s	3,042	20,939	0.58s	0.35	1:6	882.6	11,155	0.25s	0.35	1:2.38

Плюсы и минусы

Minimal Bounding Volume Hierarchy

Достоинства:

- 100:1, 30:1 compression ratio для MVH и двухуровневого решения соотв.
- Подходит для интерактивного рейтрейсинга
- Подходит для невероятно больших сцен
- Подходит для интерактивного рейтрейсинга (двухуровневая реализация только если)
- Настраиваемый компромисс между объемом дерева и скоростью

Недостатки:

- Немного дольше строить
- Сильно страдает от некогерентных лучей