

ОСОБЕННОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ ТРАССИРОВКИ ЛУЧЕЙ НА CUDA

Докладчик:

Фролов В.А. (ВМиК МГУ)

Научный руководитель:

Игнатенко А.В. (ВМиК МГУ)

Лекторы:

Боресков А.В. (ВМиК МГУ)

Харламов А.А. (NVidia)

План



- ⌘ RT – что, зачем и как?
- ⌘ Метод грубой силы
 - ☐ CUDA
- ⌘ Ускоряющие структуры
 - ☐ Регулярные и иерархические сетки
 - ☐ BVH
 - ☐ kd деревья

Ray Tracing

⌘ Фотореалистичный синтез изображений



⌘ POV-Ray

Ray Tracing

⌘ Фотореалистичный синтез изображений



⌘ POV-Ray

Real Time Ray Tracing

⌘ Скорость в ущерб качеству



Ray Tracing



⌘ Точность

- ☑ Path tracing
- ☑ Фотонные карты
- ☑ Распределенная трассировка лучей (стохастическая)

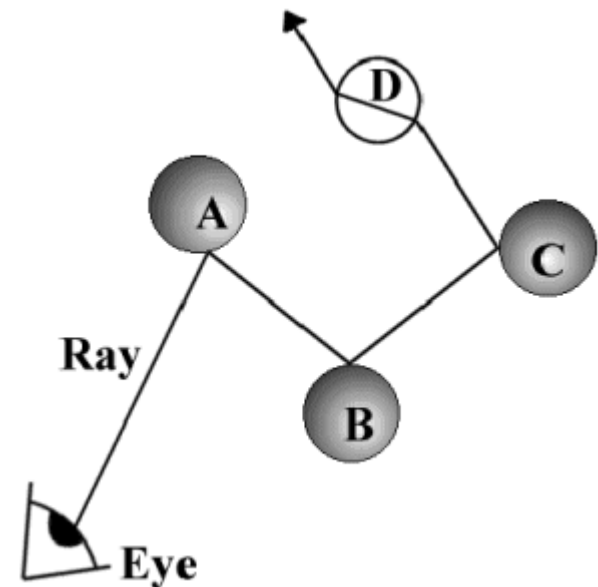
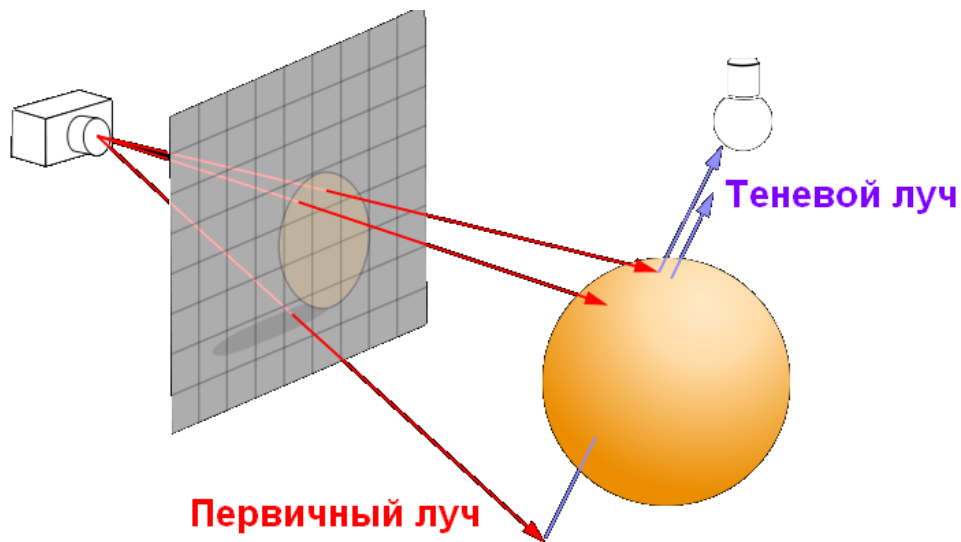
⌘ Скорость

- ☑ Обратная трассировка лучей
- ☑ Растеризация + обратная трассировка лучей

Обратная трассировка лучей

⌘ Алгоритм

☒ Первичные, теневые, отраженные лучи

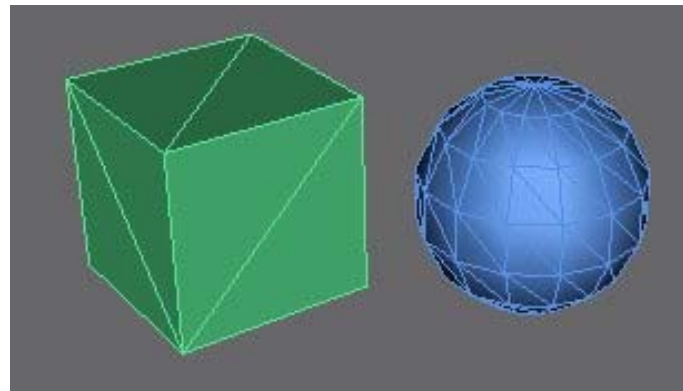
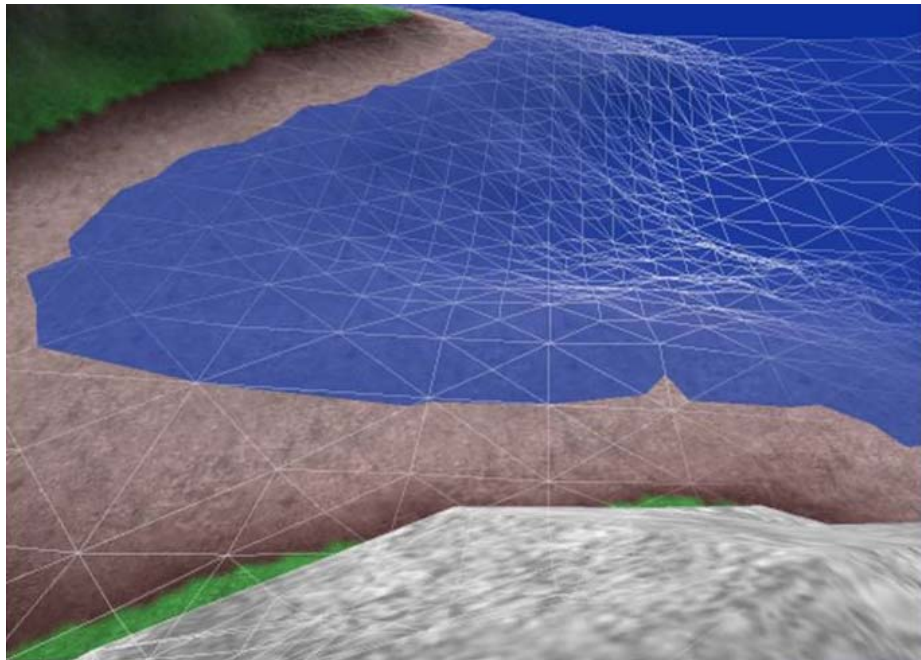
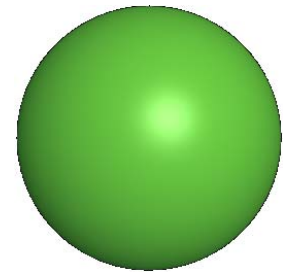


Ray Tracing

⌘ Представление 3D объектов

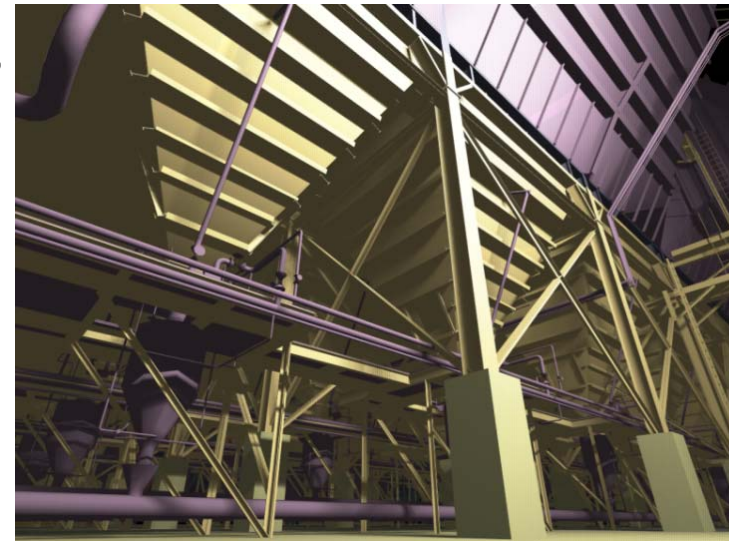
☒ Аналитическое

☒ Меши из треугольников



Ray Tracing

- ⌘ Поверхность задана как массив треугольников
- ⌘ Узкое место – поиск пересечения луча с поверхностью
 - ☑ 1 000 000 треугольников
 - ☑ 1 000 000 лучей
 - ☑ $\Rightarrow 10^{12}$ операций
 - ☑ $(\log(N))^k * 10^6$ ($k \sim [1..2]$)



Пересечение луча и треугольника

⌘ Простой вариант

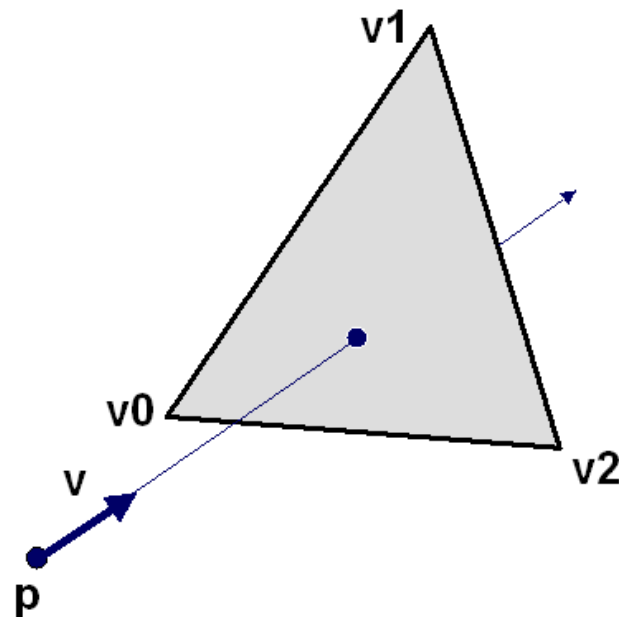
☐ $Ax + By + Cz + D = 0$

☐ Найти t

☐ $x = p.x + v.x * t$

☐ $y = p.y + v.y * t$

☐ $z = p.z + v.z * t$



$$t = - \frac{(A * p.x + B * p.y + C * p.z + D)}{A * v.x + B * v.y + C * v.z}$$

Пересечение луча и треугольника

⌘ Простой вариант

☐ t известно

☒ $z = p + v * t$

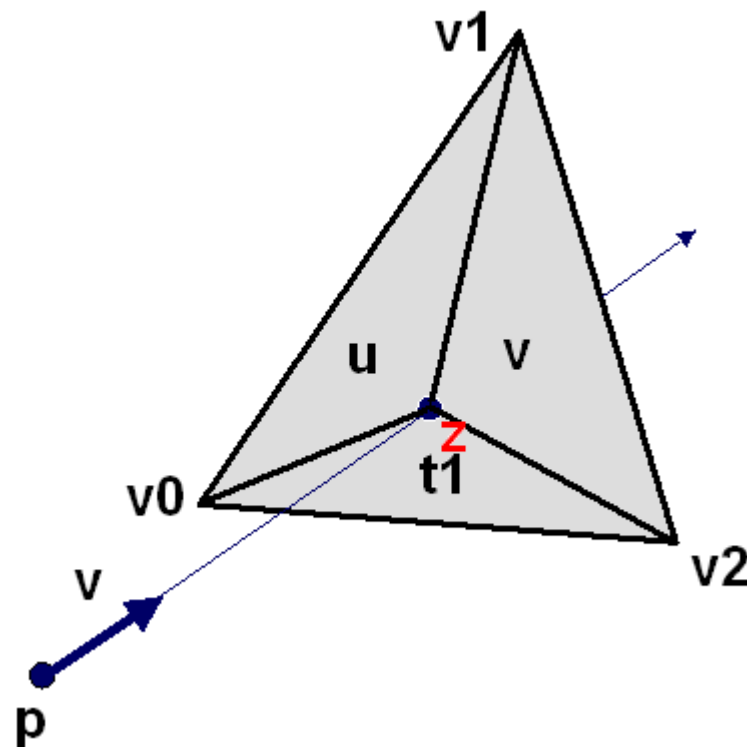
☒ $S = \text{cross}(v1-v0, v2-v0)$

☒ $u = \text{cross}(v1-z, v0-z)$

☒ $v = \text{cross}(v1-z, v2-z)$

☒ $t1 = \text{cross}(v2-z, v0-z)$

☐ $|u + v + t1 - S| < \varepsilon$



Пересечение луча и треугольника

⌘ Оптимизированный вариант

☒ Барицентрические координаты

☒ $u := u/S, v := v/S, t1 := t1/S$

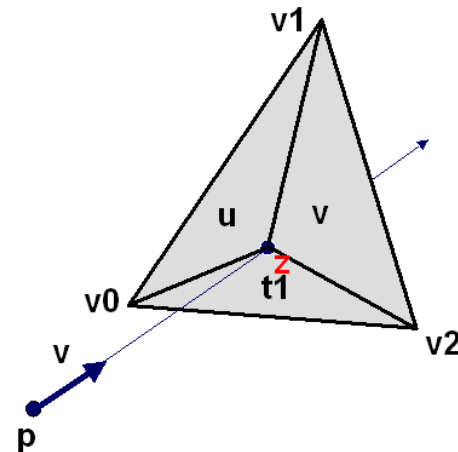
☒ $t1 = 1 - u - v$

$$z(u, v) = (1 - u - v) * v1 + u * v2 + v * v0$$

$$z(t) = p + t * d$$

$$p + t * d = (1 - u - v) * v1 + u * v2 + v * v0$$

☒ 3 уравнения, 3 неизвестных



Пересечение луча и треугольника

⌘ Оптимизированный вариант

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\text{dot}(P, E1)} * \begin{bmatrix} \text{dot}(Q, E2) \\ \text{dot}(P, T) \\ \text{dot}(Q, D) \end{bmatrix}$$

$$E1 = v1 - v0$$

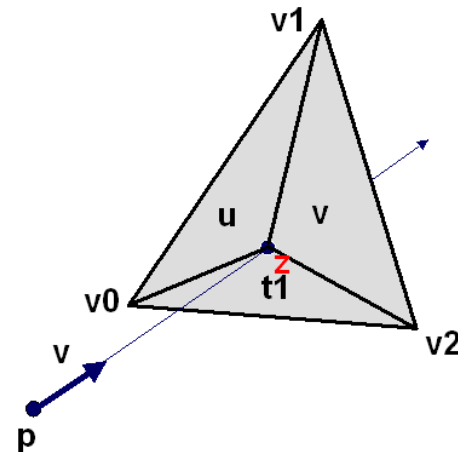
$$E2 = v2 - v0$$

$$T = p - v0$$

$$P = \text{cross}(D, E2)$$

$$Q = \text{cross}(T, E1)$$

$$D = v$$



Пересечение луча и треугольника

⌘ Простой вариант

☑ Операции ($*$: 39, $+/-$: 53, $/$: 1)

☒ 248-404 тактов

⌘ Оптимизированный вариант

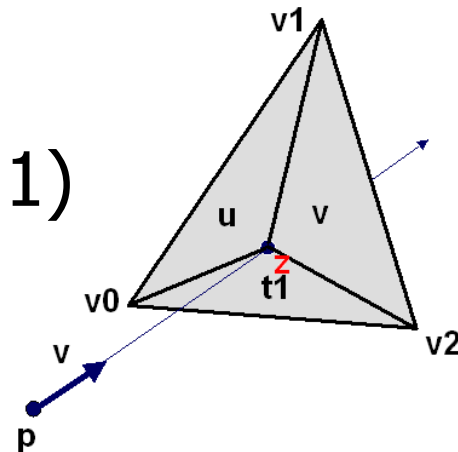
☑ Операции ($*$: 23, $+/-$: 24, $/$: 1)

☒ 132-224 такта

⌘ Как считали нижнюю оценку?

☑ использование mad вместо mul и add

☑ $4 * (N_mul + |N_add - N_mul|)$



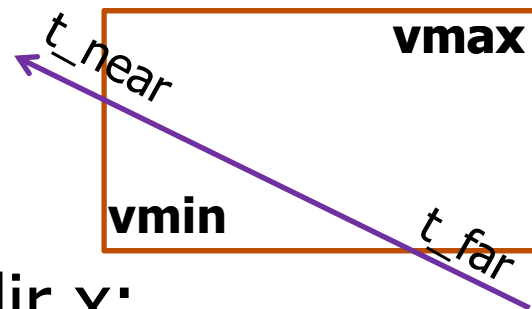
Другие примитивы

⌘ Бокс – это 6 плоскостей

⌘ $(vmin.x - r.pos.x) / r.dir.x;$

⌘ $(vmin.x + rInv.pos.x) * rInv.dir.x;$

⌘ 6 add и 6 mul == 12 mad, 48 тактов



⌘ Сфера

⌘ $\sim 13 \text{ mad} + \text{sqrtf} == 52 + 32 = 84 \text{ такта}$

⌘ меньше ветвлений

⌘ Иерархия из сфер не лучше иерархии из боксов

Multiprocessor Occupancy

⌘ Регистры

$$\text{occupancy} = \frac{\text{active warps}}{\text{max warps}}$$

☑ 8192 регистра на SM

☑ Блоки по 8x8 нитей

☑ 128 регистров на нить

☒ nvcc не дает столько регистров, почему?

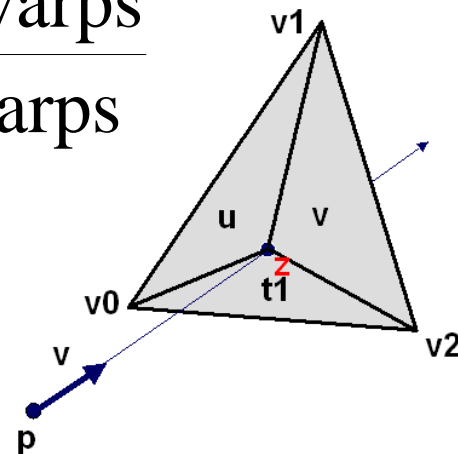
☒ рег ≤ 40: 3 блока, 6 warp-ов активны

☒ рег ≤ 32: 4 блока, 8 warp-ов активны

☒ рег ≤ 24: 5 блоков, 10 warp-ов активны

☒ рег ≤ 20: 6 блоков, 12 warp-ов активны

☒ рег ≤ 16: 8 блоков, 16 warp-ов активны

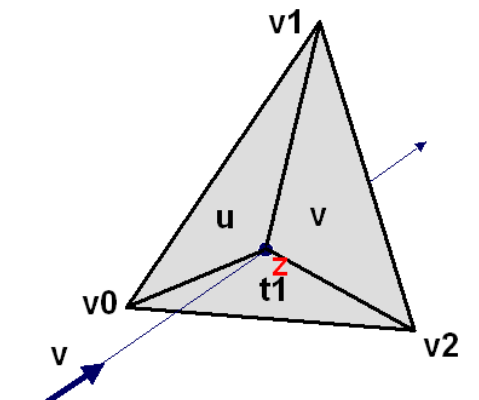


Пересечение луча и треугольника

⌘ Регистры

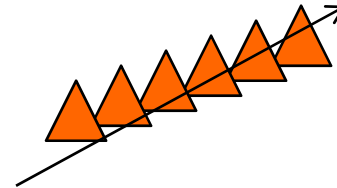
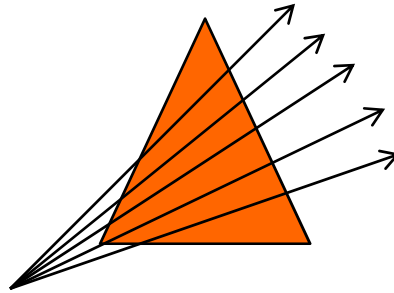
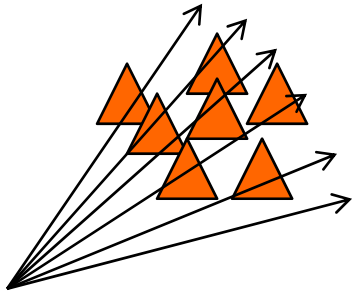
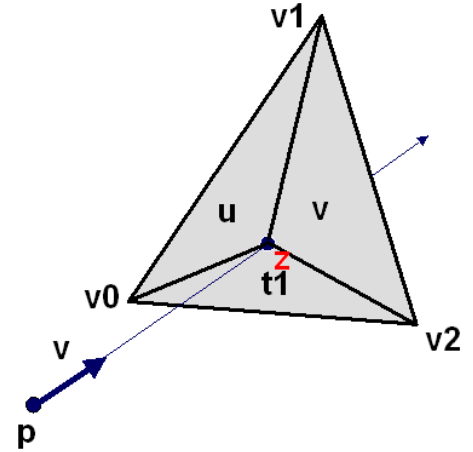
- ☑ 6 регистров на луч
- ☑ 9 регистров на вершины
- ☑ 3 регистра на (t, u, v)
- ☑ 1 регистр на triNum
- ☑ 1 на счетчик в цикле
- ☑ 1 как минимум на tid
- ☑ 2 на min_t и min_id

⌘ 23 уже занято!


$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{p_1}{\text{dot}(P, E1)} * \begin{bmatrix} \text{dot}(Q, E2) \\ \text{dot}(P, T) \\ \text{dot}(Q, D) \end{bmatrix}$$
$$E1 = v1 - v0$$
$$E2 = v2 - v0$$
$$T = p - v0$$
$$P = \text{cross}(D, E2)$$
$$Q = \text{cross}(T, E1)$$
$$D = v$$

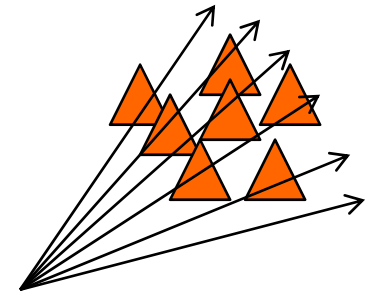
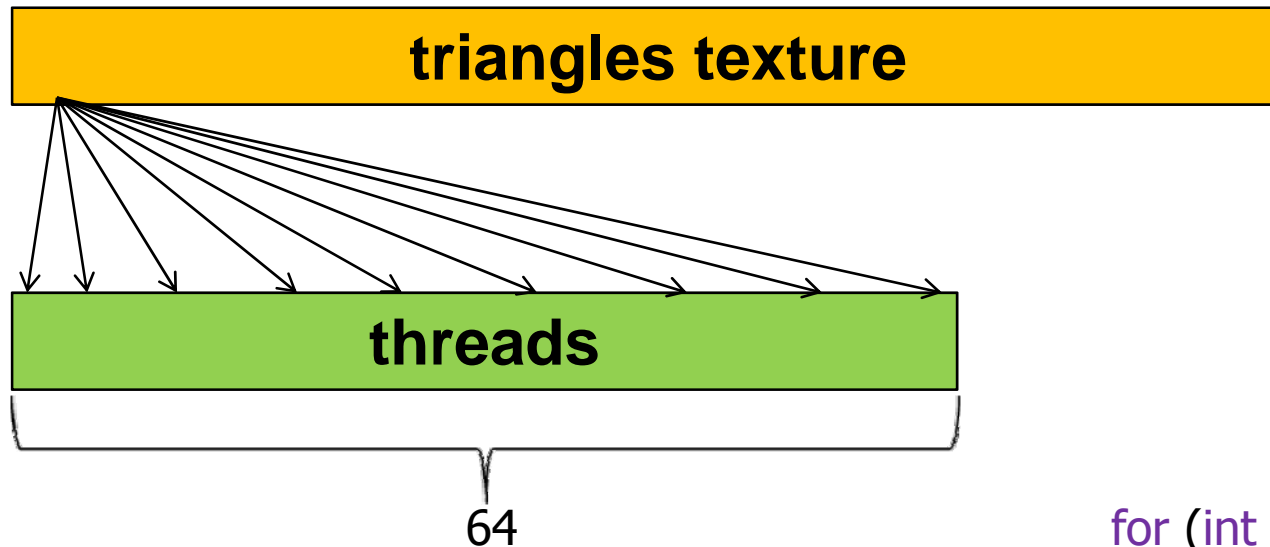
Пересечение луча и треугольника

- ⌘ Организация сетки - блоки 8x8
- ⌘ Что общее для нитей блока?
 1. Свой луч, свой треугольник
 2. Свой луч, общий треугольник
 3. Общий луч, свой треугольник



Пересечение луча и треугольника

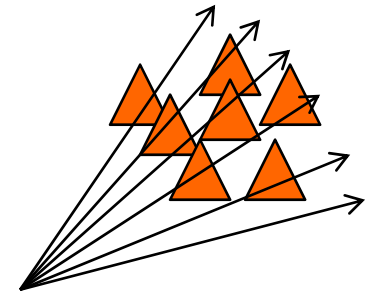
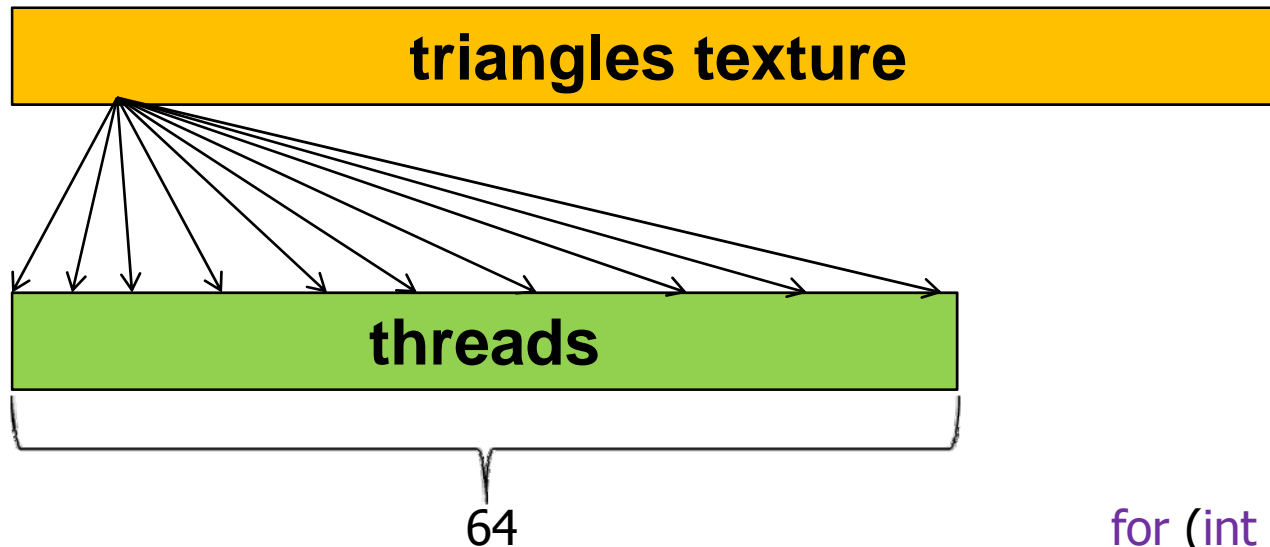
⌘ Свой луч, свой треугольник



```
for (int i=0;i<triNum;i++)  
{  
    (A,B,C) = tex1Dfetch(tex,i);  
    // intersection code  
}
```

Пересечение луча и треугольника

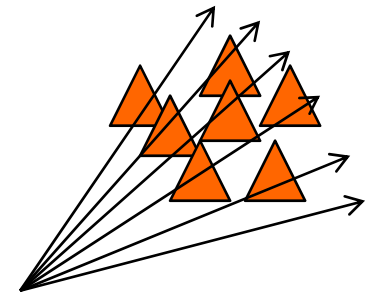
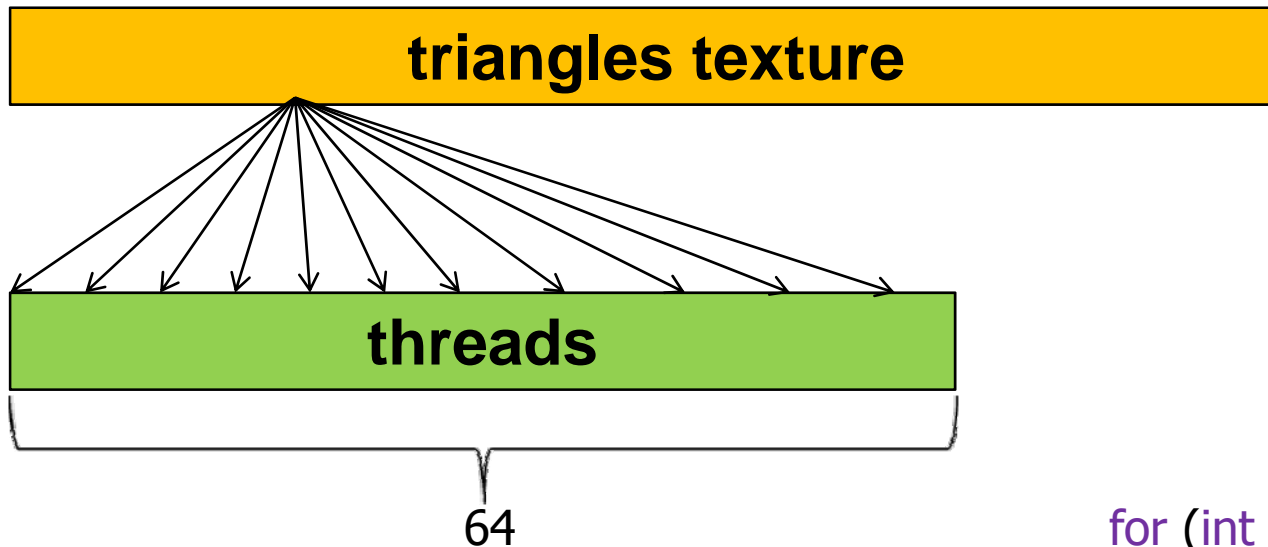
⌘ Свой луч, свой треугольник



```
for (int i=0;i<triNum;i++)  
{  
    (A,B,C) = tex1Dfetch(tex,i);  
    // intersection code  
}
```

Пересечение луча и треугольника

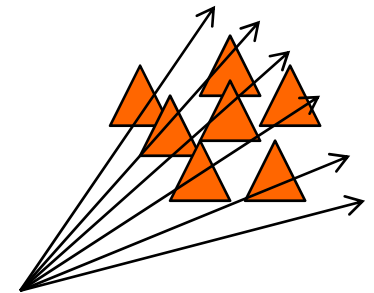
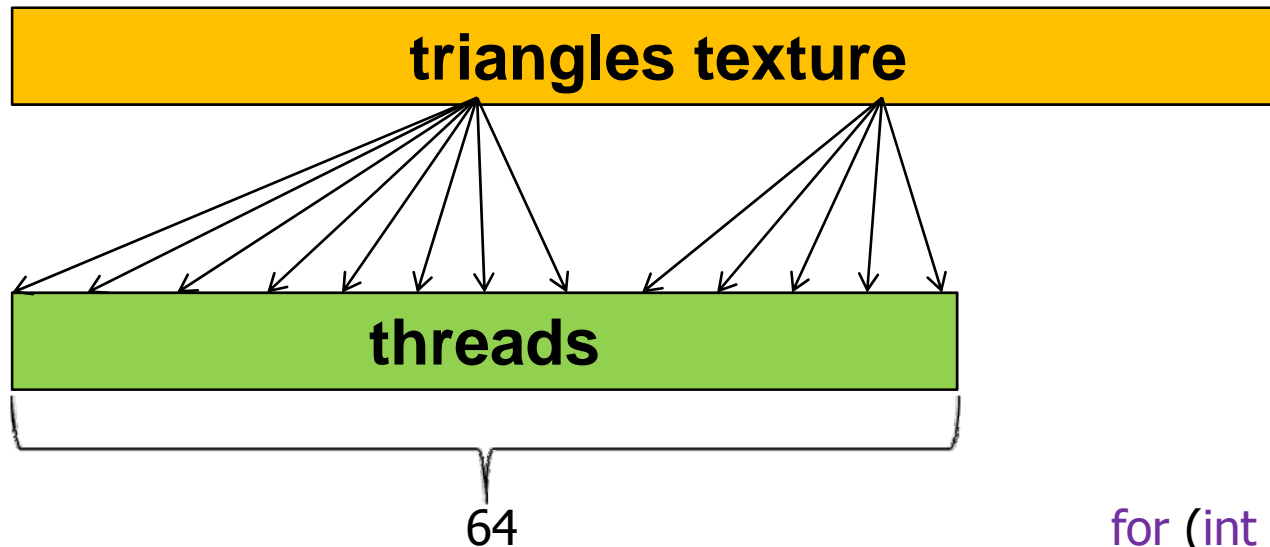
⌘ Свой луч, свой треугольник



```
for (int i=0;i<triNum;i++)  
{  
    (A,B,C) = tex1Dfetch(tex,i);  
    // intersection code  
}
```

Пересечение луча и треугольника

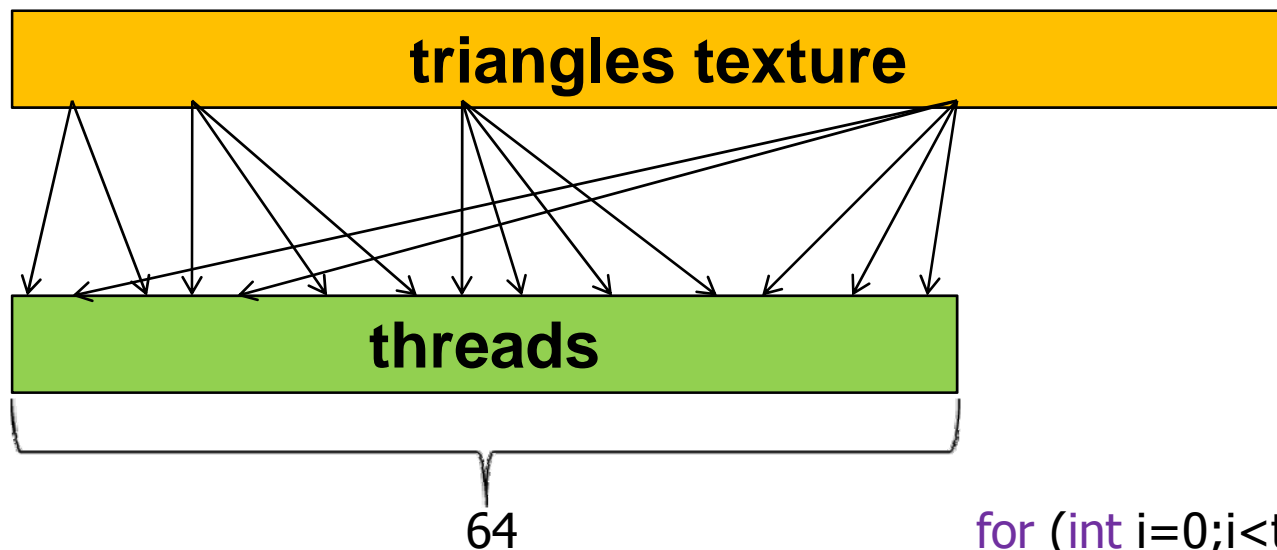
⌘ Свой луч, свой треугольник



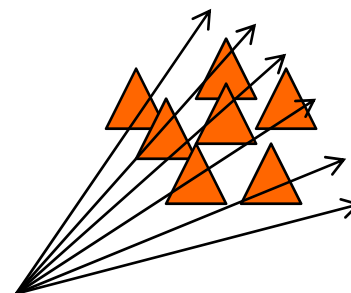
```
for (int i=0;i<triNum;i++)  
{  
    (A,B,C) = tex1Dfetch(tex,i);  
    // intersection code  
}
```

Пересечение луча и треугольника

⌘ Свой луч, свой треугольник



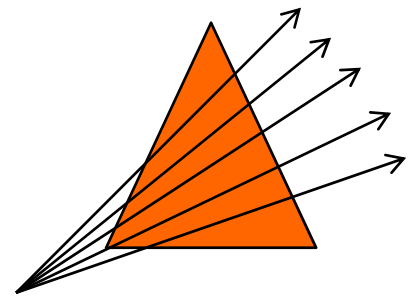
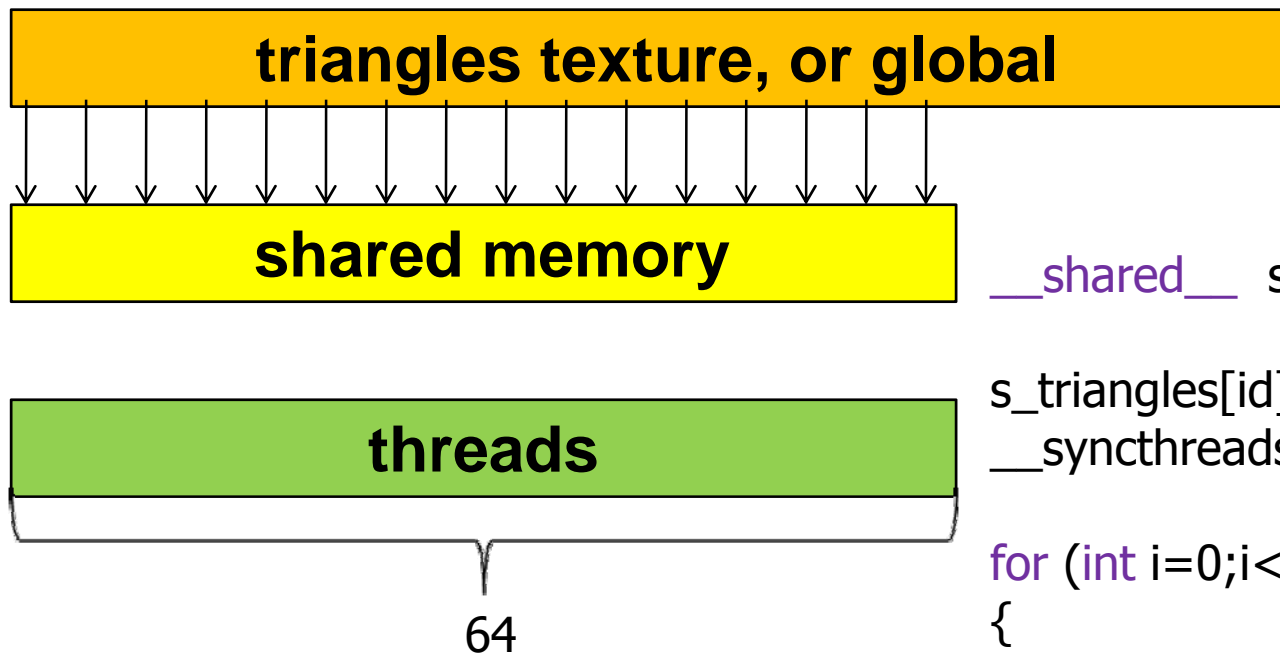
Ядро занимает 32 регистра



```
for (int i=0;i<triNum;i++)  
{  
    (A,B,C) = tex1Dfetch(tex, i+offset);  
    // intersection code  
}
```


Пересечение луча и треугольника

⌘ Свой луч, общий треугольник



```
__shared__ s_triangles[64];
```

```
s_triangles[id] = g_triangles[id];  
__syncthreads();
```

```
for (int i=0;i<64;i++)  
{  
    (A,B,C) = s_triangles[i];  
    // intersection code  
}
```

Пересечение луча и треугольника

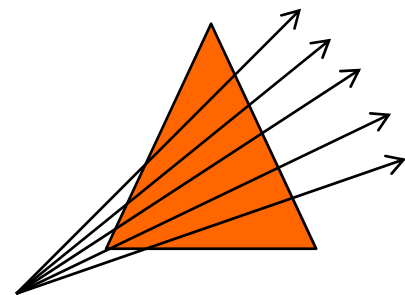
⌘ Свой луч, общий треугольник

triangles texture, or global

shared memory

threads

64



```
__shared__ s_triangles[64];
```

```
s_triangles[id] = g_triangles[id];  
__syncthreads();
```

```
for (int i=0;i<64;i++)  
{  
    (A,B,C) = s_triangles[i];  
    // intersection code  
}
```

Пересечение луча и треугольника

⌘ Свой луч, общий треугольник

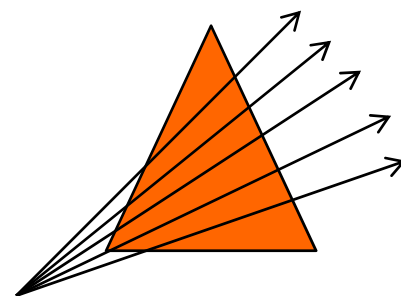
triangles texture, or global

shared memory

threads

64

Ядро занимает 20 регистров



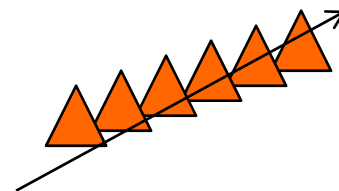
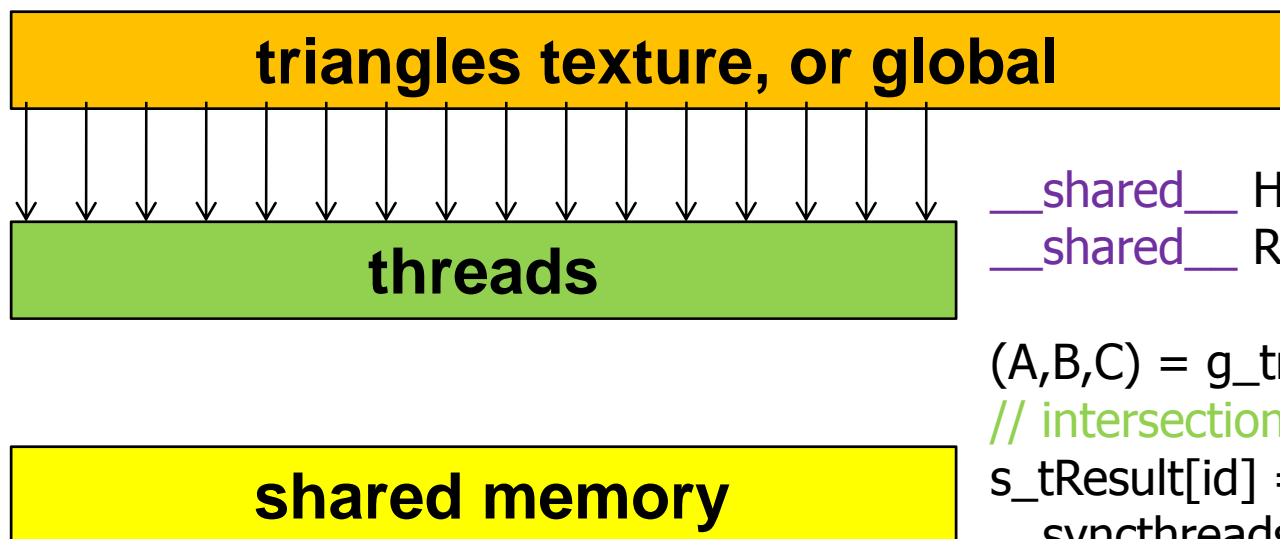
```
__shared__ s_triangles[64];
```

```
s_triangles[id] = g_triangles[id];  
__syncthreads();
```

```
for (int i=0;i<64;i++)  
{  
    (A,B,C) = s_triangles[i];  
    // intersection code  
}
```

Пересечение луча и треугольника

⌘ Общий луч, свой треугольник



```
__shared__ Hit s_tResult[64];  
__shared__ Ray ray;
```

```
(A,B,C) = g_triangles[id];
```

```
// intersection code
```

```
s_tResult[id] = resHit;
```

```
__syncthreads();
```

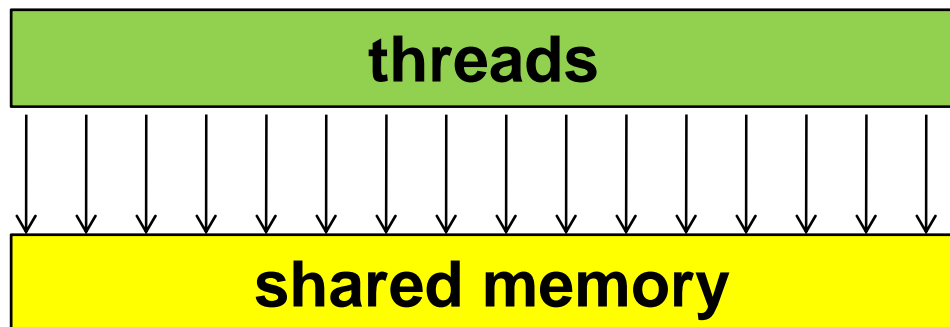
```
ReduceMin(s_tResult, 64);
```

```
// s_tResult[0] – ближайшее  
пересечение
```

Пересечение луча и треугольника

⌘ Общий луч, свой треугольник

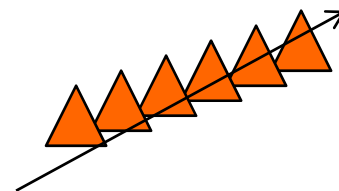
triangles texture, or global



```
__shared__ Hit s_tResult[64];  
__shared__ Ray ray;
```

```
(A,B,C) = g_triangles[id];  
// intersection code  
s_tResult[id] = resHit;  
__syncthreads();
```

```
ReduceMin(s_tResult, 64);  
// s_tResult[0] – ближайшее  
пересечение
```



Пересечение луча и треугольника

⌘ Общий луч, свой треугольник

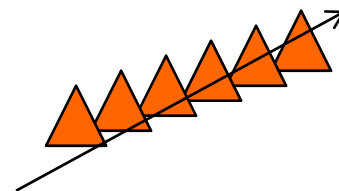
triangles texture, or global

threads

shared memory

ReduceMin

Ядро занимает 21 регистр



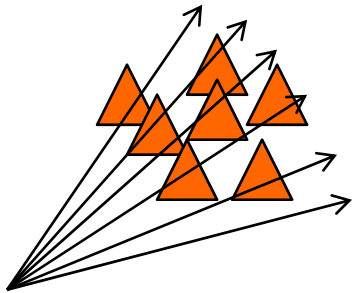
```
__shared__ Hit s_tResult[64];  
__shared__ Ray ray;
```

```
(A,B,C) = g_triangles[id];  
// intersection code  
s_tResult[id] = resHit;  
__syncthreads();
```

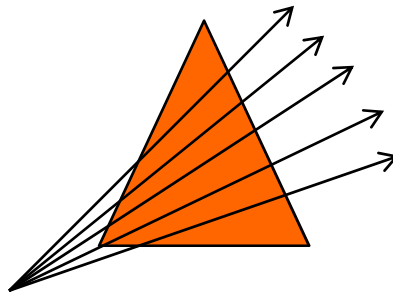
```
ReduceMin(s_tResult, 64);  
// s_tResult[0] – ближайшее  
пересечение
```

Пересечение луча и треугольника

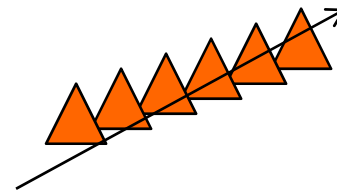
- ⌘ Организация сетки - блоки 8x8
- ⌘ Что общее для нитей блока?
 1. Свой луч, свой треугольник (100%)
 2. Свой луч, общий треугольник (130%)
 3. Общий луч, свой треугольник (40-50%)



32 регистра



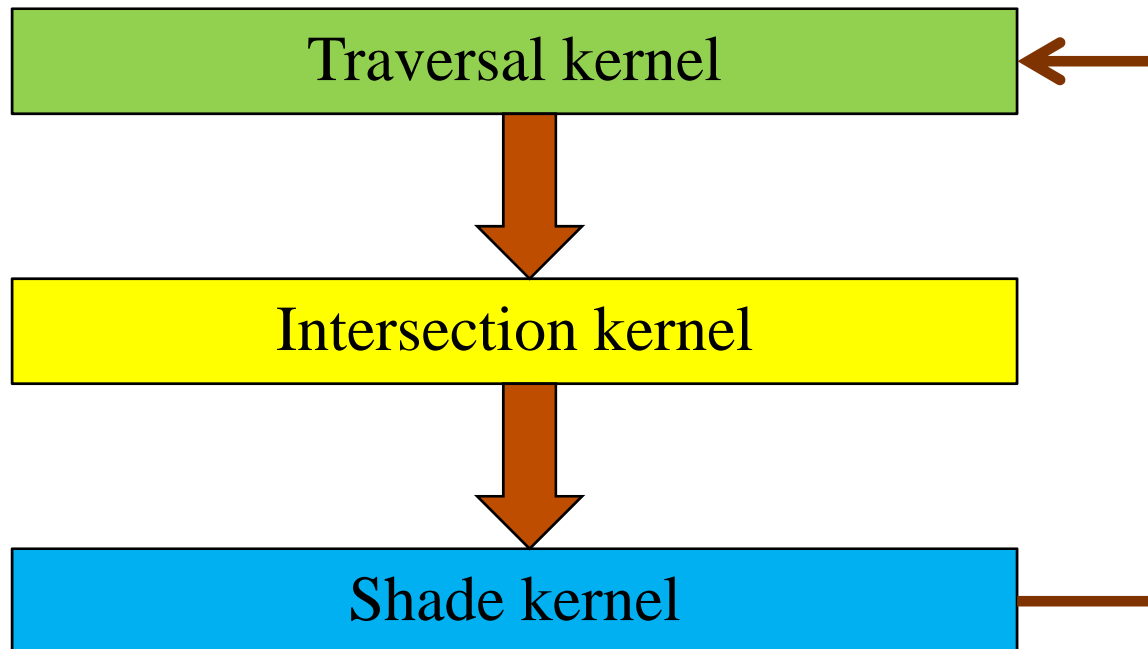
20 регистров



21 регистр

Архитектура рейтрейсера

- ⌘ Ядро пересечений – 32 регистра
- ⌘ Нужно разбить алгоритм трассировки на несколько ядер



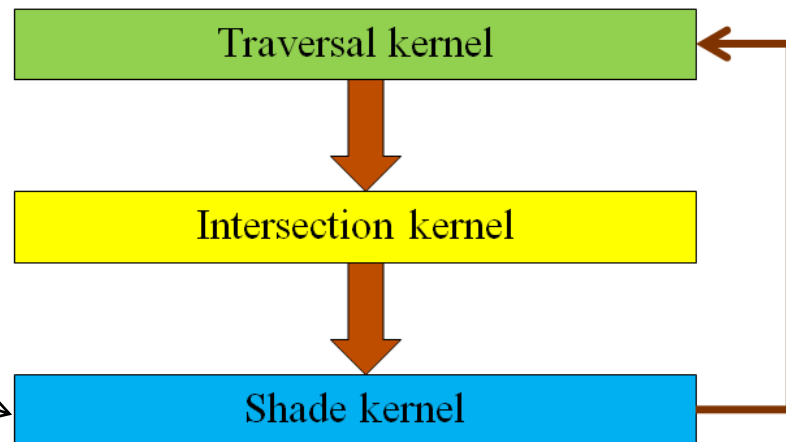
Архитектура рейтрейсера

⌘ Traversal kernel – блоки 16x4

⌘ Как хранить геометрию?

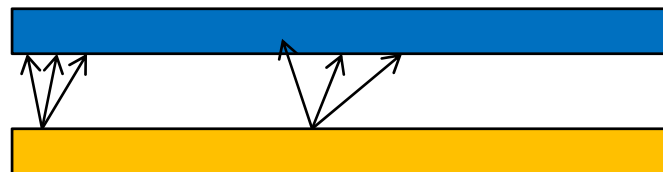
```
struct Vertex
{
    float3 pos[3];
    float3 norm[3];
    float2 texCoord;
    uint materialIndex;
};
```

```
struct Triangle
{
    uint v[3];
};
```



Vertex array:

Index array:



Архитектура рейтрейсера

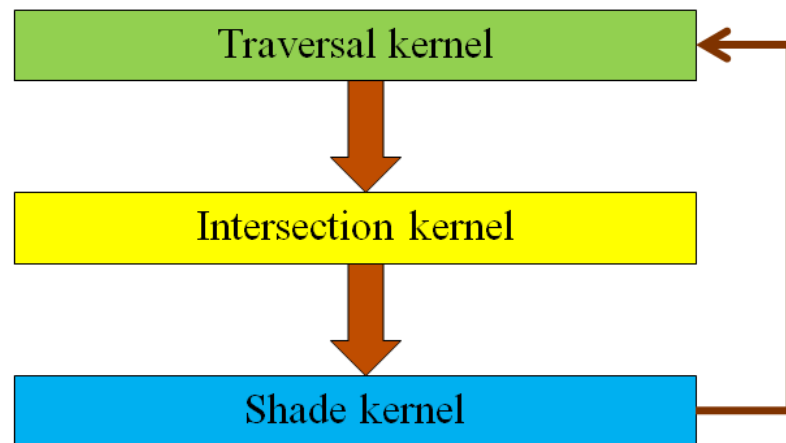
⌘ Traversal kernel – блоки 16x4

⌘ Дублирование геометрии

```
struct Triangle
{
    vec3f v[3];
    unsigned int selfIndex;
}; // 40
```

```
struct Sphere
{
    PackedSphere3f sph;
    unsigned int selfIndex;
    unsigned int dummy;
}; // 24
```

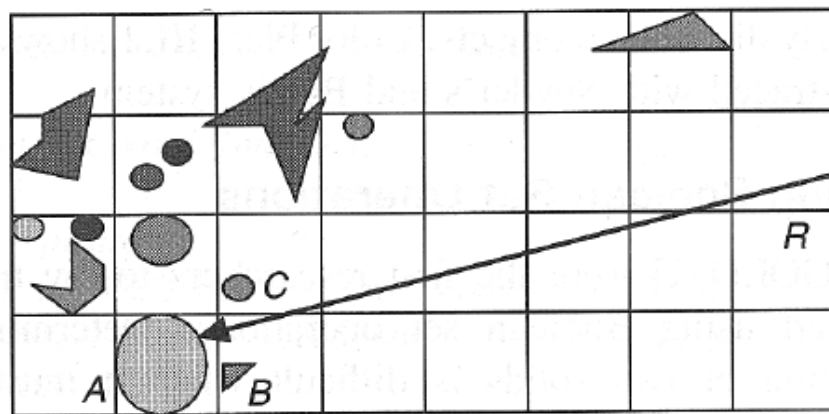
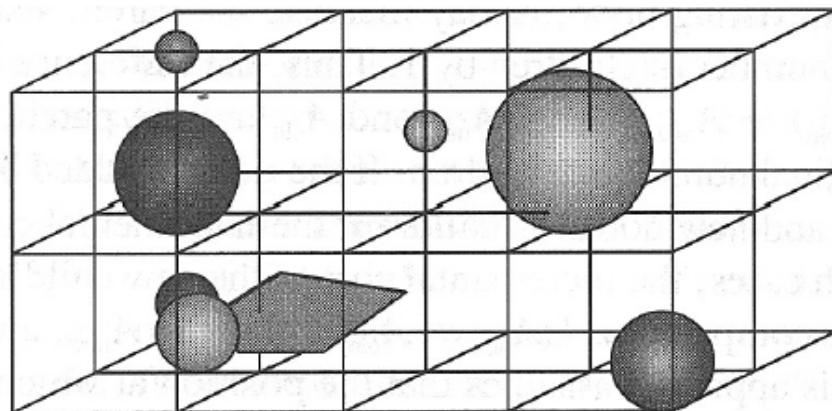
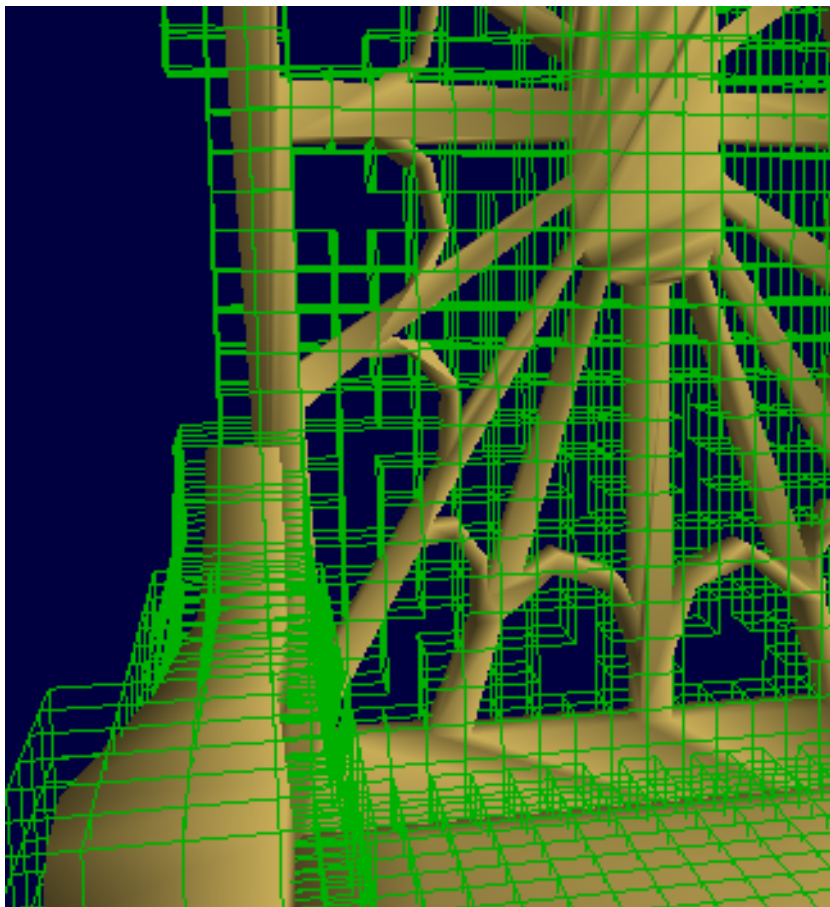
// выборки по float2



```
struct Hit
{
    float t;
    unsigned int objectIdAndType;
};
```



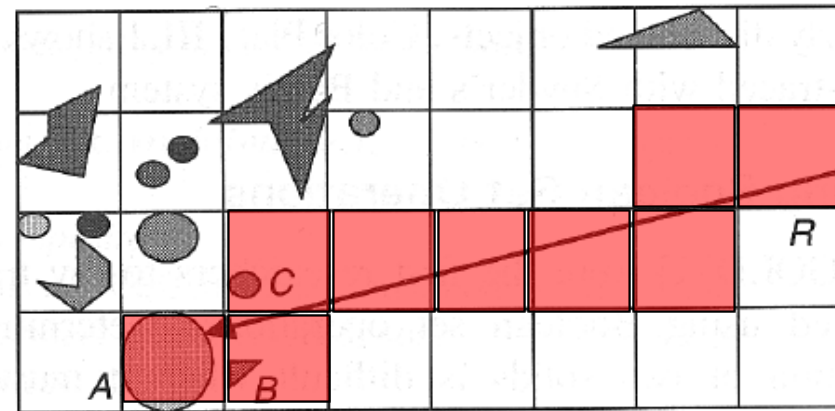
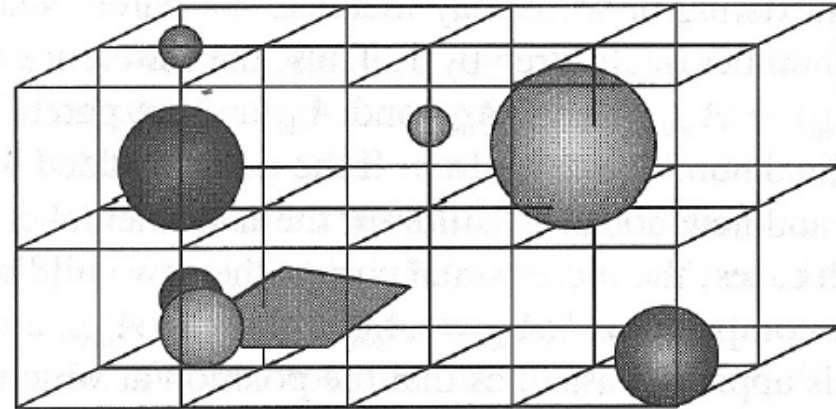
Регулярная сетка



Регулярная сетка

⌘ Регулярная сетка

```
if (tMaxX <= tMaxY && tMaxX <= tMaxZ)
{
    tMaxX += tDeltaX;
    x += stepX;
}
else if (tMaxY <= tMaxZ && tMaxY <= tMaxX)
{
    tMaxY += tDeltaY;
    y += stepY;
}
else
{
    tMaxZ += tDeltaZ;
    z += stepZ;
}
```



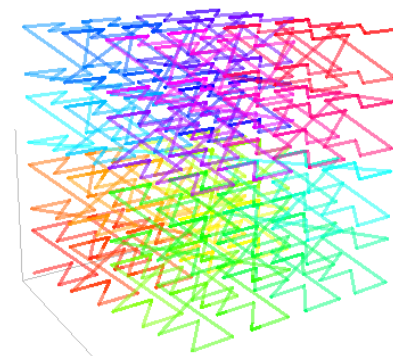
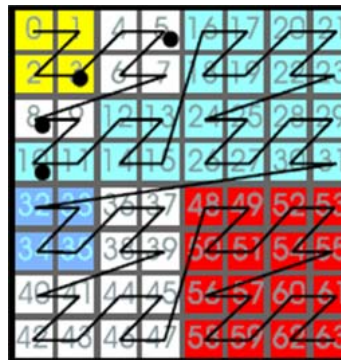
Регулярная сетка и CUDA

⌘ Преимущества

- ☑ Занято как минимум $12 + 2$ регистров
- ☑ Z-ordering для 3D массива, кэш промахи↓

⌘ Недостатки!

- ☑ Жуткая нагрузка на текстурные блоки, латентность памяти не покрывается вычислениями



Регулярная сетка

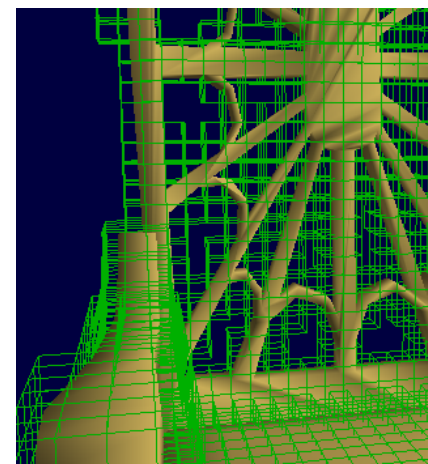
⌘ Преимущества

- ☑ Просто и быстро строится
- ☑ Простой алгоритм траверса

⌘ Недостатки

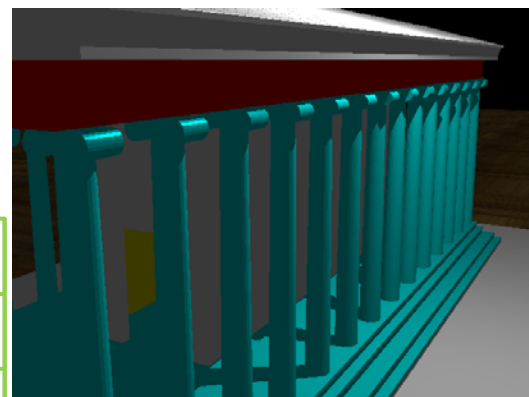
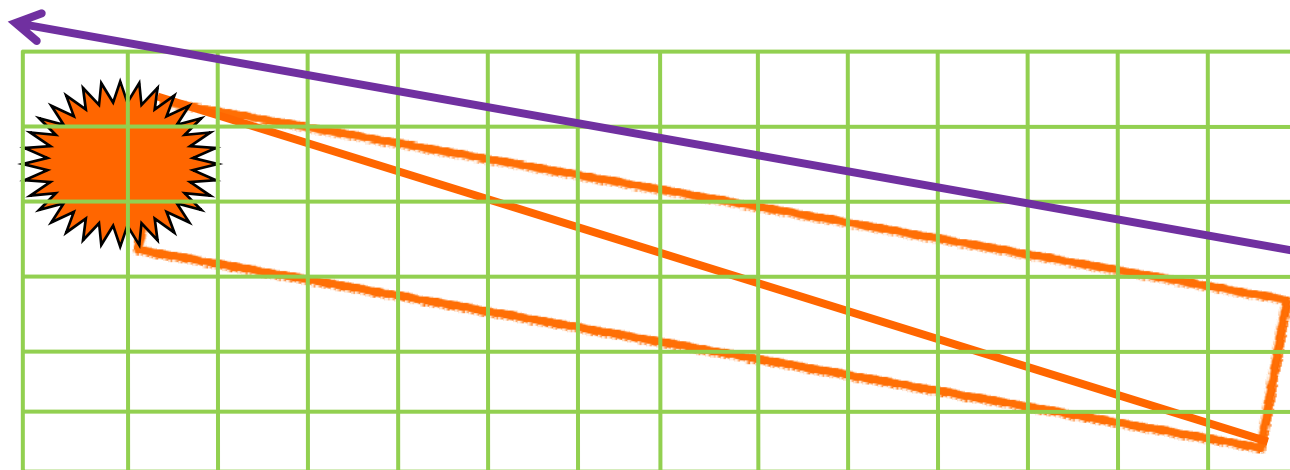
- ☑ Плохо справляется с пустым пространством
- ☑ Требует много памяти
- ☑ Много повторных пересечений – **отвратительно** разбивает геометрию

⌘ Только для небольших сцен (1-50К)



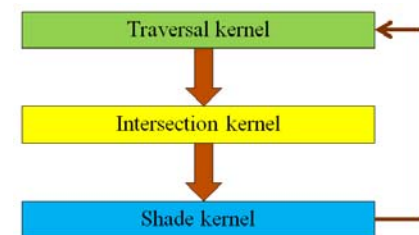
Регулярная сетка

⌘ Почему сетка плохо разбивает геометрию?



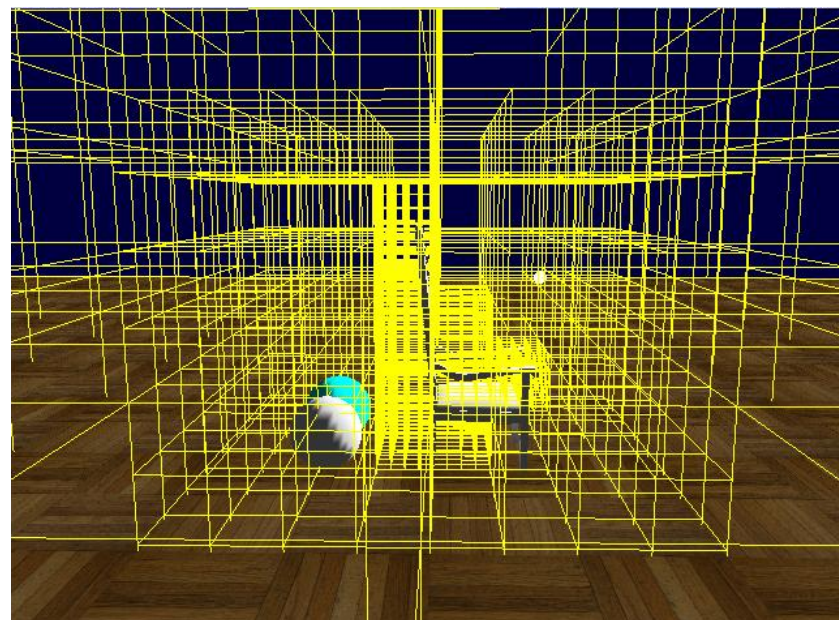
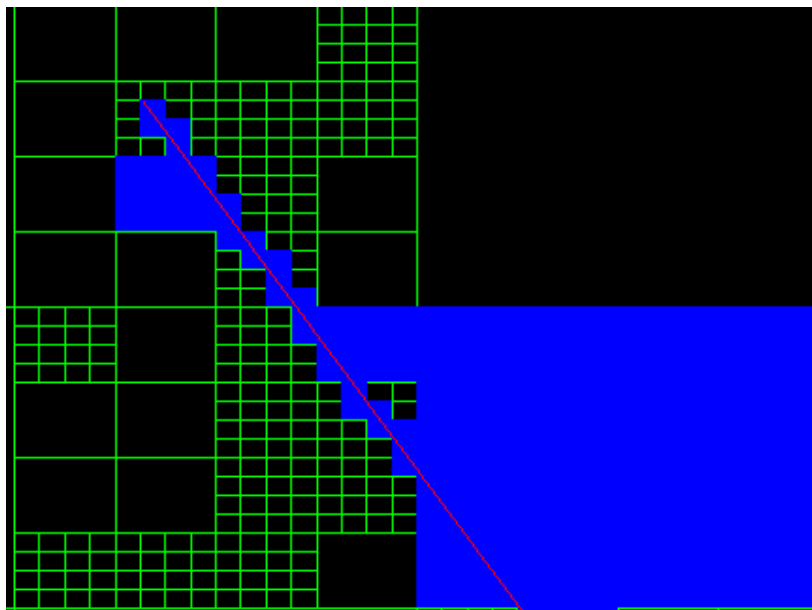
⌘ Перебрали 15 вокселей

⌘ 7 раз посчитали пересечение с одним и тем же треугольником!



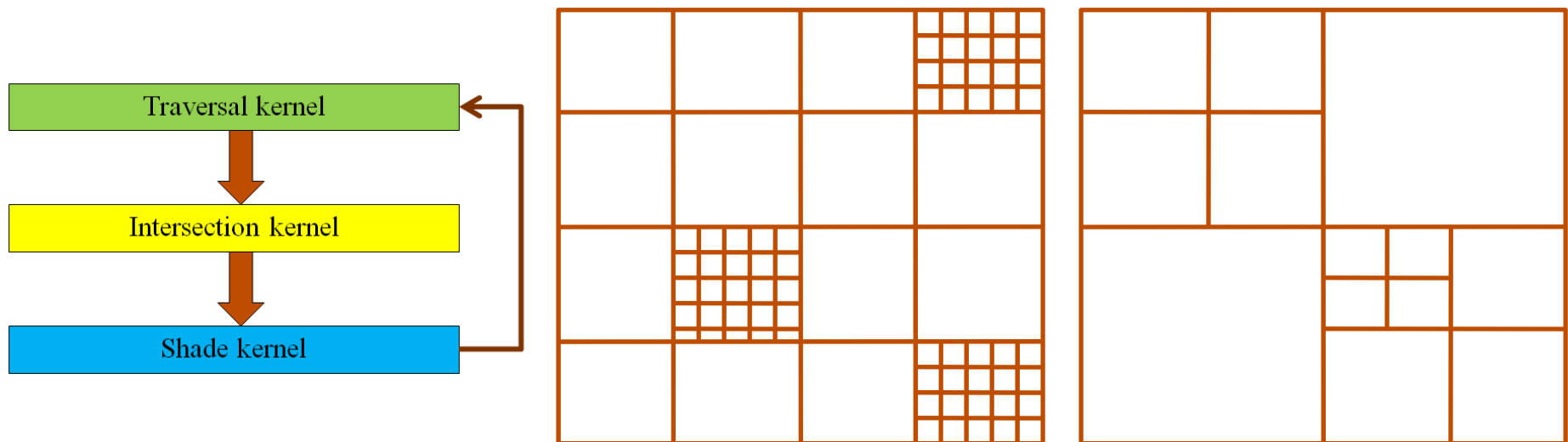
Иерархическая сетка

- ⌘ Небольшое число вокселей
- ⌘ Рекурсивно разбиваем воксели в местах с плотной геометрией



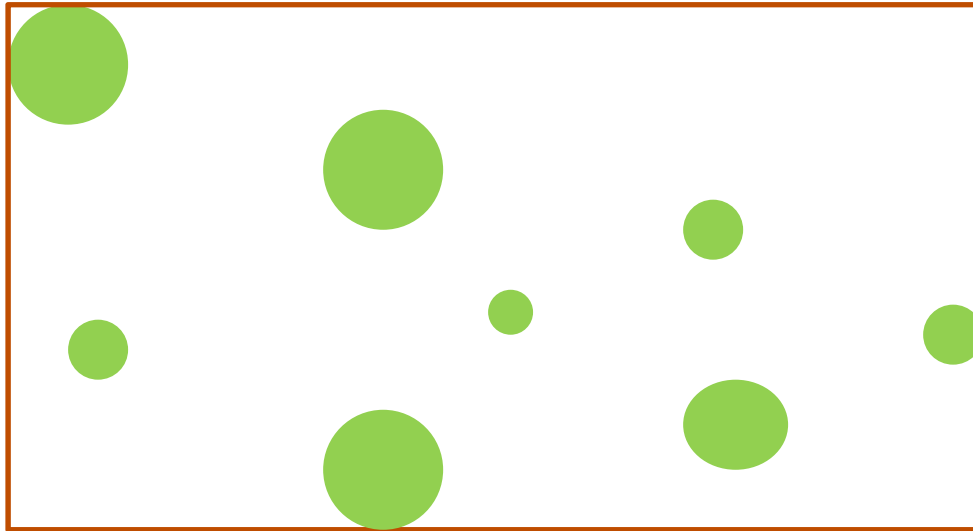
Что дает иерархическая сетка?

- + Решает проблему чайника на стадионе
- Переход между узлами вычислительно сложен
- + 12 регистров как минимум
- Нужно устранять рекурсию



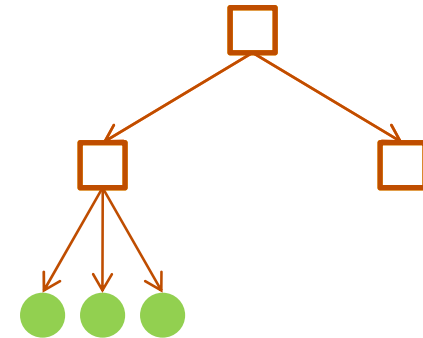
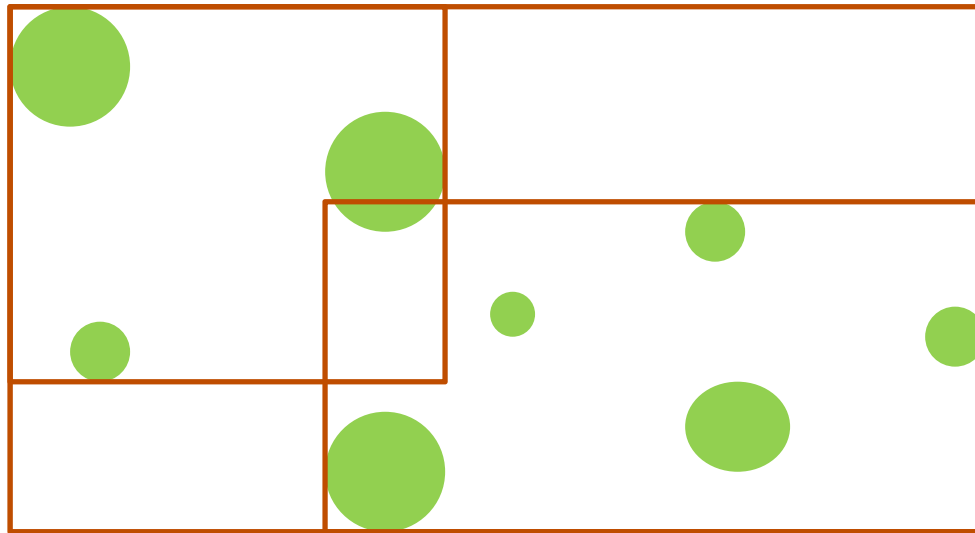
BVH деревья

⌘ Bounding Volume Hierarchy



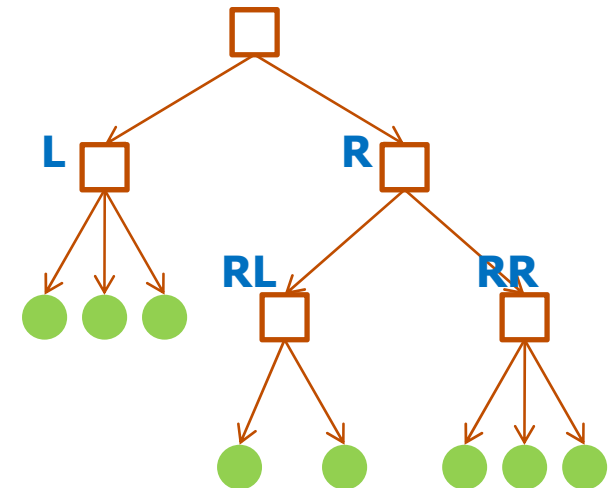
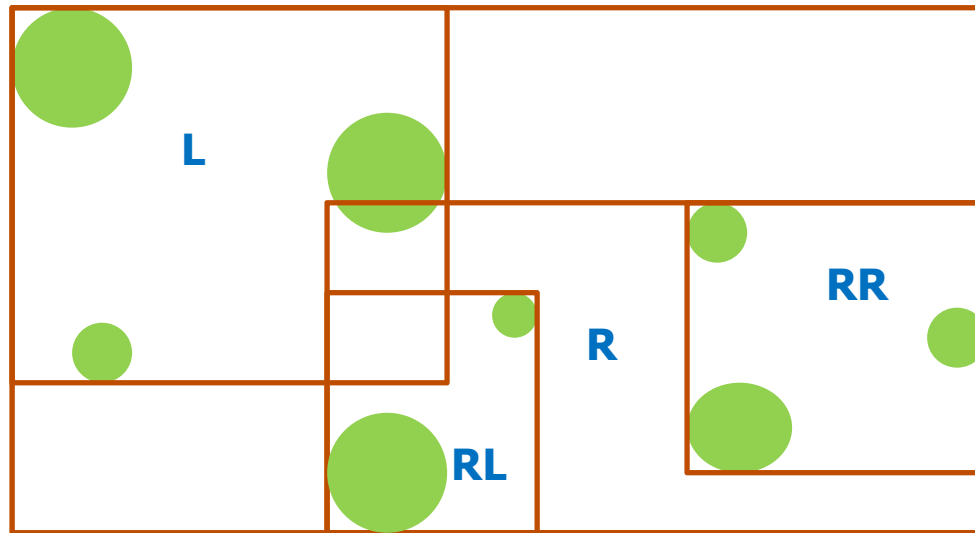
BVH деревья

⌘ Bounding Volume Hierarchy



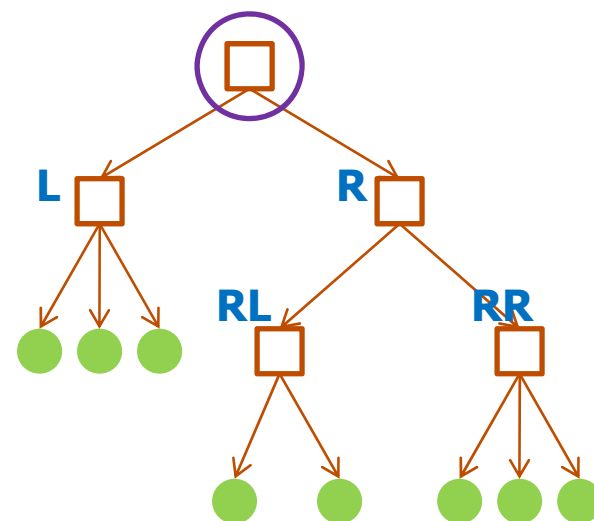
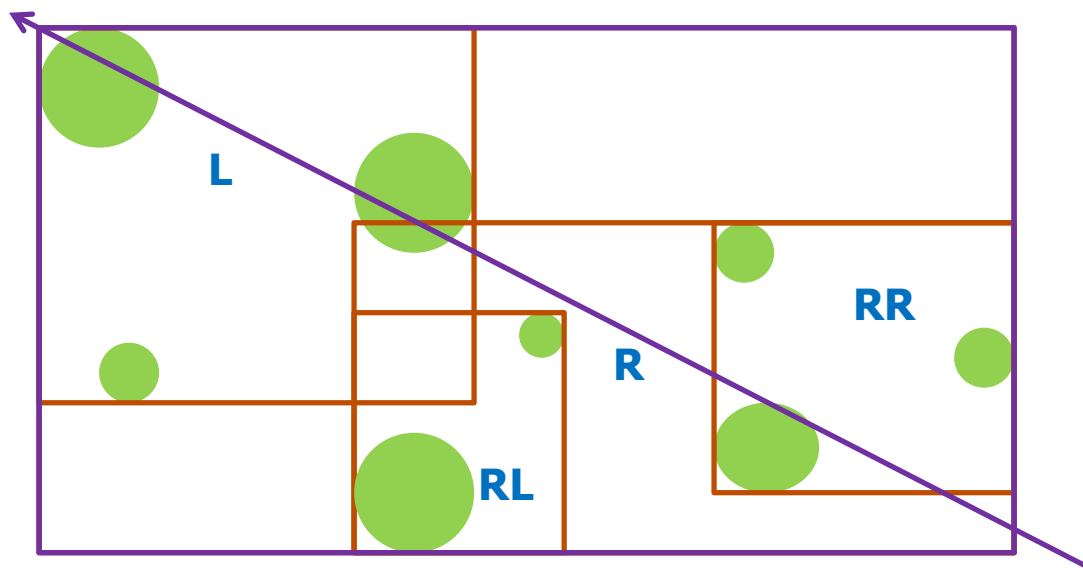
BVH деревья

⌘ Bounding Volume Hierarchy



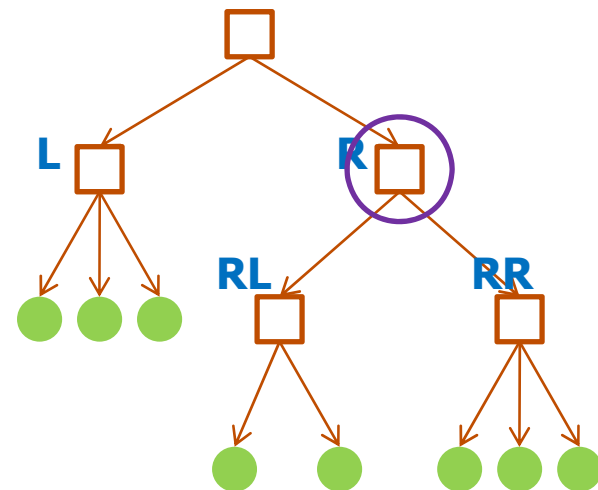
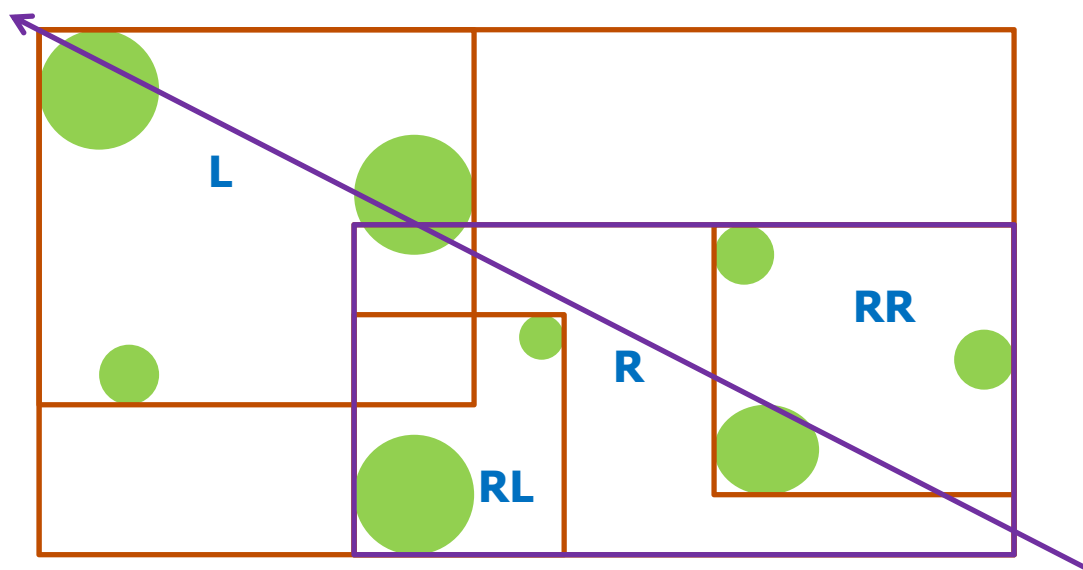
BVH деревья

✂ Траверс на CPU



BVH деревья

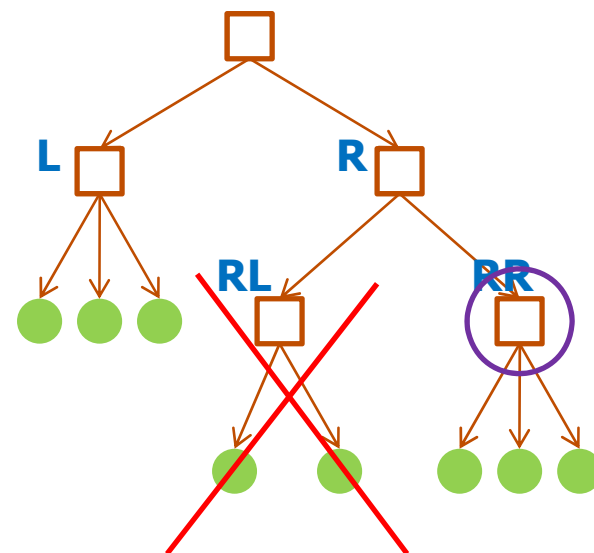
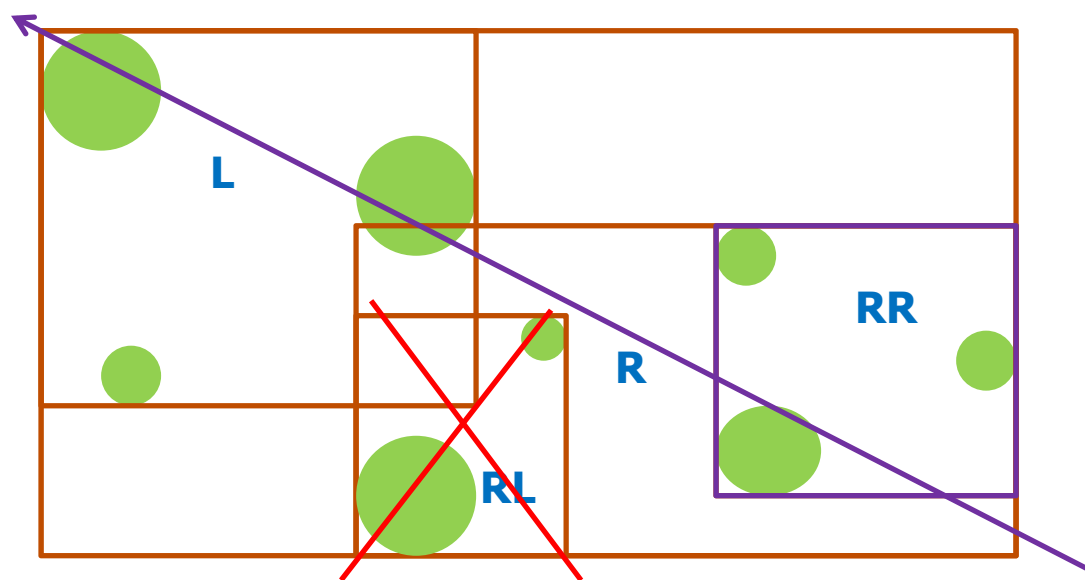
✂ Траверс на CPU



Стек: L

ВУН деревья

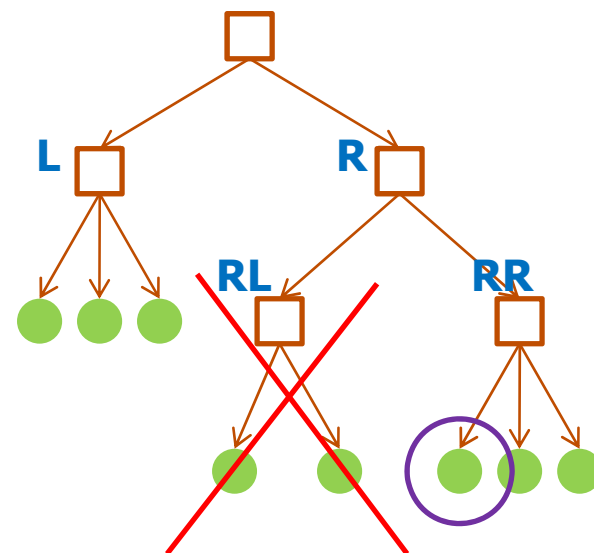
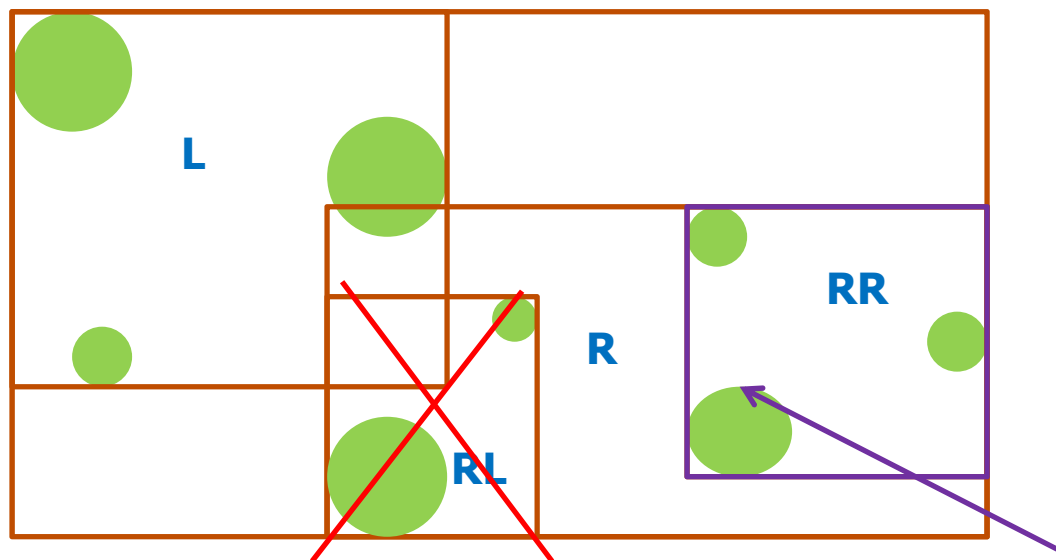
✂ Траверс на CPU



Стек: L

ВУН деревья

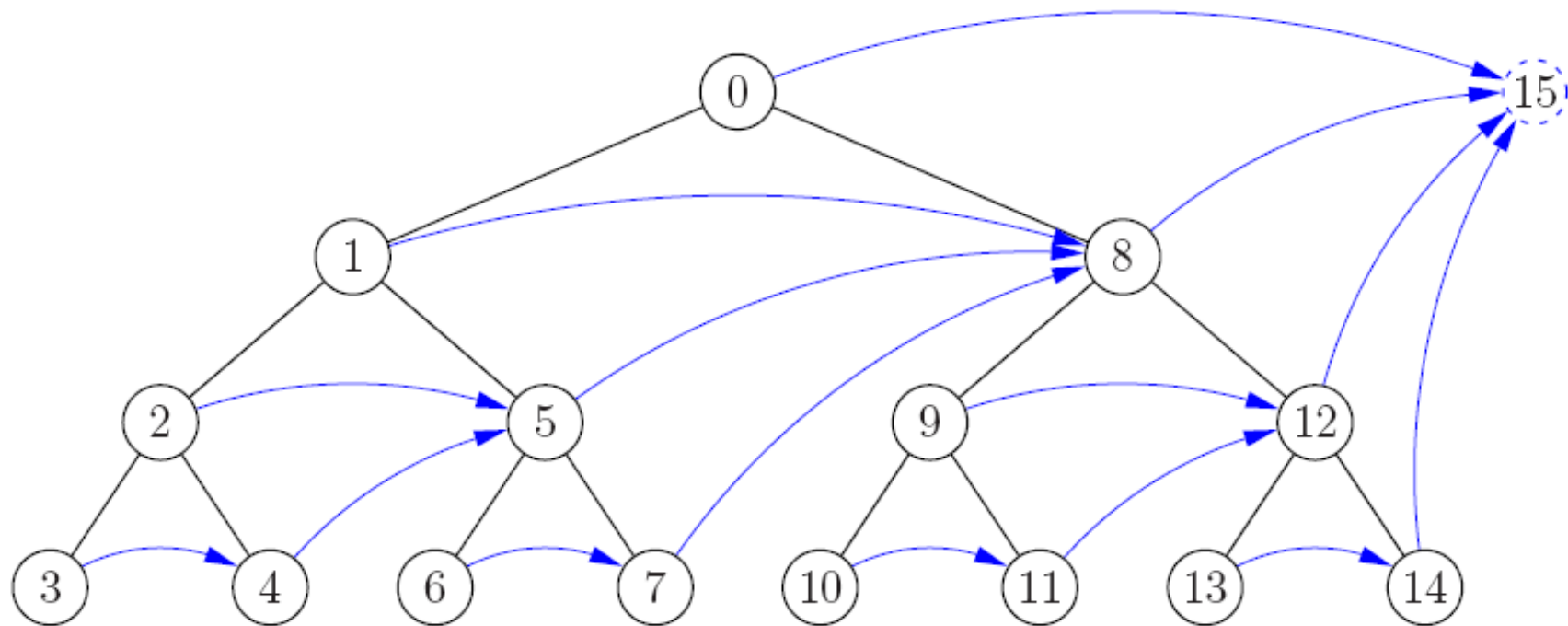
✂ Траверс на CPU



Стек: L

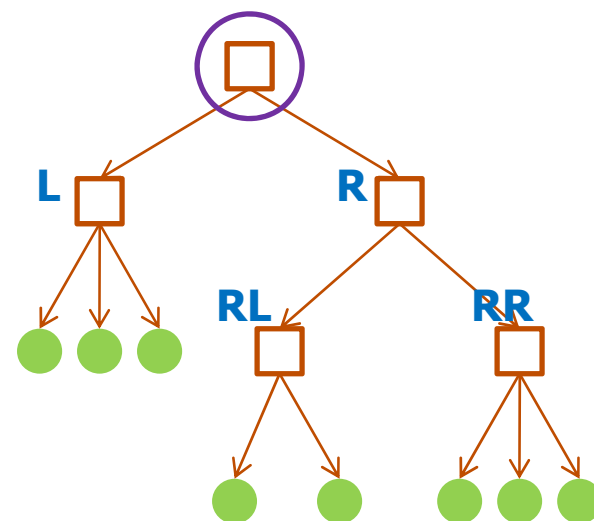
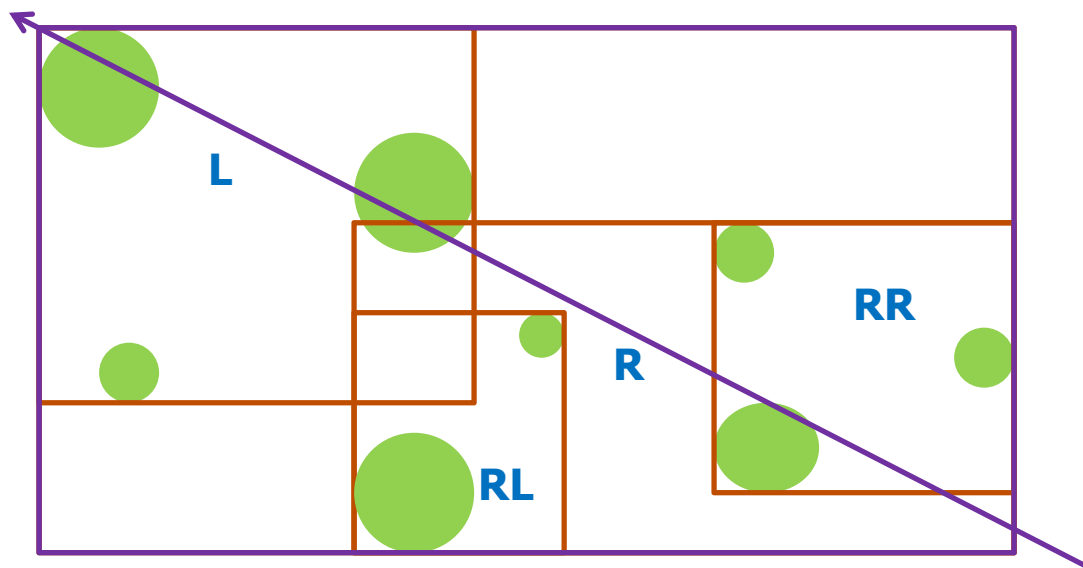
ВУН деревья

⌘ Траверс на GPU



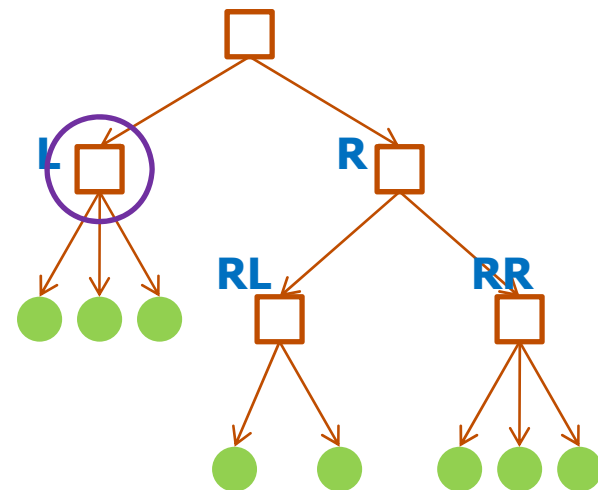
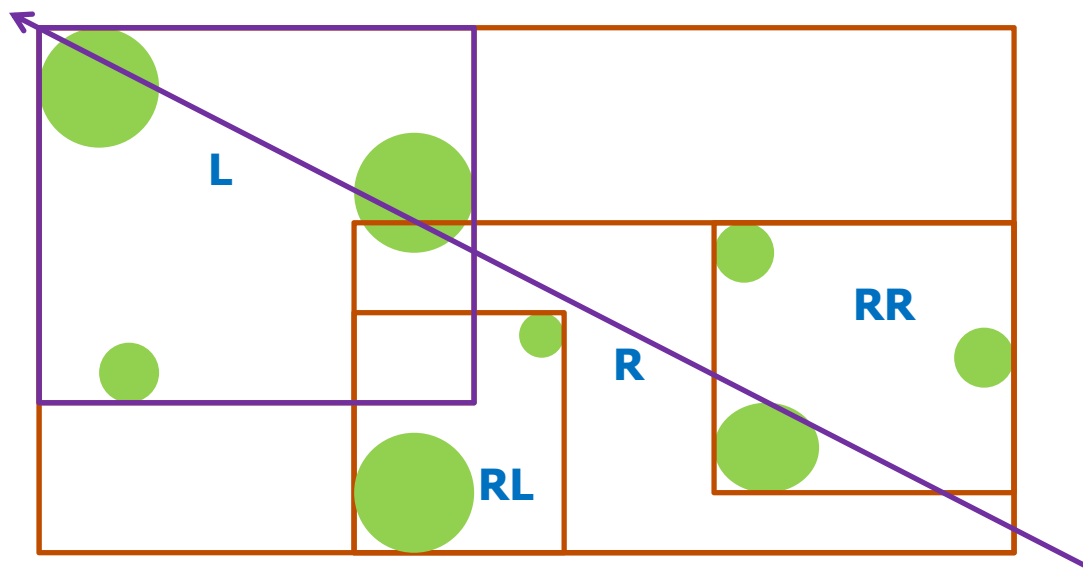
BVH деревья

✂ Траверс на GPU



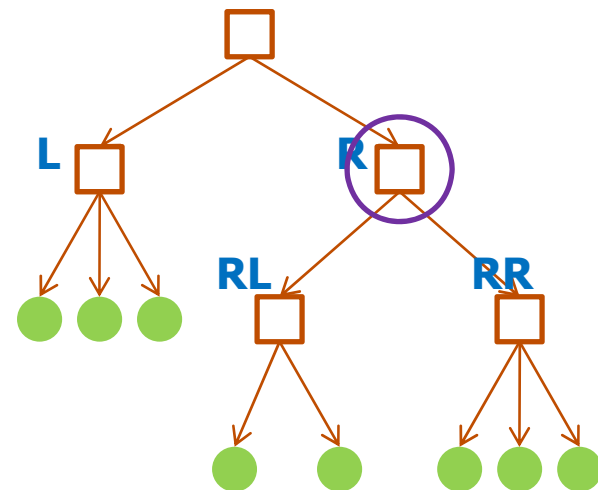
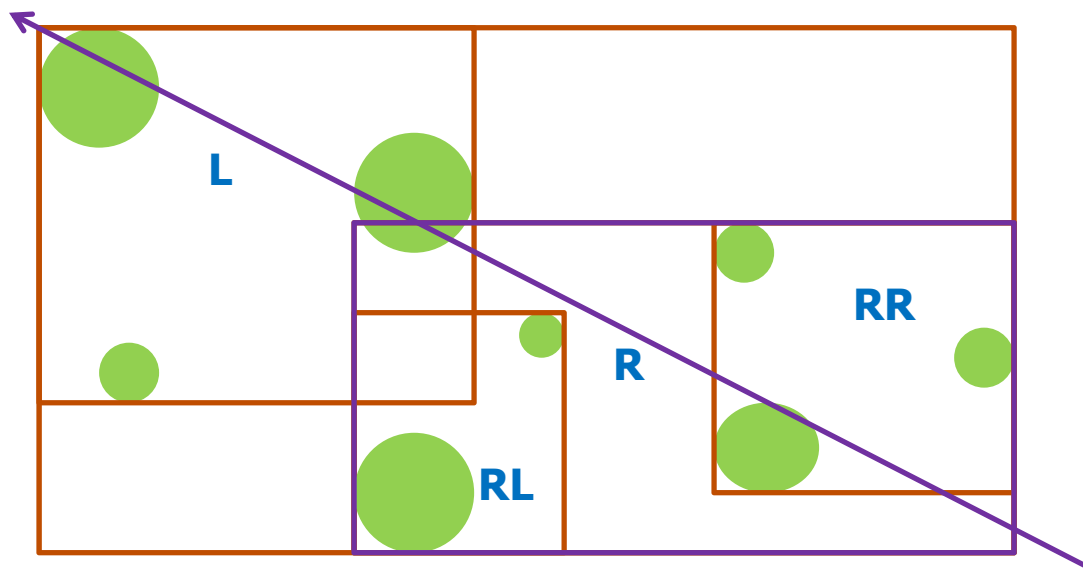
BVH деревья

⌘ Траверс на GPU



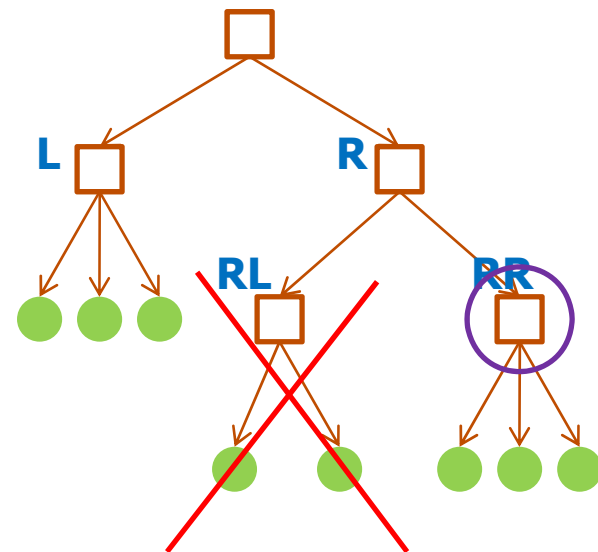
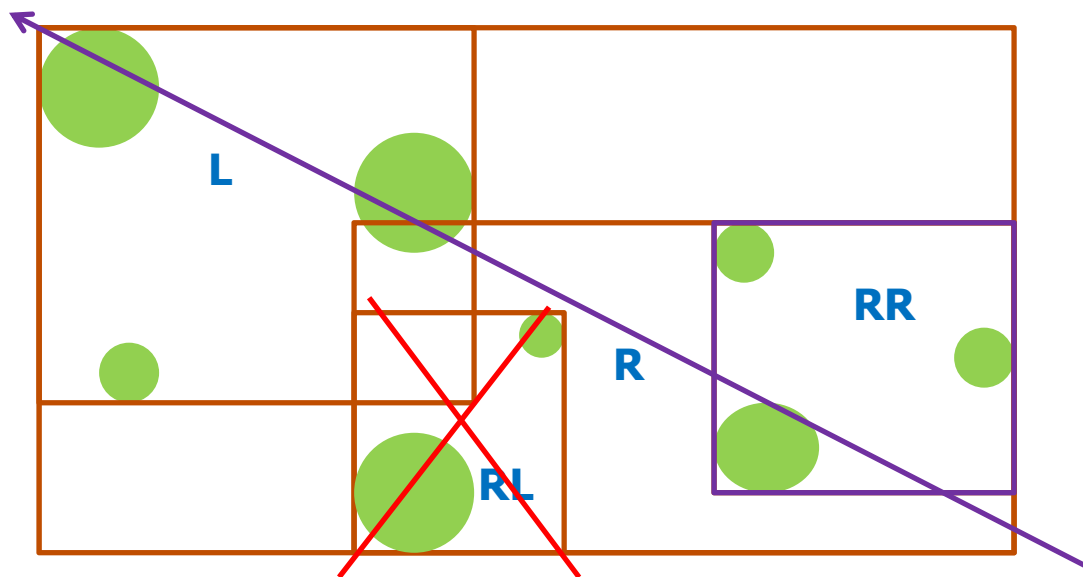
BVH деревья

✂ Траверс на GPU



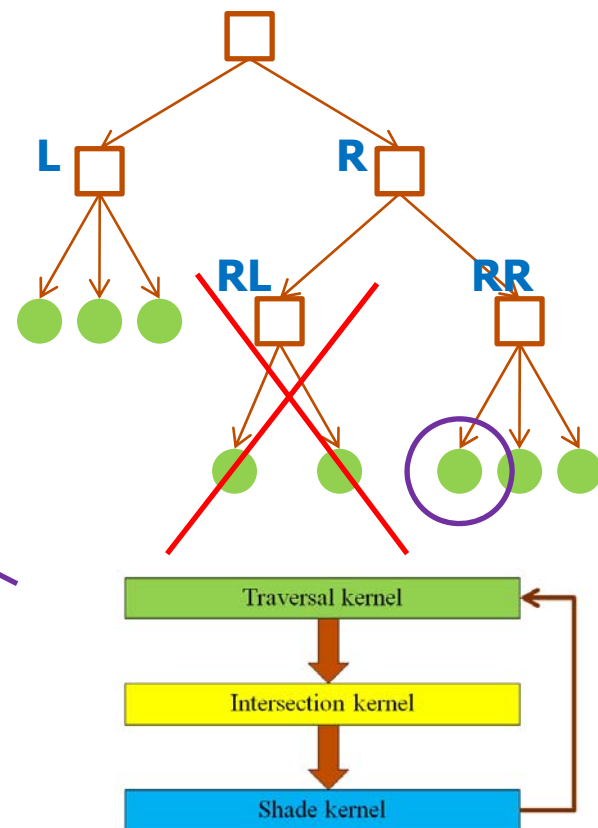
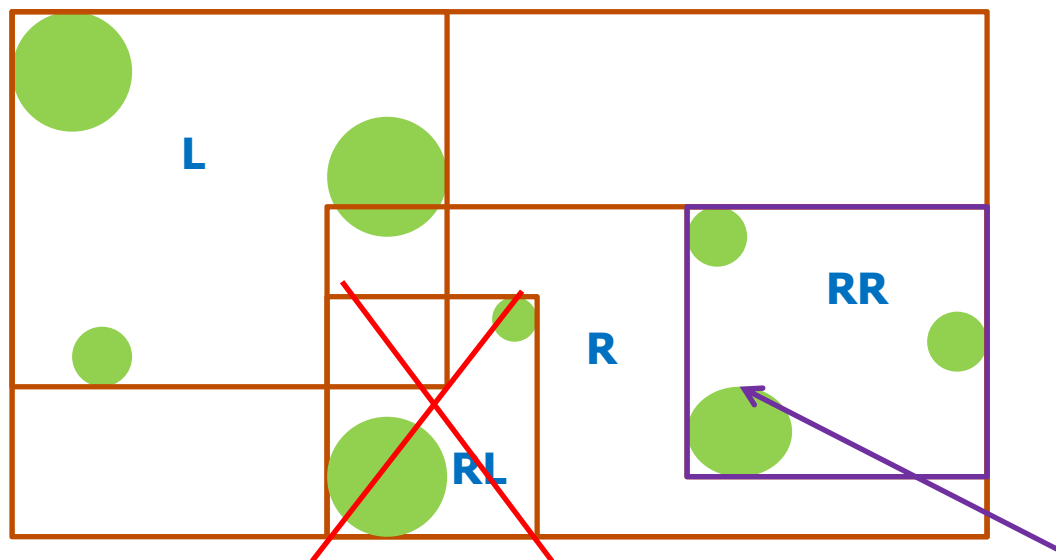
BVH деревья

✂ Траверс на GPU



BVH деревья

✂ Траверс на GPU



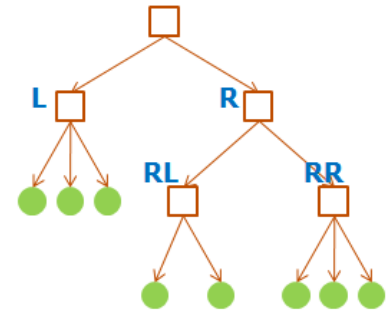
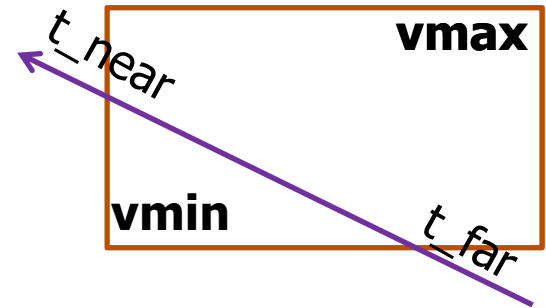
BVH деревья

⌘ Как делать на CUDA?

- ☑ Луч – 6 регистров
- ☑ Бокс – 6 регистров
- ☑ t_near , t_far , - 2
- ☑ $nodeOffset$, $leftOffset$, tid – 3

⌘ Пересечение луча с боксом

- ☑ Минимум по всем 6 плоскостям
 - ☒ $(vmin[0] + rInv.pos[0]) * rInv.dir[0];$



BVN деревья

⌘ Как делать на CUDA?

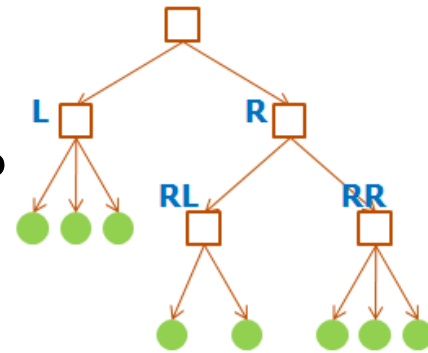
☑ 24 mad-а покрывают латентность текстурной памяти

1. Стек на локальной памяти

☑ Локальная память это не так медленно, как может показаться

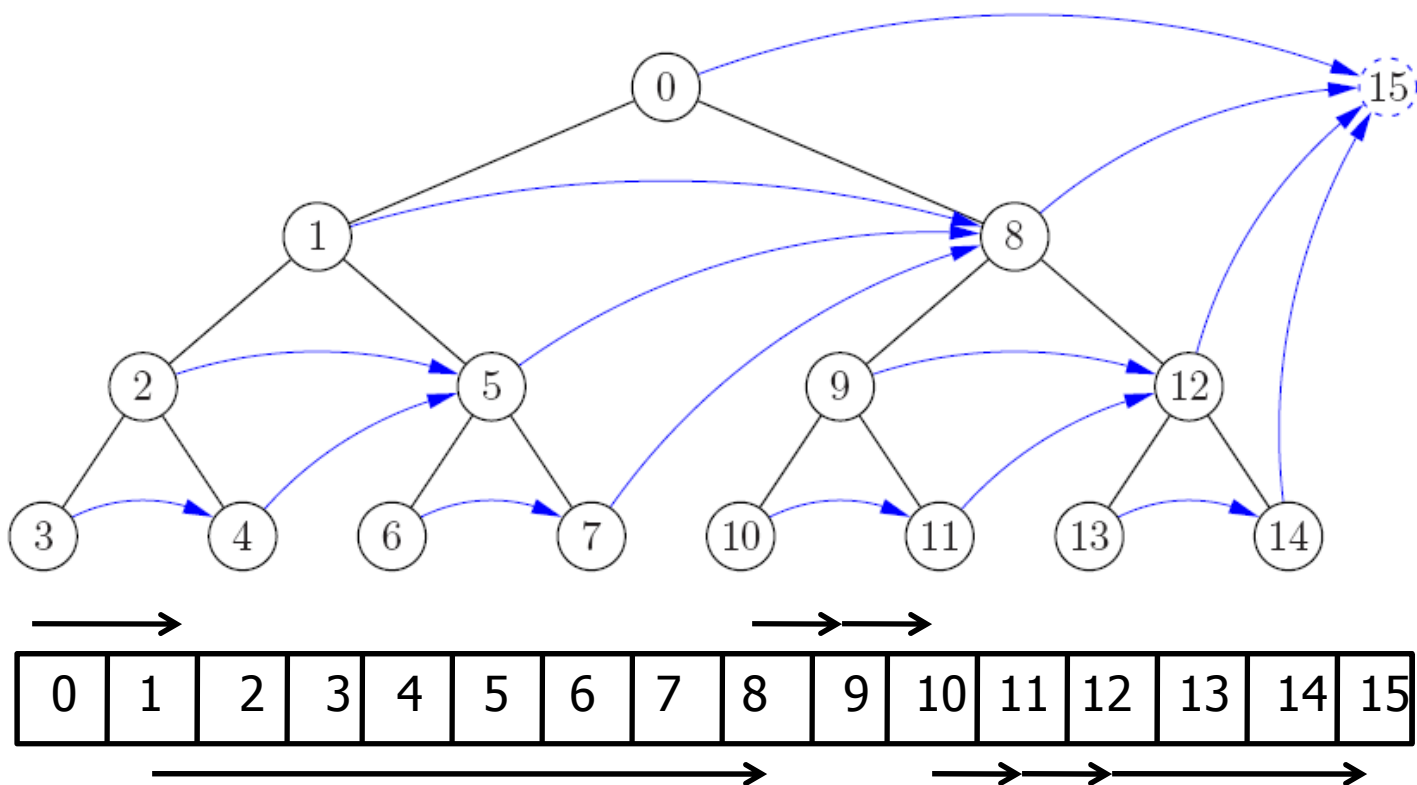
2. Бесстековый алгоритм

☑ Перебираем массив всегда строго слева направо



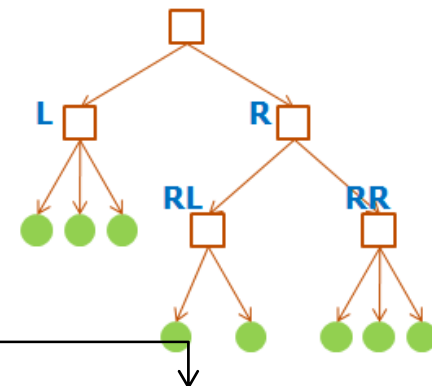
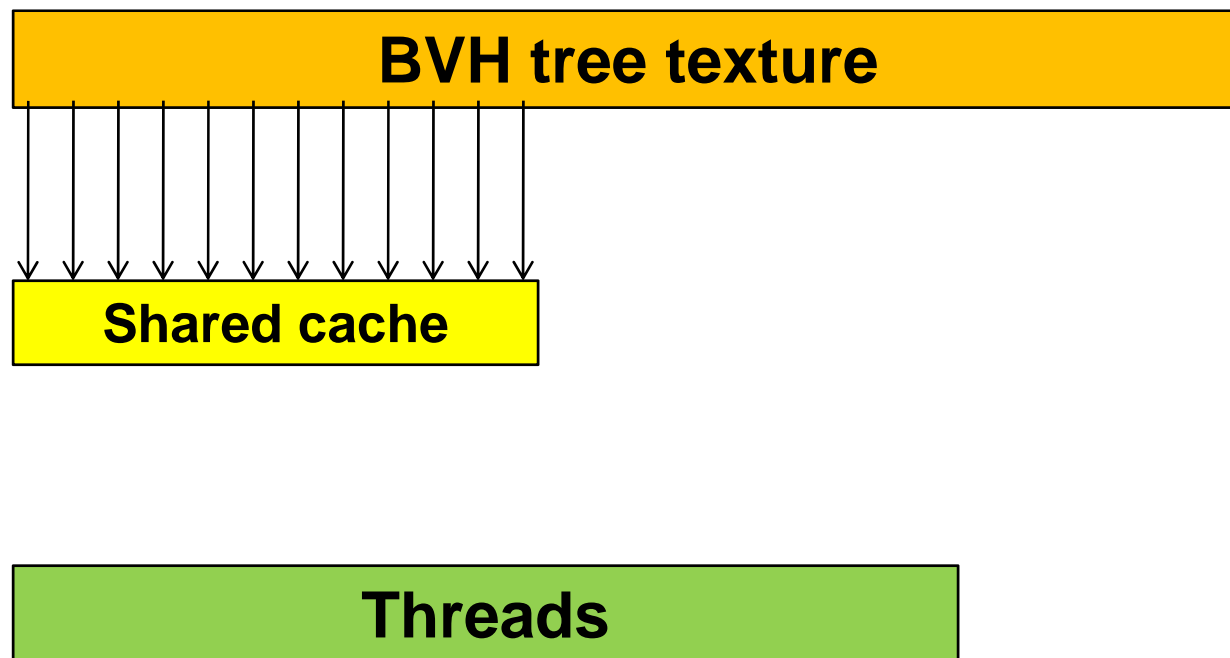
ВУН деревья

⌘ Траверс на GPU



BVH деревья

⌘ Бесстековый траверс на CUDA



FetchToSharedMem();

```
while (что-то там)
{
    траверсим дерево;
}
```

SLEEP:
__syncthreads();
ReduceMin(addr);

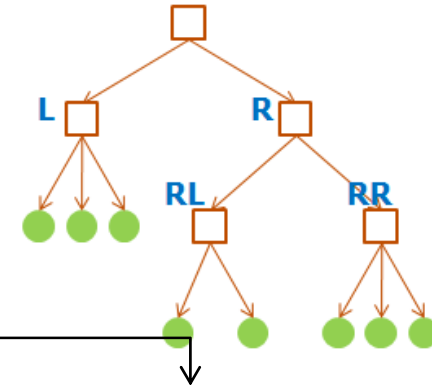
BVH деревья

⌘ Бесстековый траверс на CUDA

BVH tree texture

Shared cache

Threads



```
FetchToSharedMem();
```

```
while (что-то там)
{
    траверсим дерево;
}
```

```
SLEEP:
__syncthreads();
ReduceMin(addr);
```

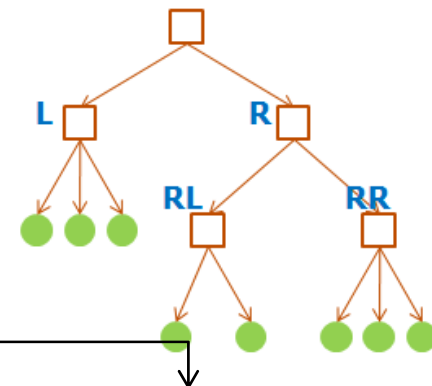
BVH деревья

⌘ Бесстековый траверс на CUDA

BVH tree texture

Shared cache

Threads



```
FetchToSharedMem();
```

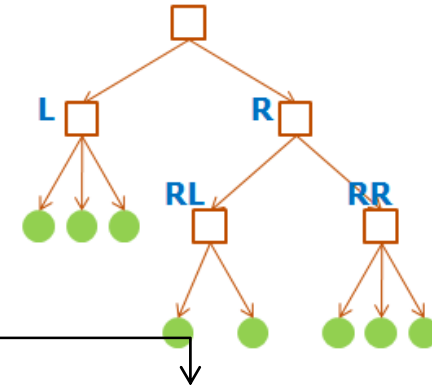
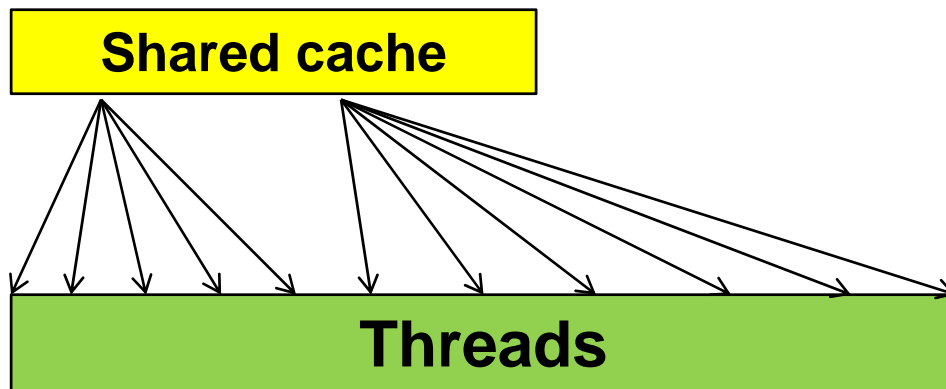
```
while (что-то там)
{
    траверсим дерево;
}
```

```
SLEEP:
__syncthreads();
ReduceMin(addr);
```


BVH деревья

⌘ Бесстековый траверс на CUDA

BVH tree texture



FetchToSharedMem();

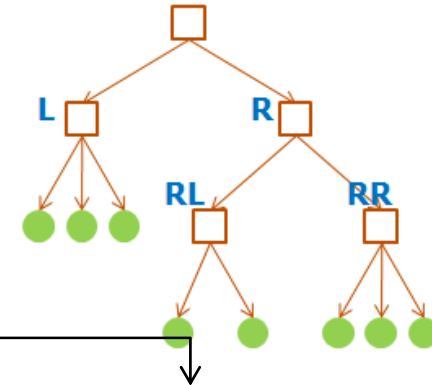
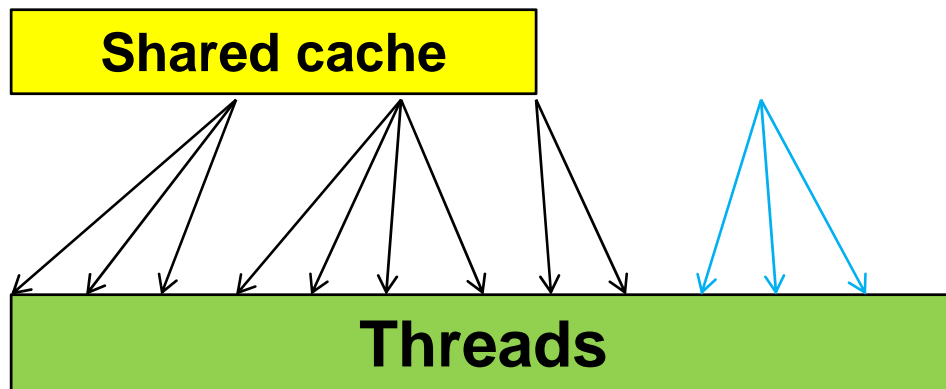
```
while (что-то там)
{
    траверсим дерево;
}
```

```
SLEEP:
__syncthreads();
ReduceMin(addr);
```

BVH деревья

⌘ Бесстековый траверс на CUDA

BVH tree texture



FetchToSharedMem();

```
while (что-то там)
{
    траверсим дерево;
}
```

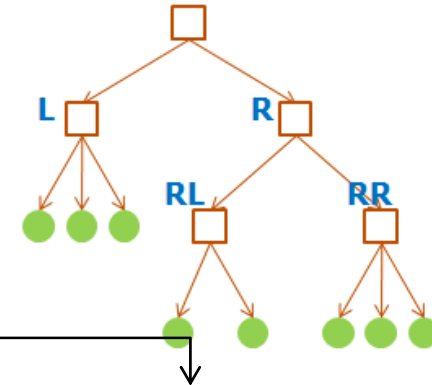
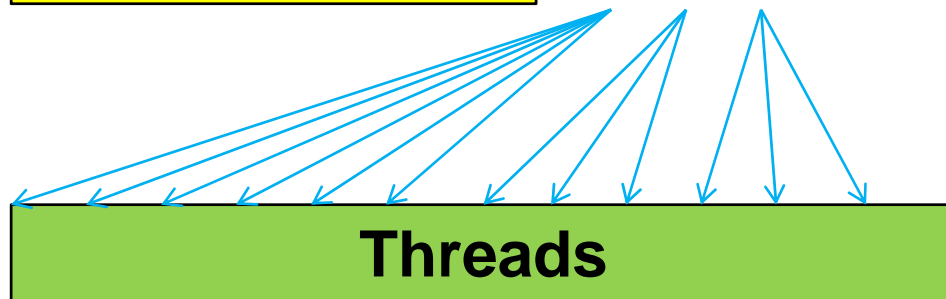
SLEEP:
__syncthreads();
ReduceMin(addr);

BVH деревья

⌘ Бесстековый траверс на CUDA

BVH tree texture

Shared cache



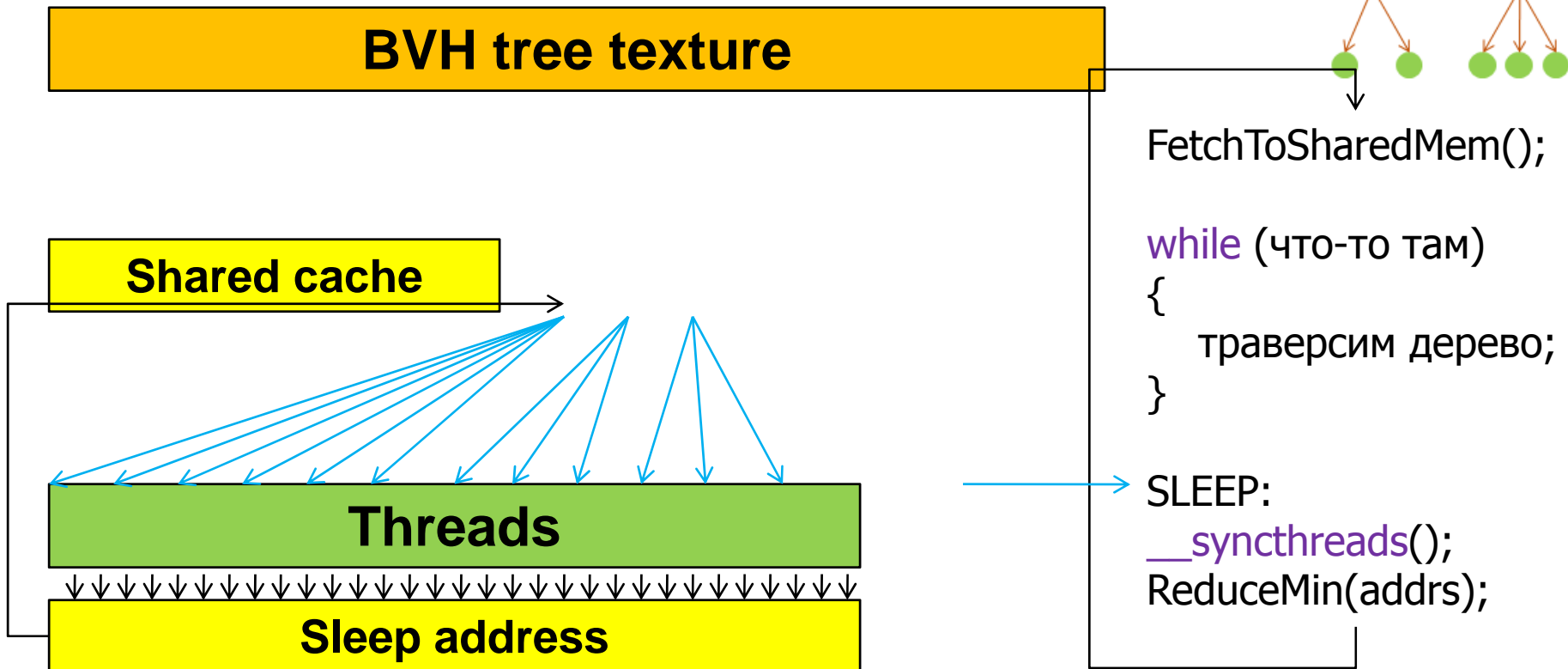
FetchToSharedMem();

```
while (что-то там)
{
    траверсим дерево;
}
```

SLEEP:
__syncthreads();
ReduceMin(addr);

BVH деревья

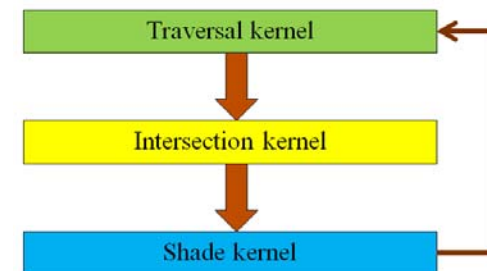
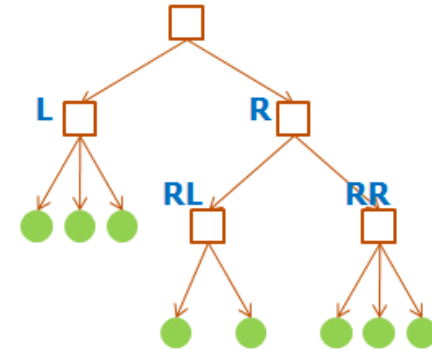
⌘ Бесстековый траверс на CUDA



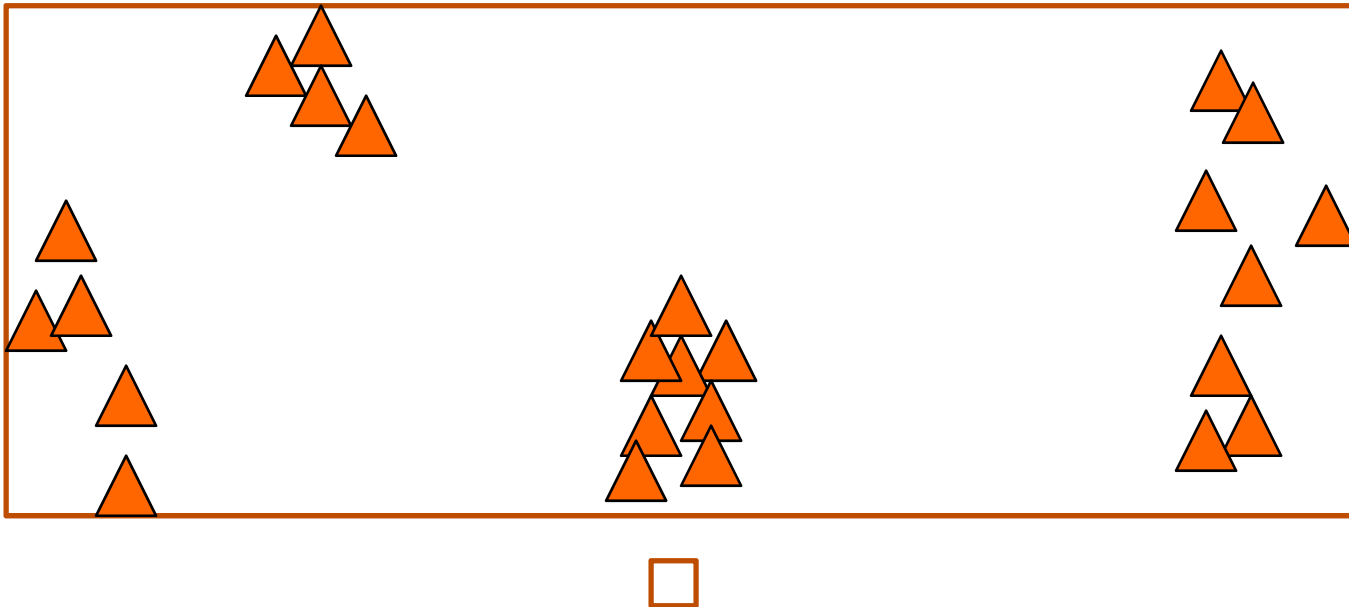
BVH деревья

⌘ Бесстековый траверс на CUDA

- ☑ Ядро уместилось в 20 регистров
- ☑ Узел BVH – 32 байта
- ☑ Shared кэш слишком маленький
- ☑ На больших сценах эффективность падает из-за частых промахов
- ☑ Плохо для некогерентных лучей
- ☑ Лишние пересечения

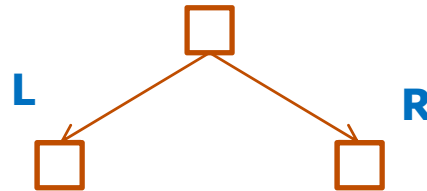
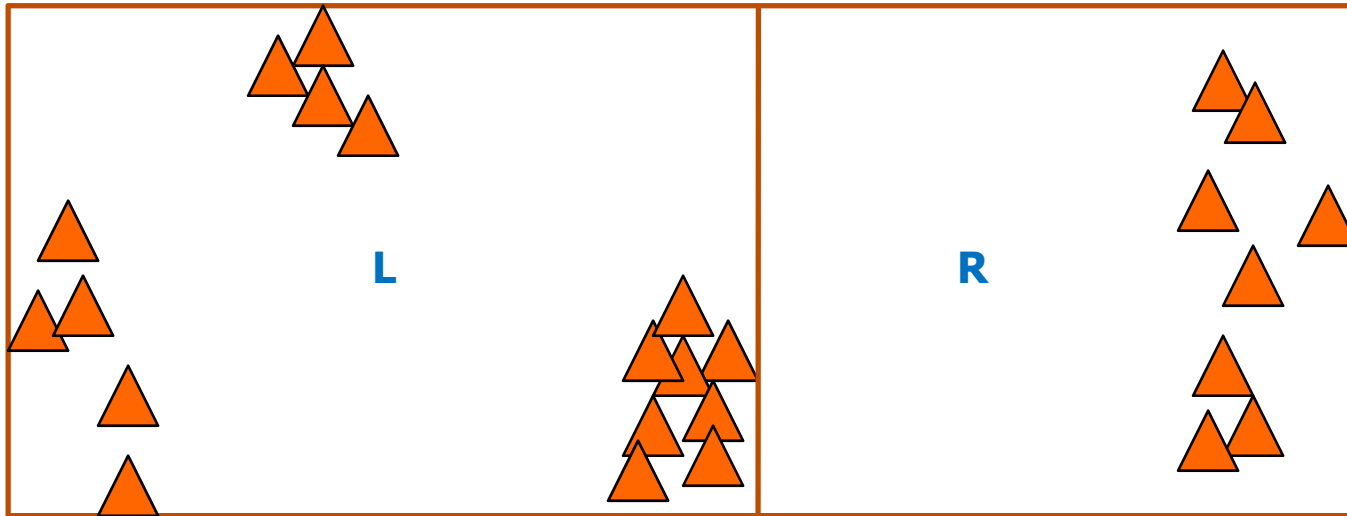


kd-деревья



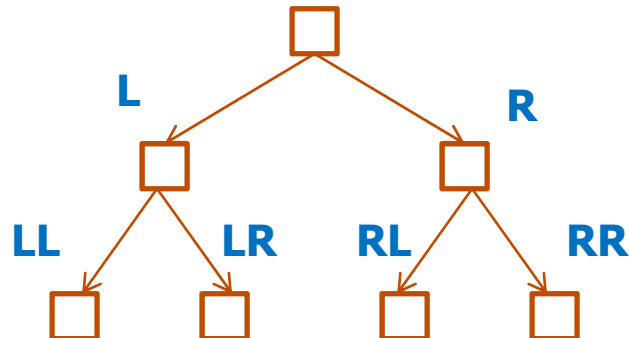
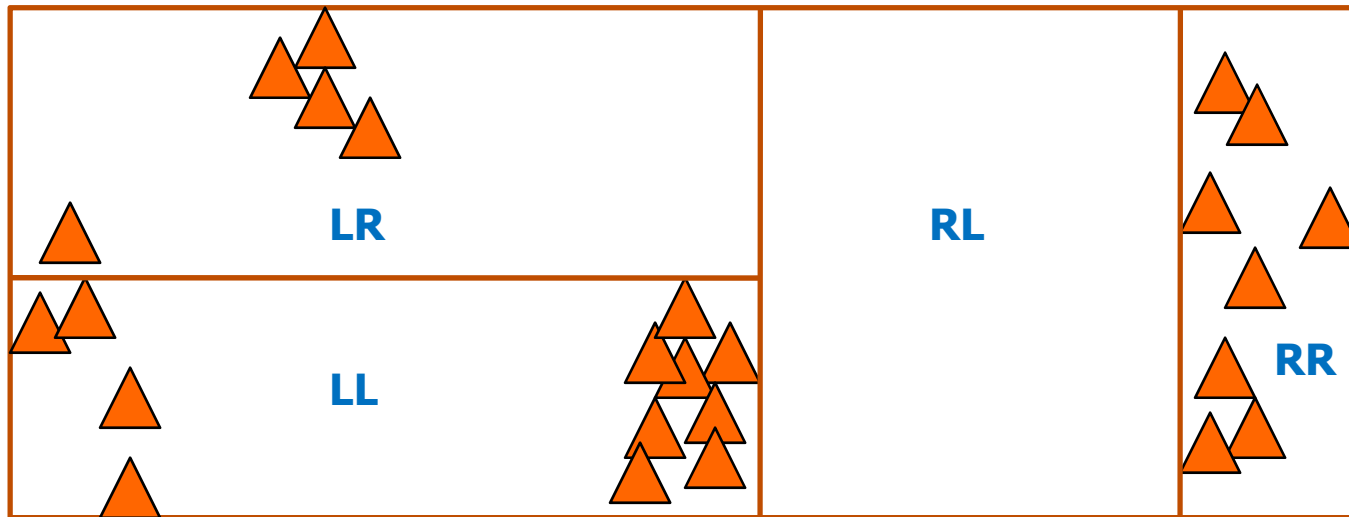
```
struct KdTreeNode
{
    float split;
    uint leftOffset: 29;
    uint splitAxis: 2;
    uint leaf: 1;
};
```

kd-деревья



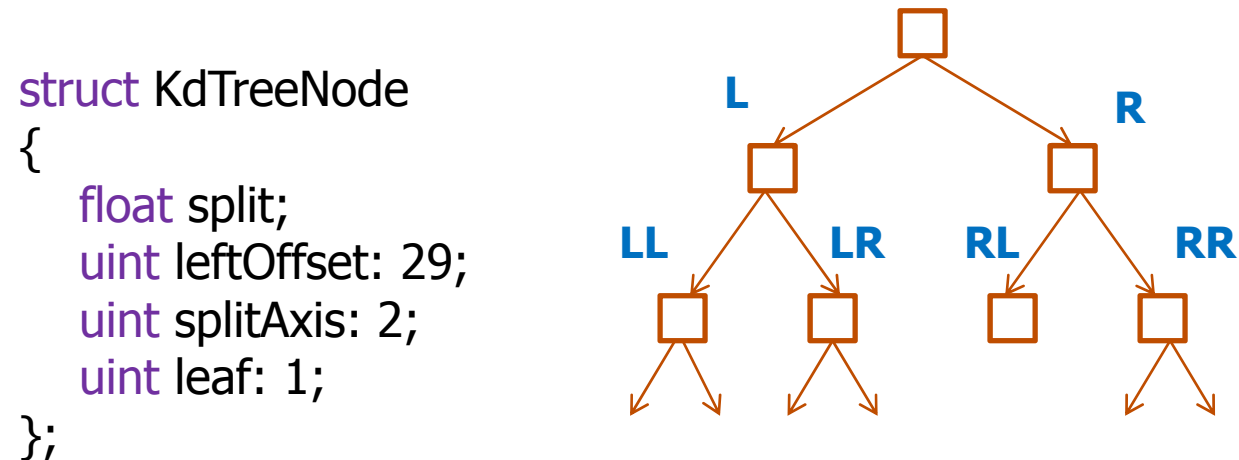
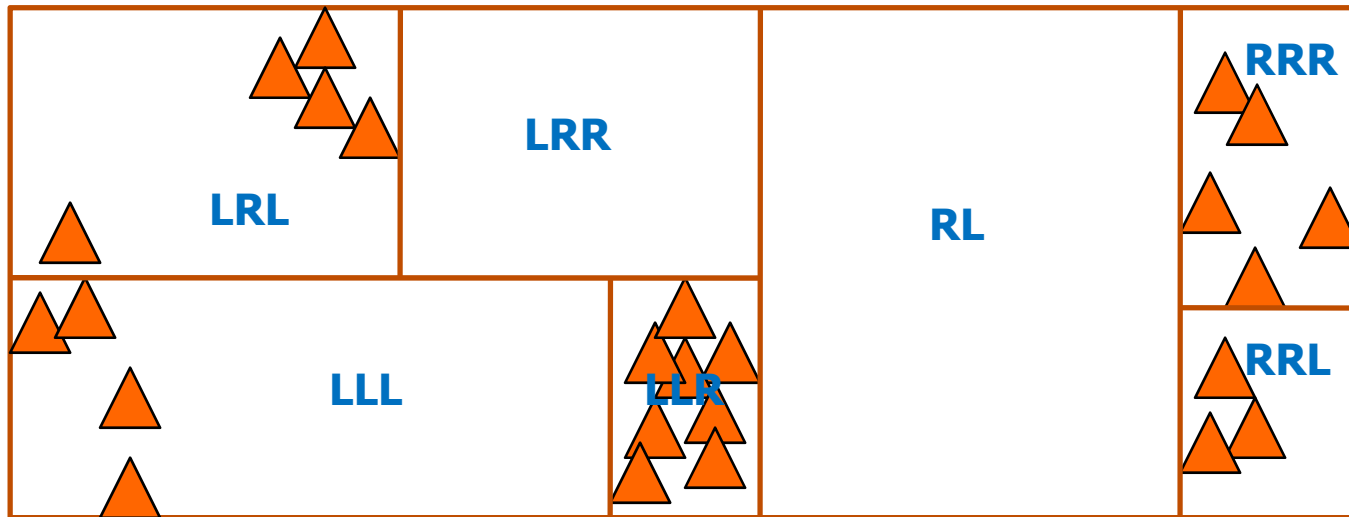
```
struct KdTreeNode
{
    float split;
    uint leftOffset: 29;
    uint splitAxis: 2;
    uint leaf: 1;
};
```

kd-деревья

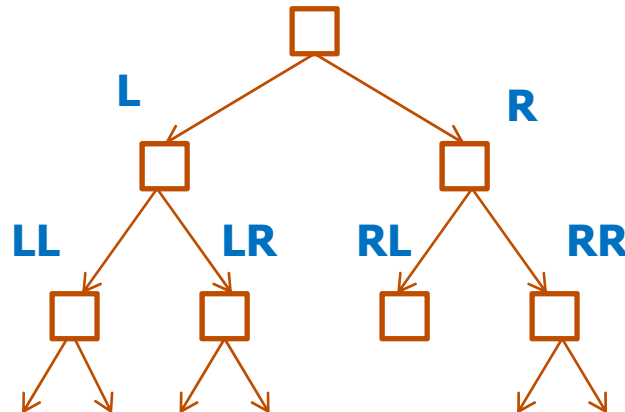
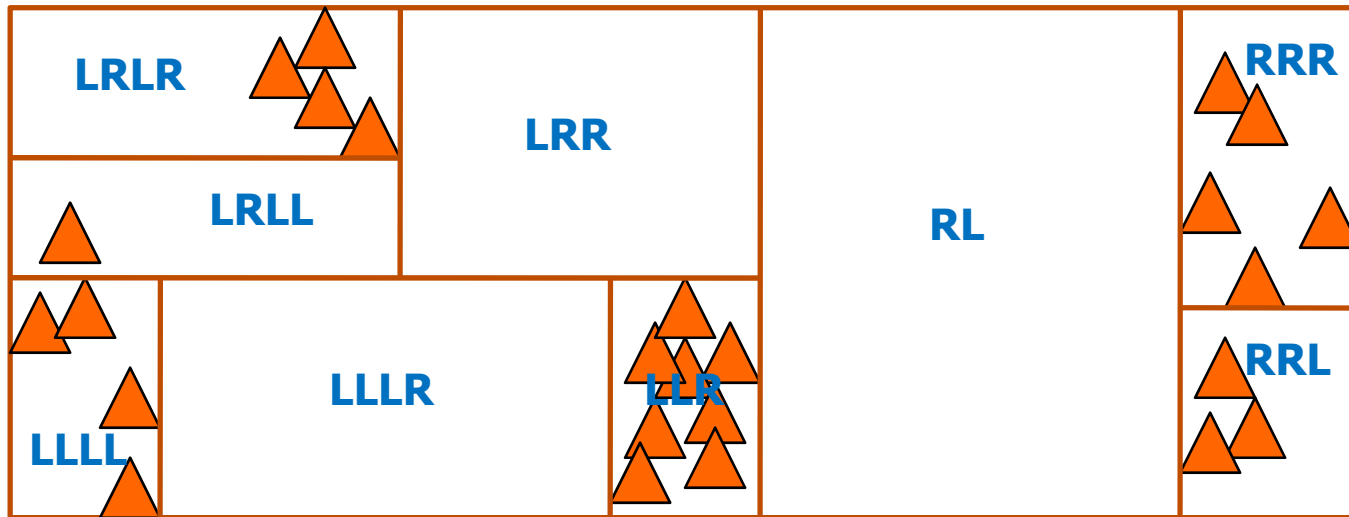


```
struct KdTreeNode
{
    float split;
    uint leftOffset: 29;
    uint splitAxis: 2;
    uint leaf: 1;
};
```


kd-деревья

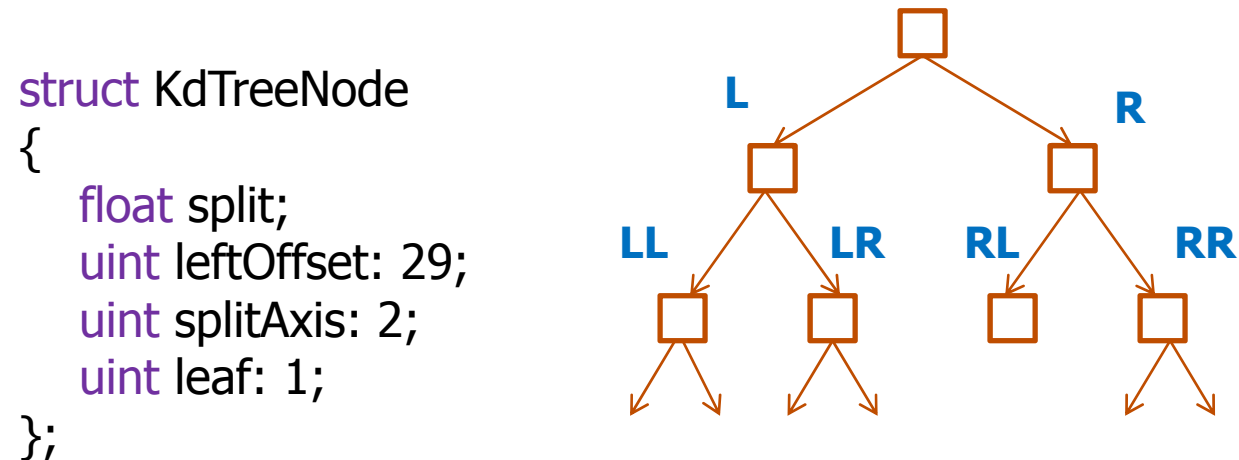
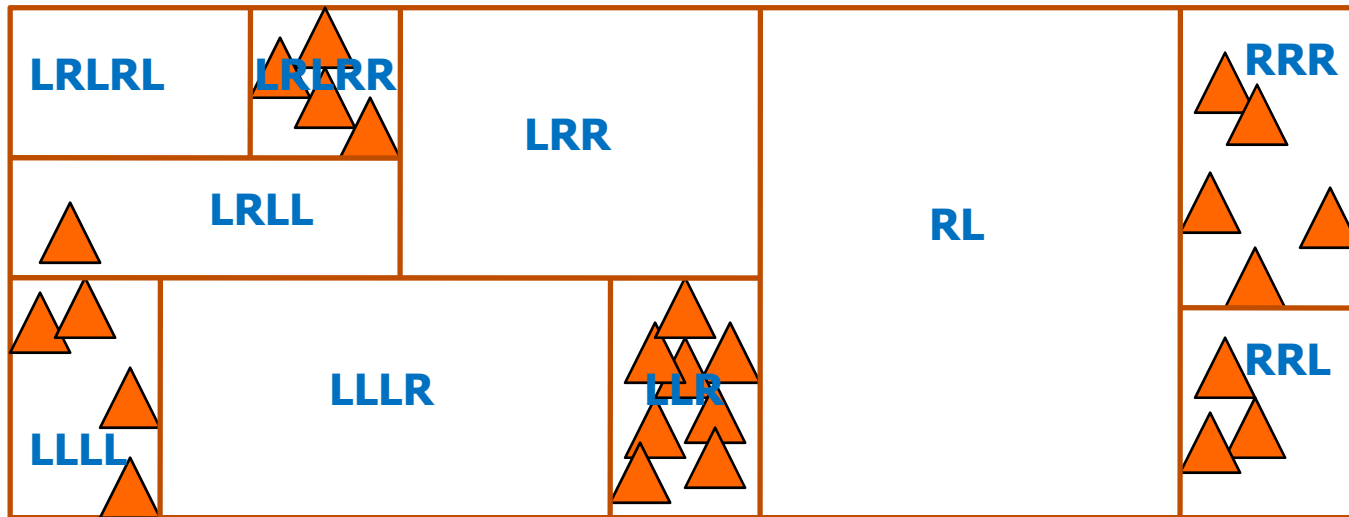


kd-деревья



```
struct KdTreeNode
{
    float split;
    uint leftOffset: 29;
    uint splitAxis: 2;
    uint leaf: 1;
};
```

kd-деревья



kd-деревья

⌘ Алгоритм траверса

⌘ Регистры – 13 min:

☑ луч - 6

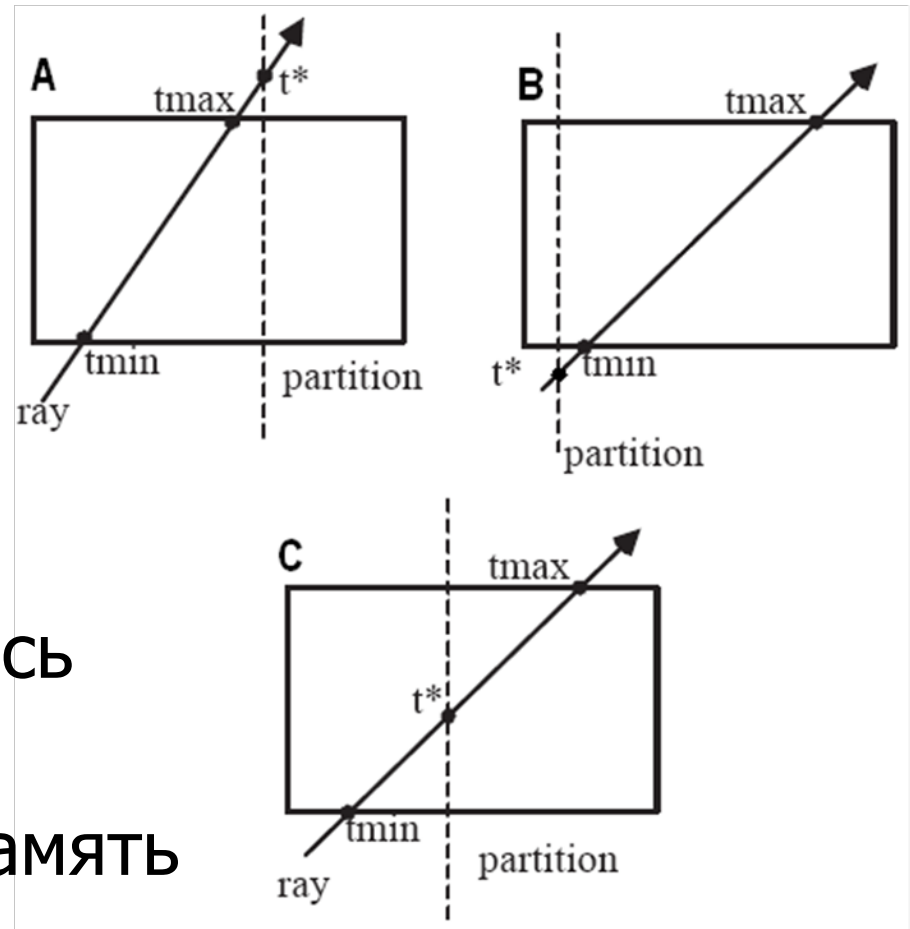
☑ t , t_{\min} , t_{\max} – 3

☑ node – 2

☑ t_{id} , $stack_top$ – 2

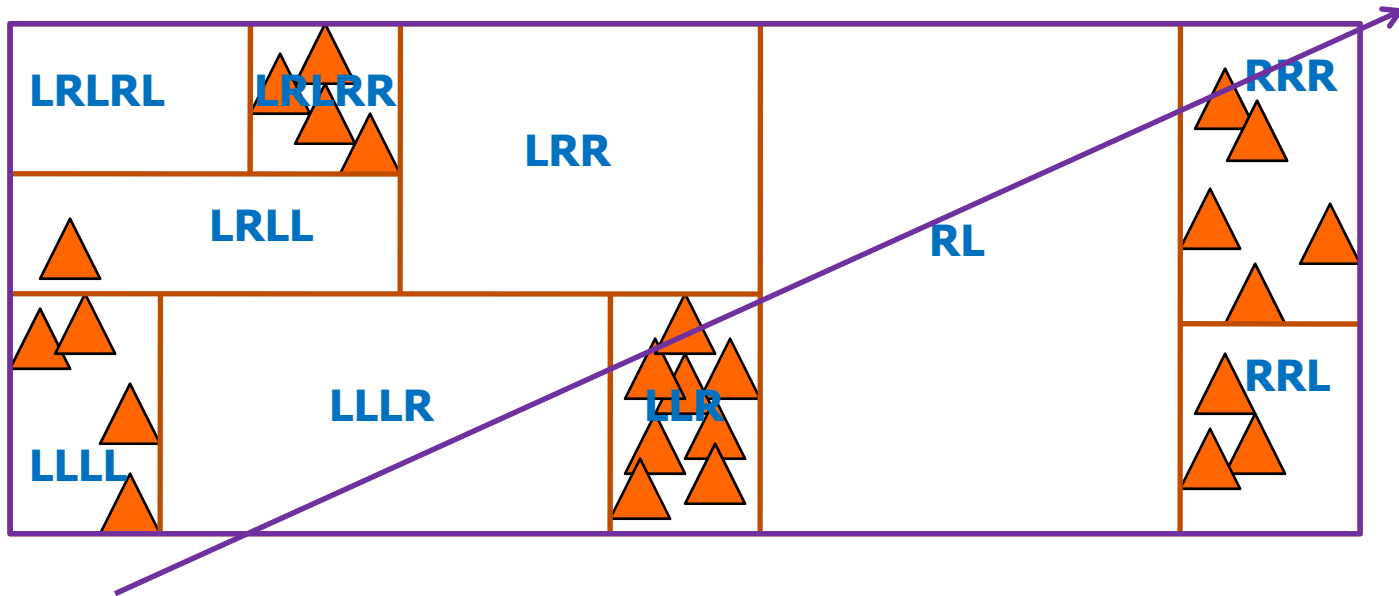
☑ На практике удалось уложиться в 19

☑ Стек: локальная память



kd-деревья

⌘ Алгоритм траверса

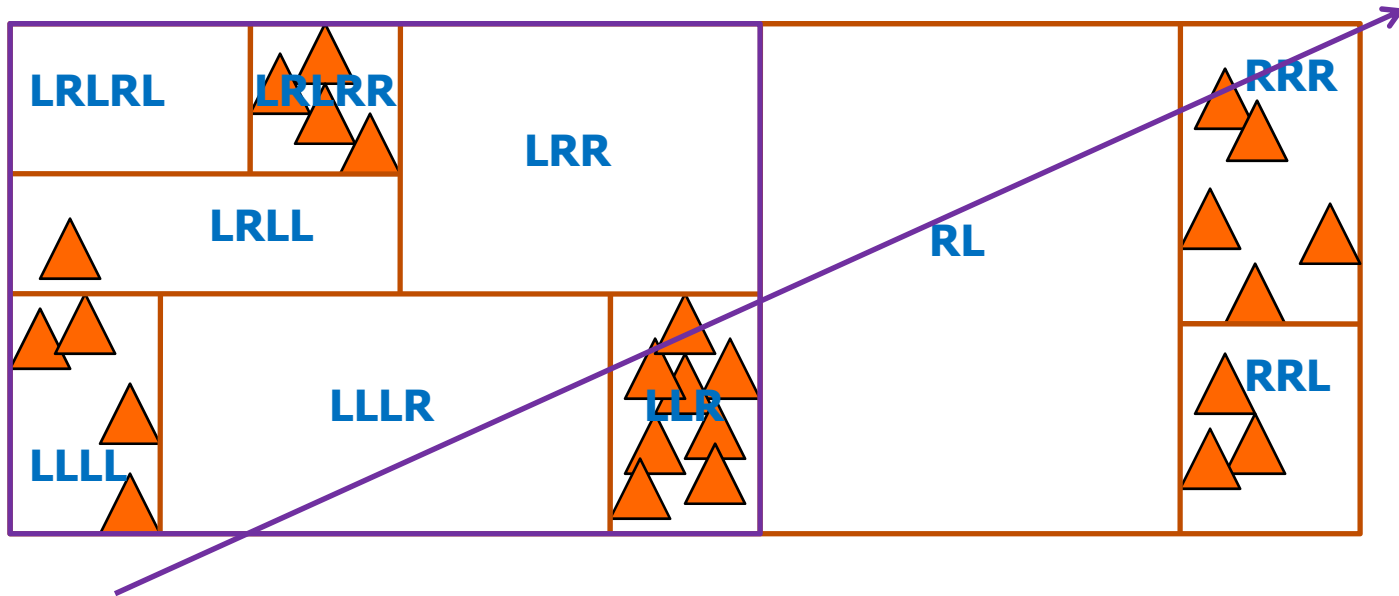


Стек:

Текущий узел:

kd-деревья

⌘ Алгоритм траверса

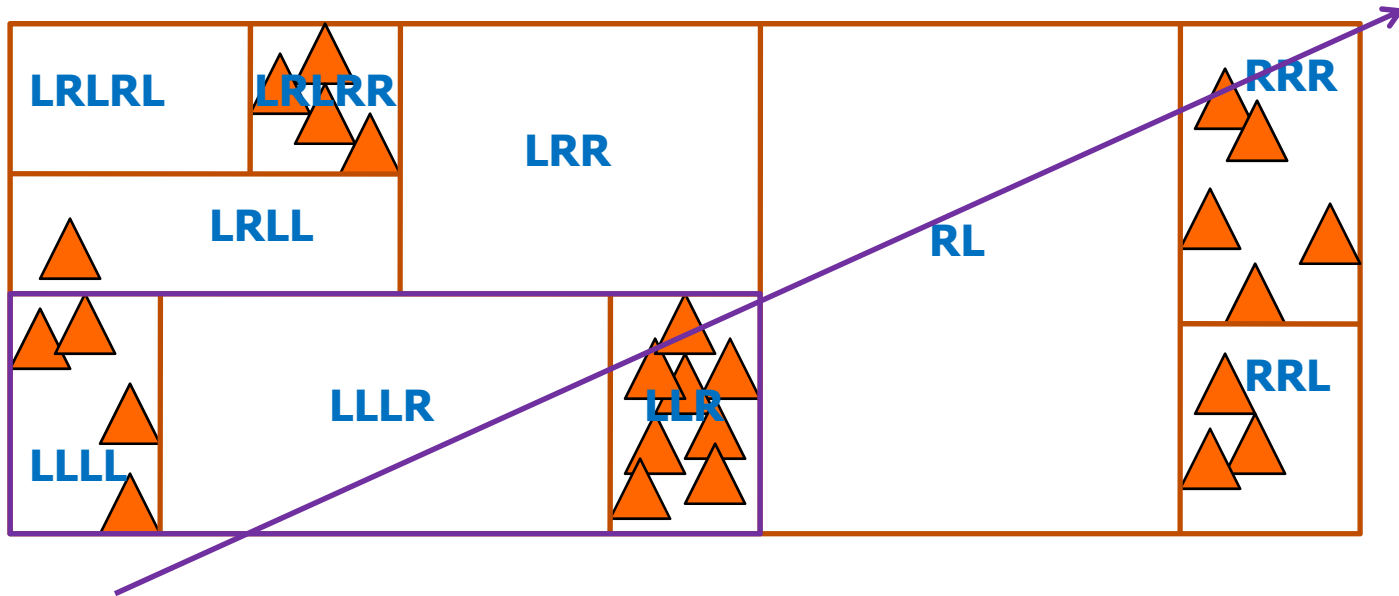


Стек: **R**

Текущий узел: **L**

kd-деревья

⌘ Алгоритм траверса

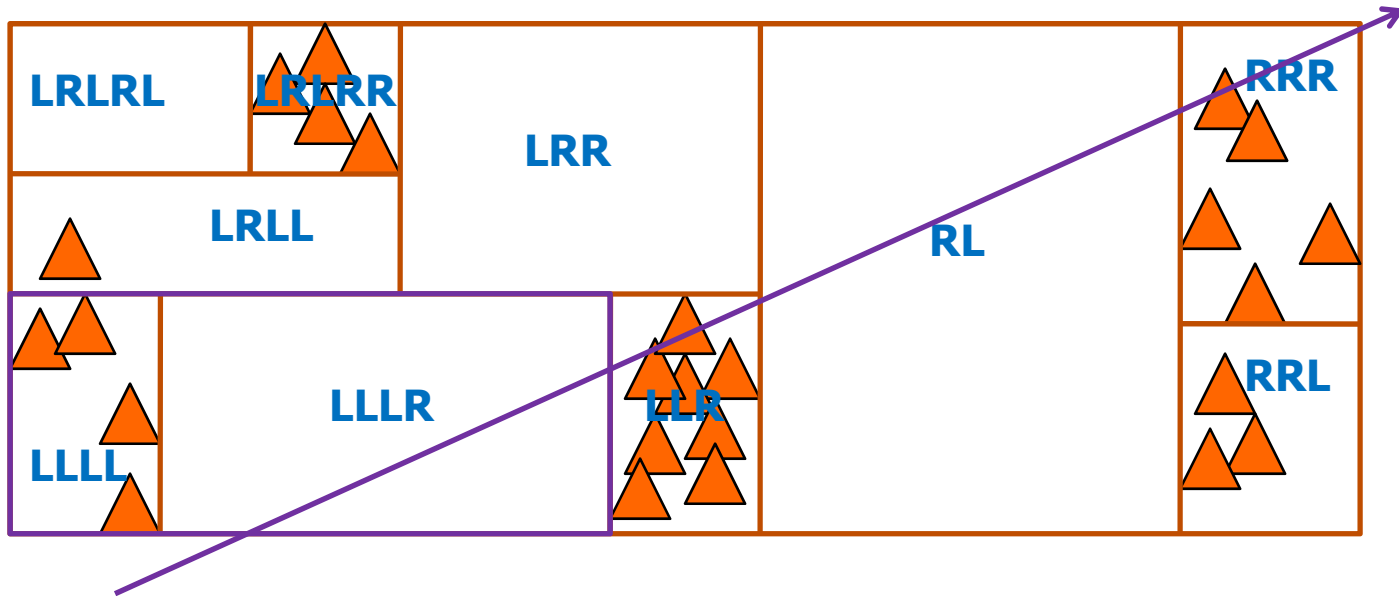


Стек: **R**

Текущий узел: **LL**

kd-деревья

⌘ Алгоритм траверса

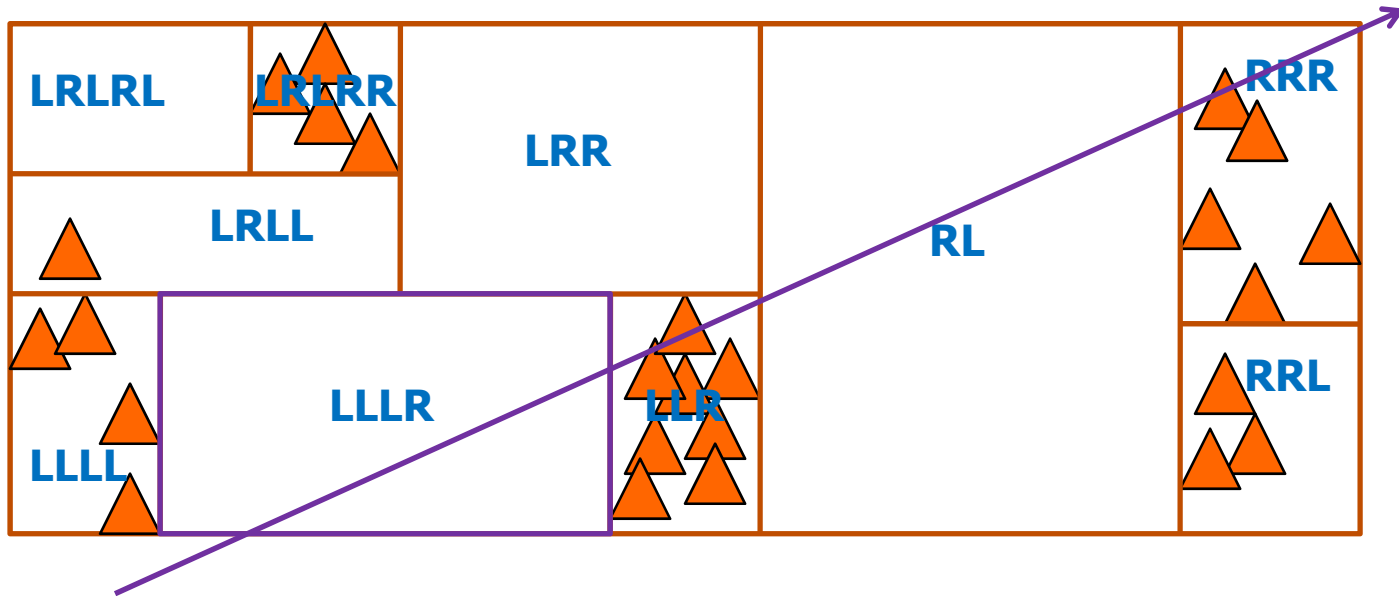


Стек: **LLR, R**

Текущий узел: **LLL**

kd-деревья

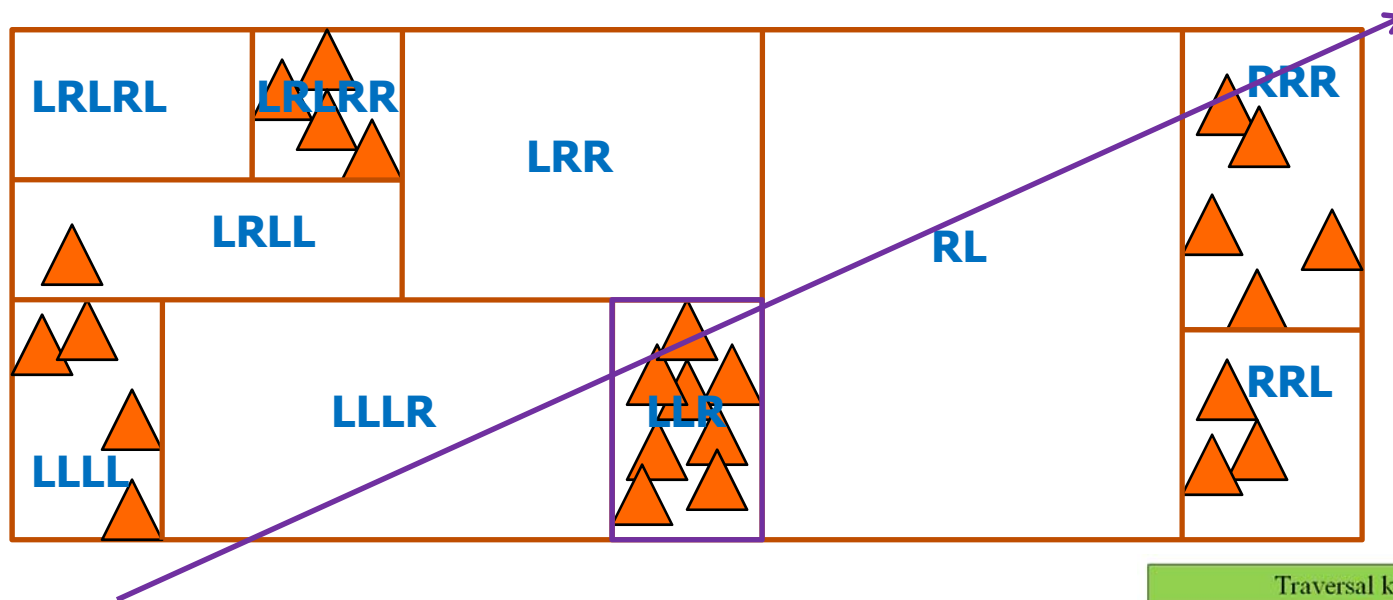
⌘ Алгоритм траверса



Стек: **LLR, R**
→ Текущий узел: **LLLRL**

kd-деревья

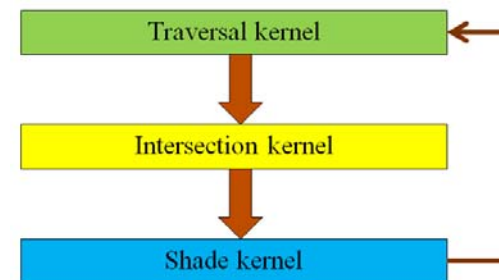
⌘ Алгоритм траверса



Стек: **R**

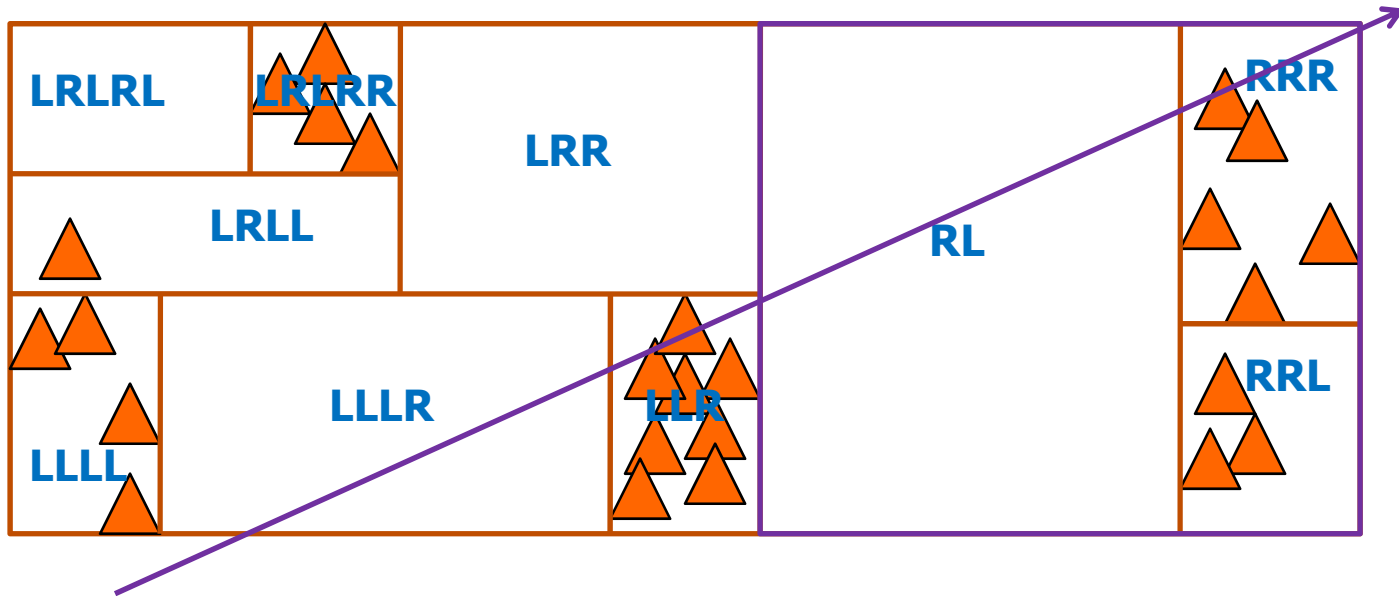
→ Текущий узел: **LLR**

Можно было бы остановиться!



kd-деревья

⌘ Алгоритм траверса



Стек:

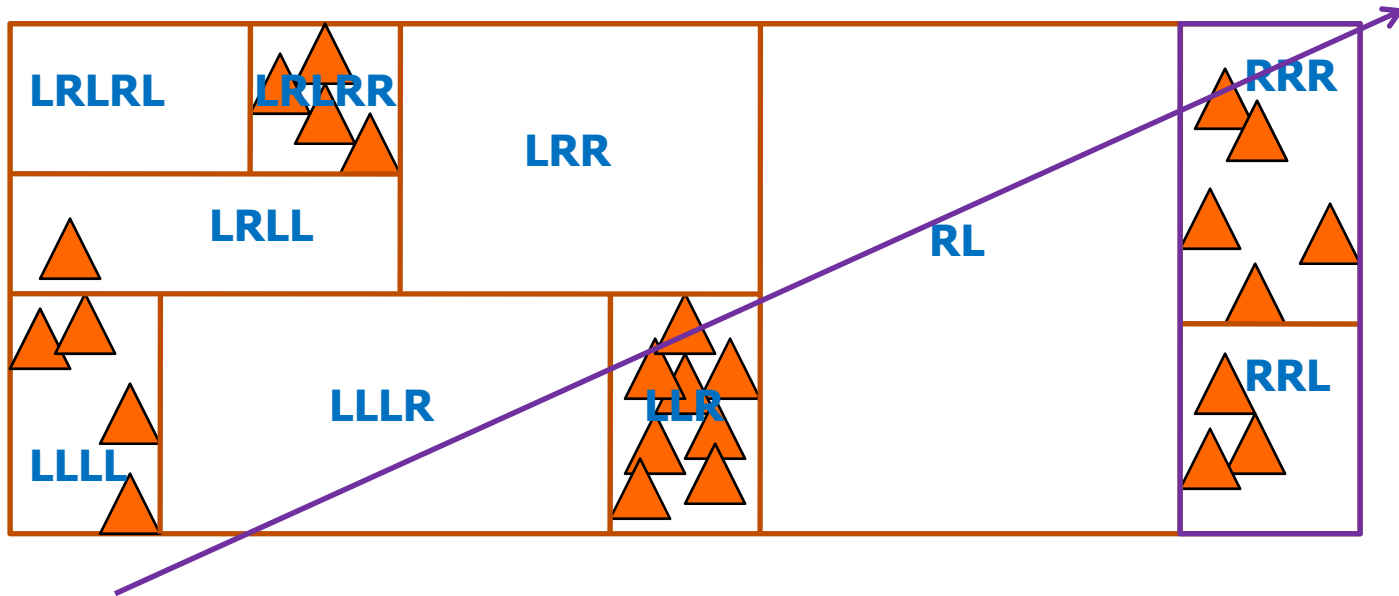
Текущий узел: **R**

⌘ Алгоритм траверса



kd-деревья

⌘ Алгоритм траверса

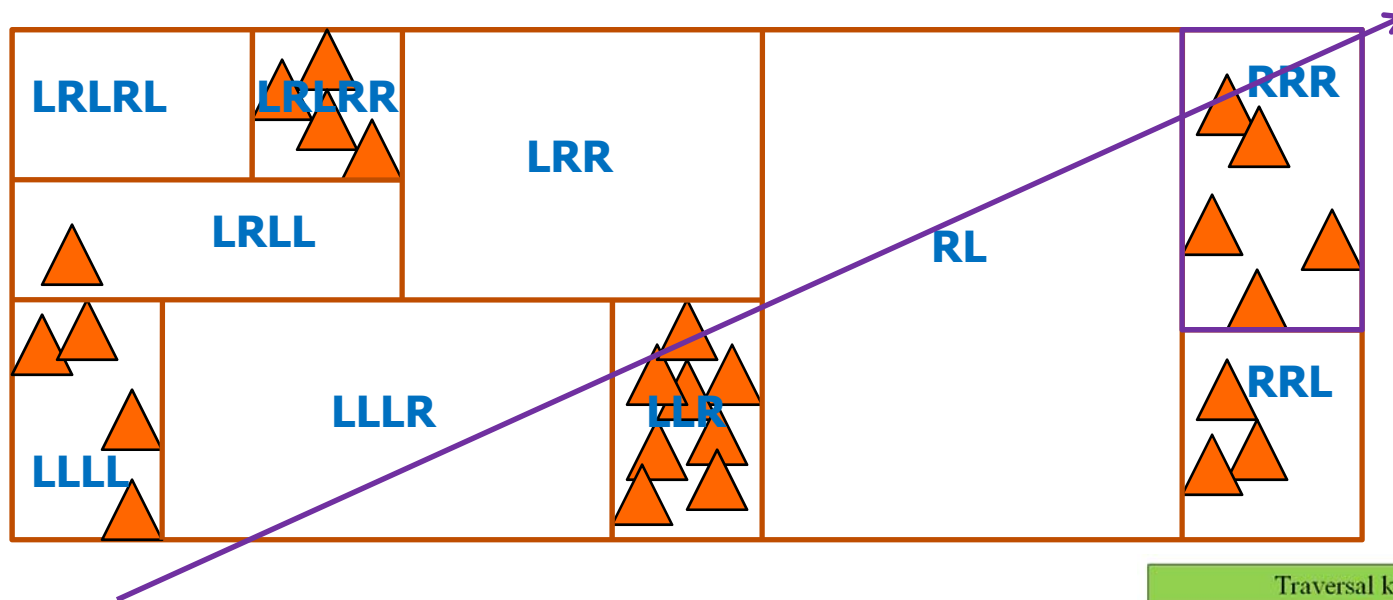


Стек:

Текущий узел: **RR**

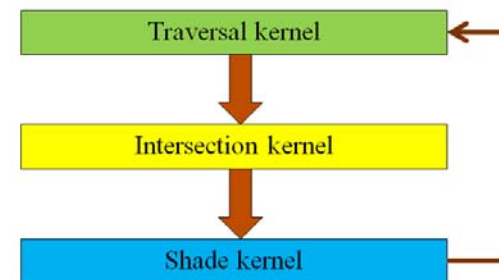
kd-деревья

⌘ Алгоритм траверса



Стек:

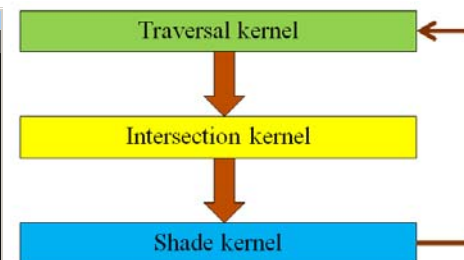
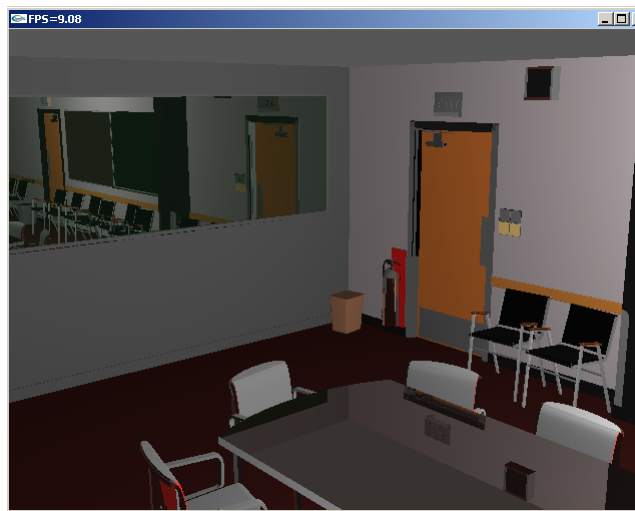
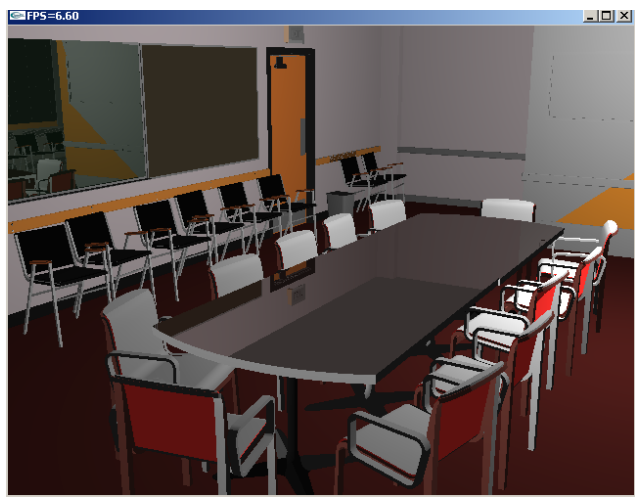
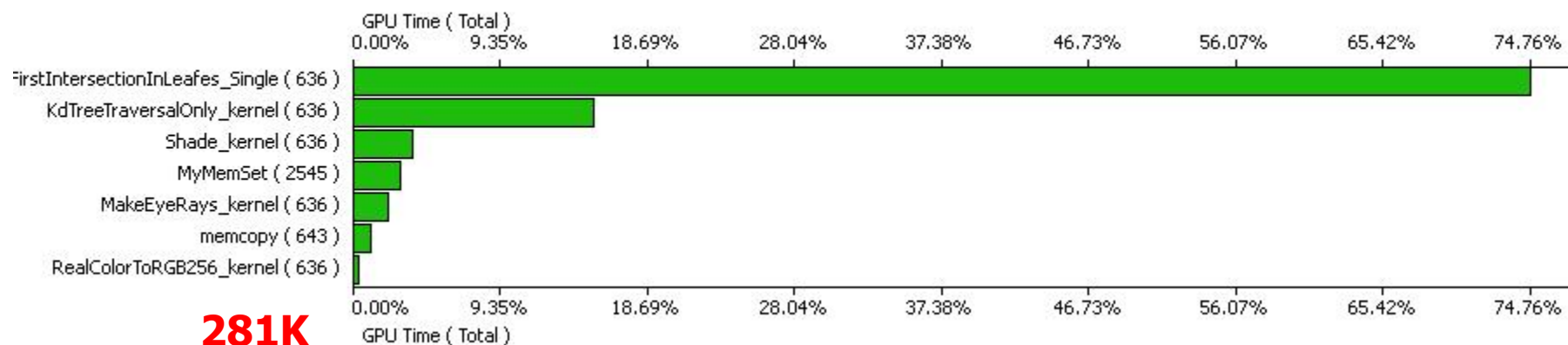
Текущий узел: **RRR** Конец, результат: **LLR, RRR**



kd-деревья

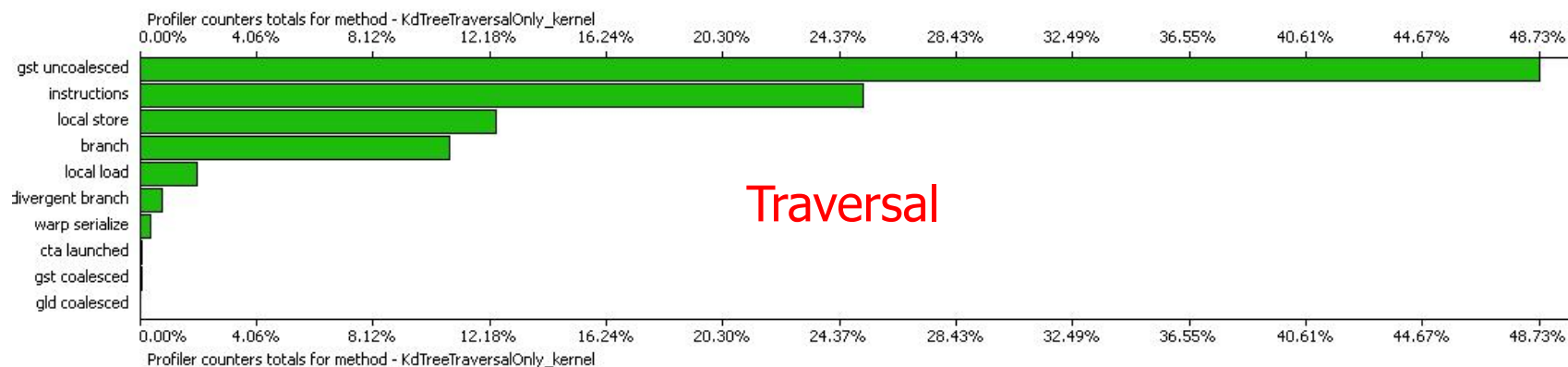
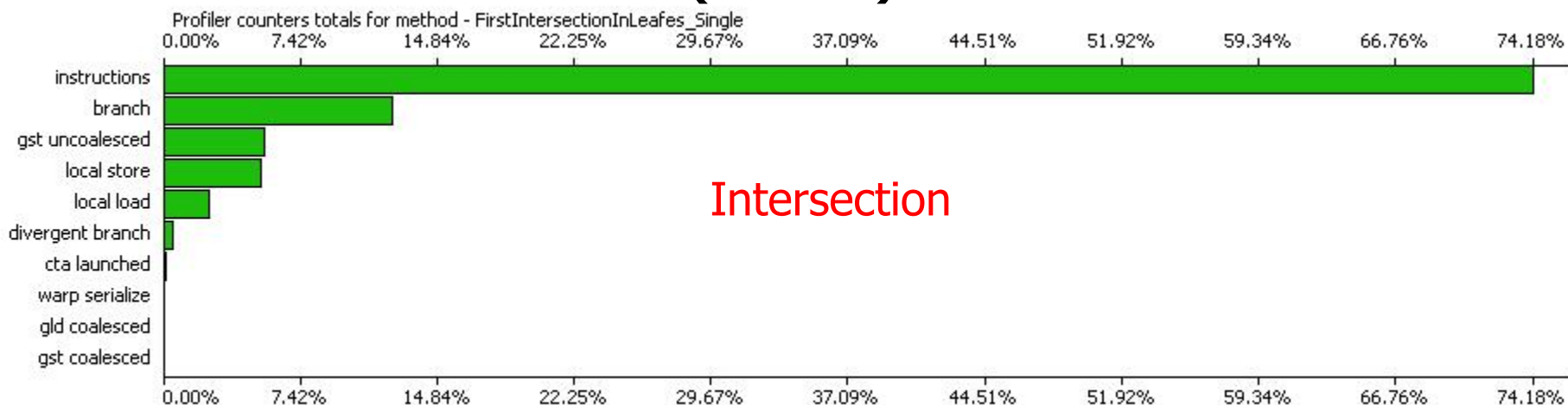
⌘ Где узкое место?

Summary Plot



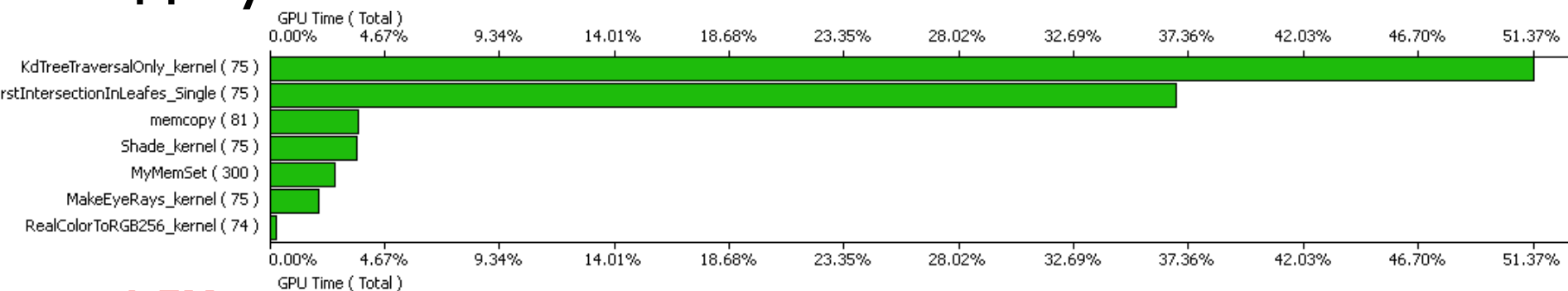
kd-деревья

Conference Room (281K)

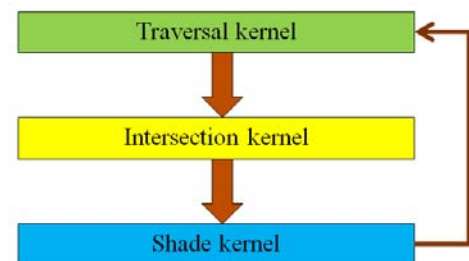
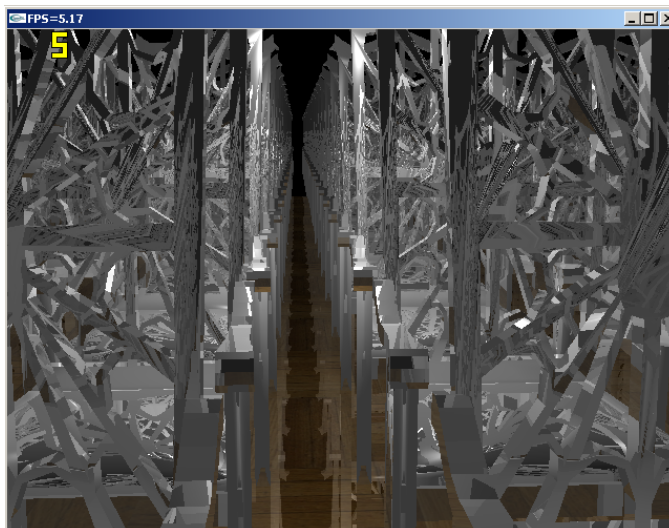
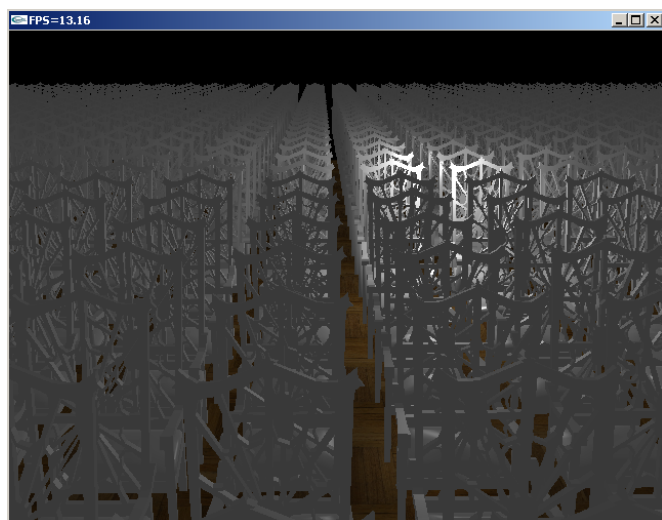


kd-деревья

⌘ Где узкое место?

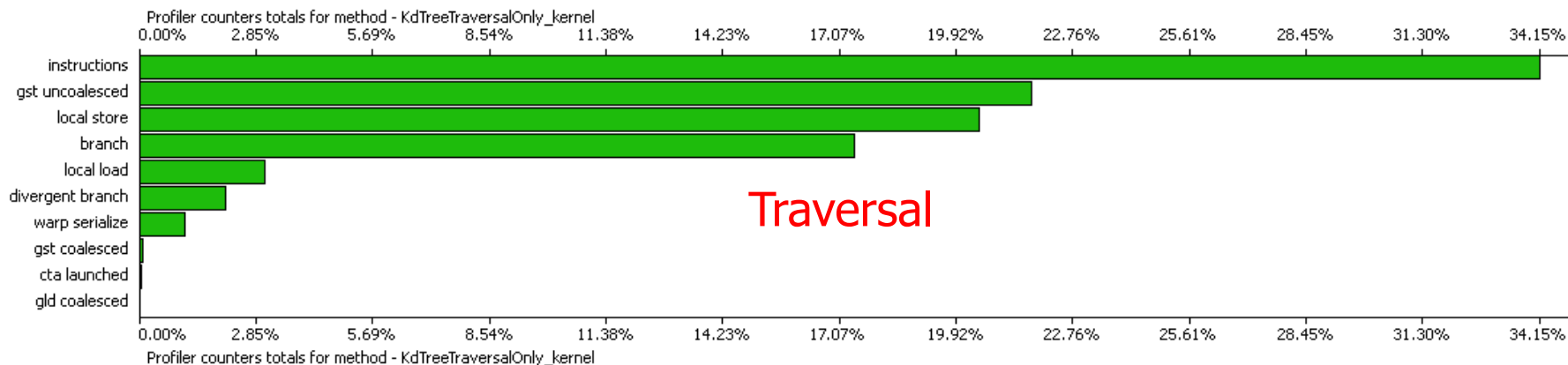
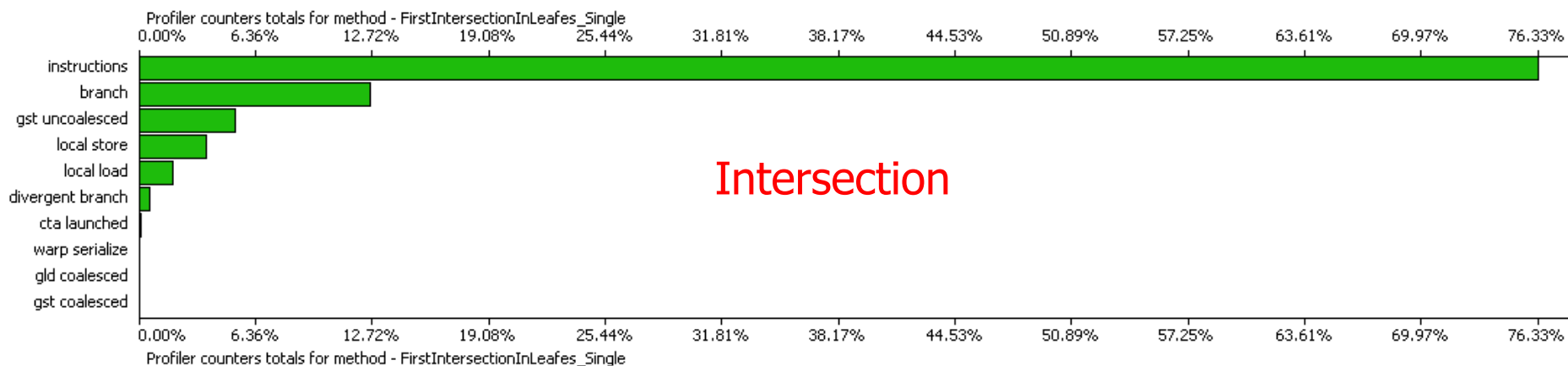


1.5M



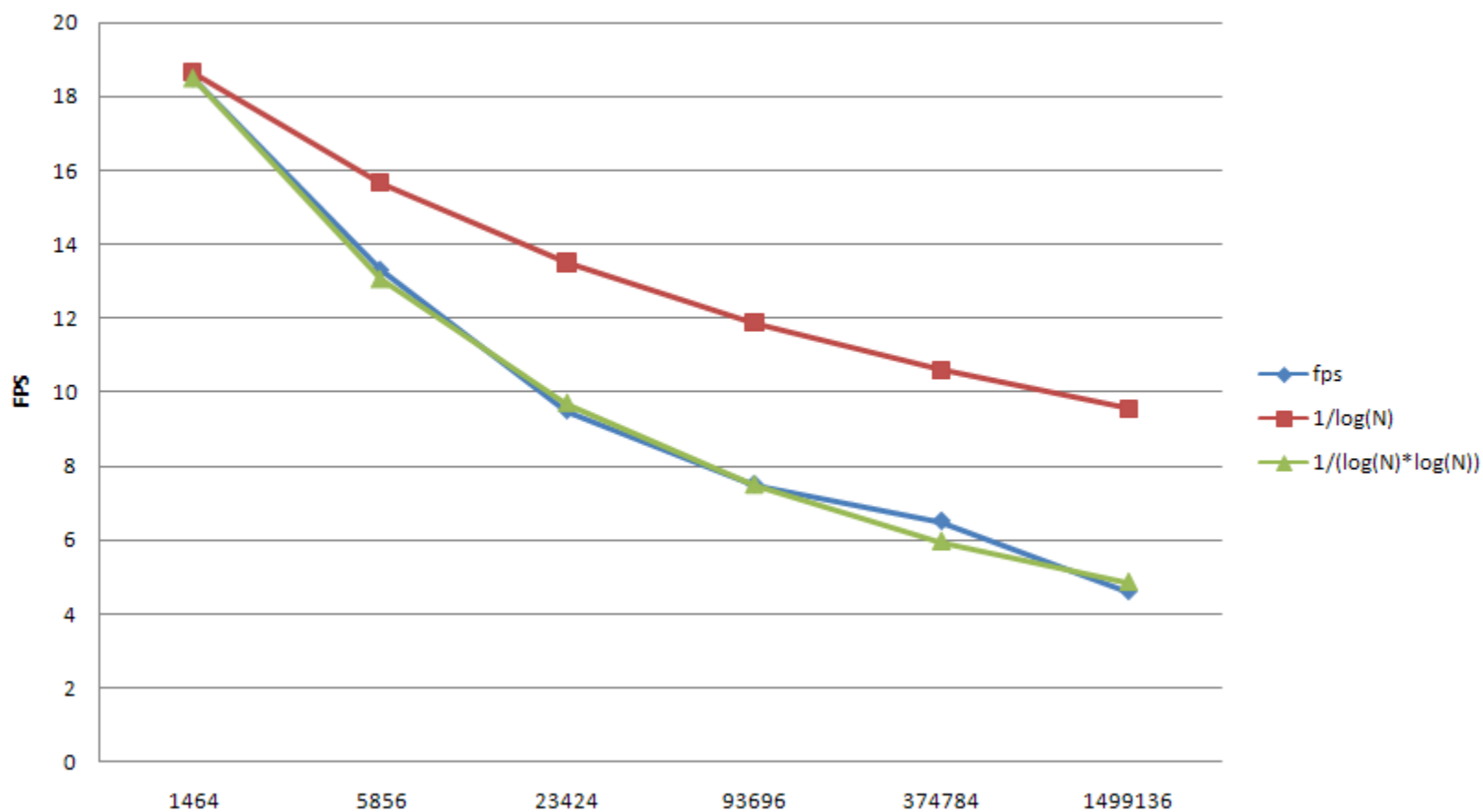
kd-деревья

⌘ Стулья (1.5М)



Производительность

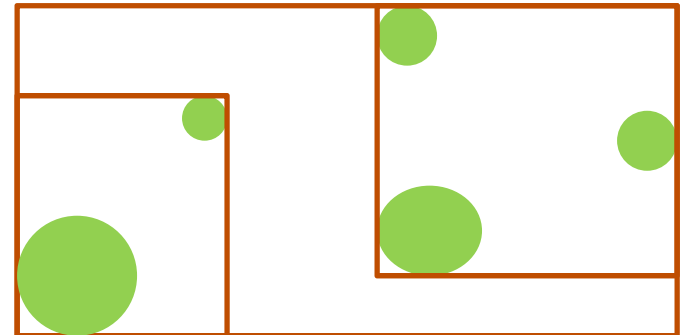
1024x1024, GF8800GTX



kd-tree vs BVH на CUDA

⌘ BVH со стеком на локальной памяти

- ☑ Покрывается латентность текстурной памяти
- ☑ Меньше глубина
- ☑ Алгоритм сложнее, нужно больше регистров
- ☑ Лишние плоскости



⌘ kd-tree

- ☑ Экономит регистры
- ☑ Можно эффективнее задействовать кэш?
- ☑ 1 mad и пара ветвлений на одну tex1Dfetch

Грабли



⌘ Что делает этот код?

⌘ Почему он так написан?

```
const float3 V[2];
```

```
register int sign = (r.dir.x < 0);
```

```
register float tmin = ( V[sign].x - r.pos.x) / r.dir.x;
```

```
register float tmax = ( V[1-sign].x - r.pos.x) / r.dir.x;
```

⌘ А как надо?

```
if (r.dir.x < 0);
```

```
    tmin = ( V[1].x - r.pos.x) / r.dir.x;
```

```
else
```

```
    tmin = ( V[0].x - r.pos.x) / r.dir.x;
```

Габли

⌘ Очень удобно?

```
struct VECTOR
{
    union
    {
        struct
        {
            float x,y,z;
        };
        float M[3];
    };
};
```

```
struct VECTOR
{
    float M[3];

    IDH_CALL float& x() { return M[0]; }
    IDH_CALL float& y() { return M[1]; }
    IDH_CALL float& z() { return M[2]; }

    IDH_CALL float x() const { return M[0]; }
    IDH_CALL float y() const { return M[1]; }
    IDH_CALL float z() const { return M[2]; }
};
```

```
#define IDH_CALL inline __device__ __host__
```

Ссылки



- ⌘ <http://ray-tracing.ru>
- ⌘ http://home.mindspring.com/~eric_rollins/ray/cuda.html (не надо так делать)
- ⌘ http://www.nvidia.com/object/cuda_home.html#state=detailsOpen;aid=73926e40-99f0-11dd-ad8b-0800200c9a66
- ⌘ <http://graphics.stanford.edu/papers/i3dkdtree/> (rt на GPU: HLSL, CUDA)
- ⌘ <http://www.mpi-inf.mpg.de/~guenther/BVHonGPU/index.html> (BVH packet traversal + CUDA)
- ⌘ <http://www.cs.unc.edu/~lauterb/GPUBVH/> (BVH, CUDA, быстрое построение)

