Особенности реализации алгоритмов трассировки лучей на CUDA

- Лекторы:
  - Фролов В.А. (ВМиК МГУ)
  - Игнатенко А.В. (ВМиК МГУ)
  - 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

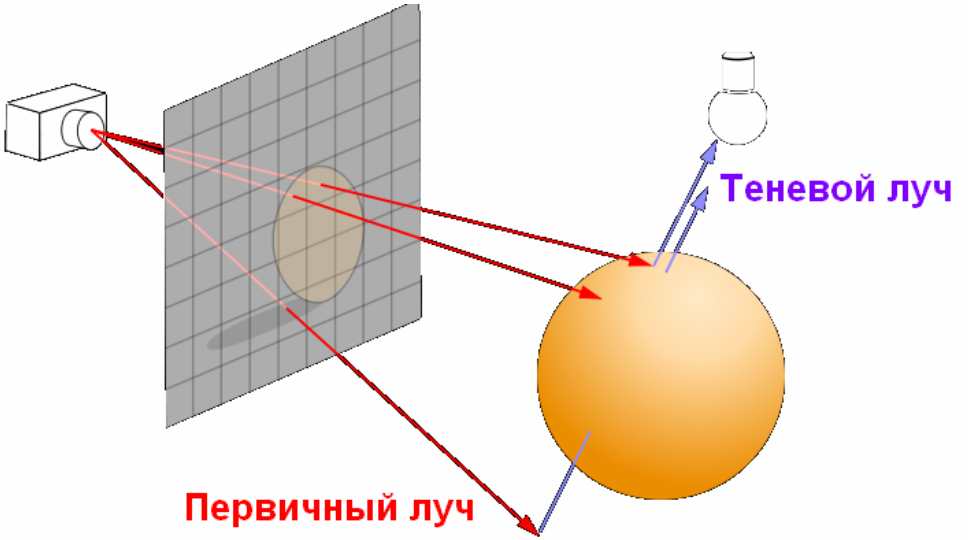
- Скорость в ущерб качеству

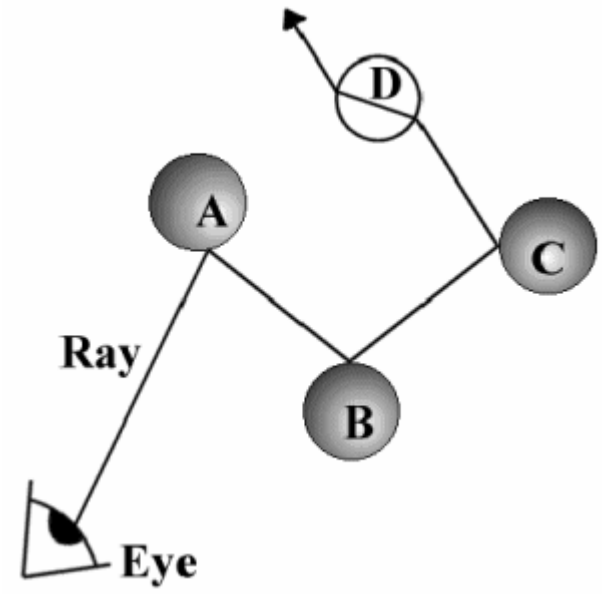
Ray Tracing

- **Точность**
  - Path tracing
  - Фотонные карты
  - Распределенная трассировка лучей (стохастическая)
- **Скорость**
  - Обратная трассировка лучей
  - Растеризация + обратная трассировка лучей

Обратная трассировка лучей

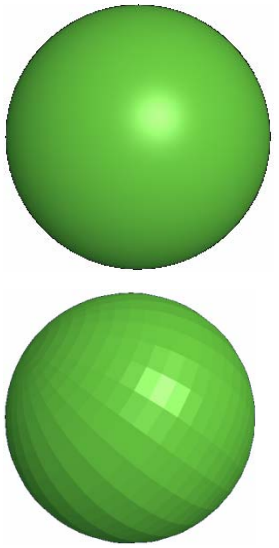
- **Алгоритм**
  - Первичные, теневые, отраженные лучи

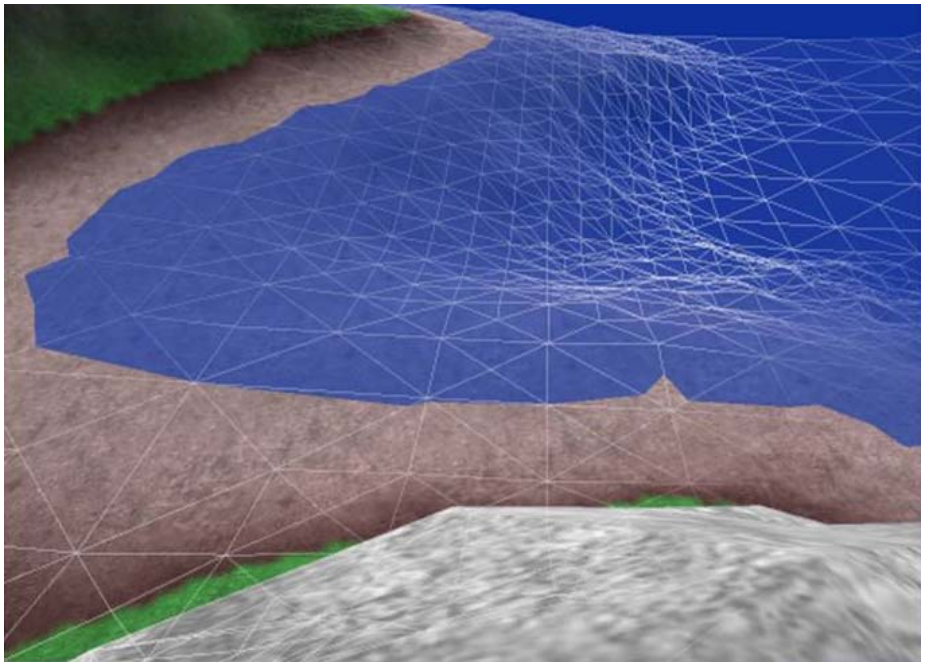


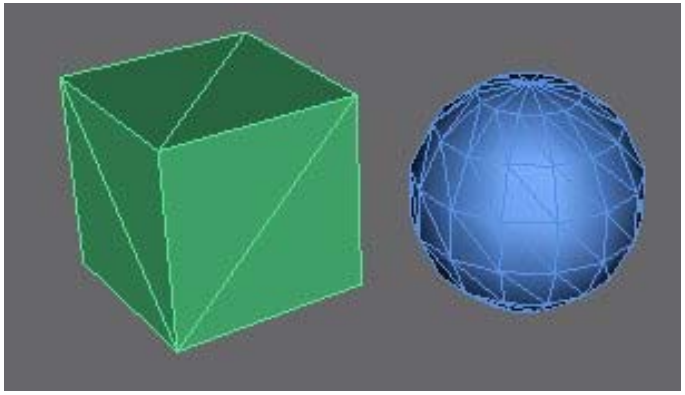


Ray Tracing

- **Представление 3D объектов**
  - Аналитическое
  - Меши из треугольников

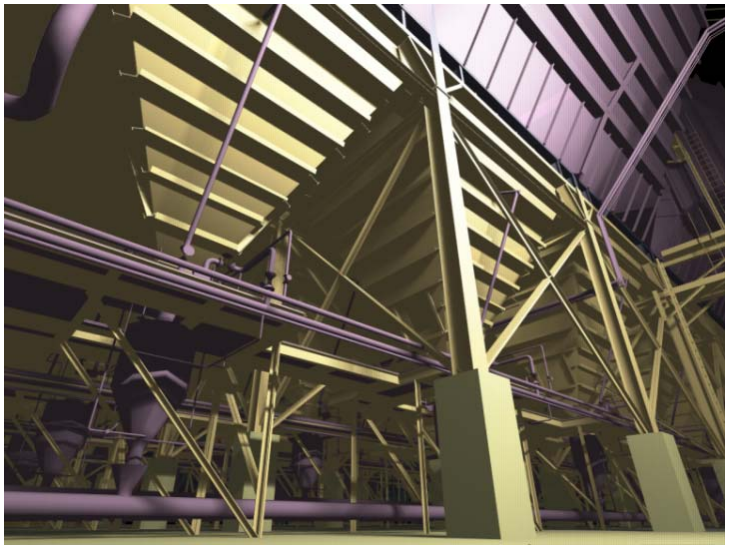






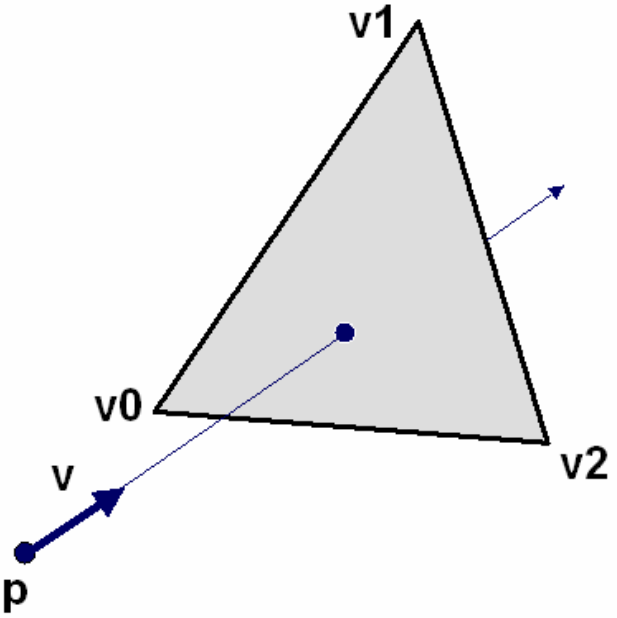
Ray Tracing

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



Пересечение луча и треугольника

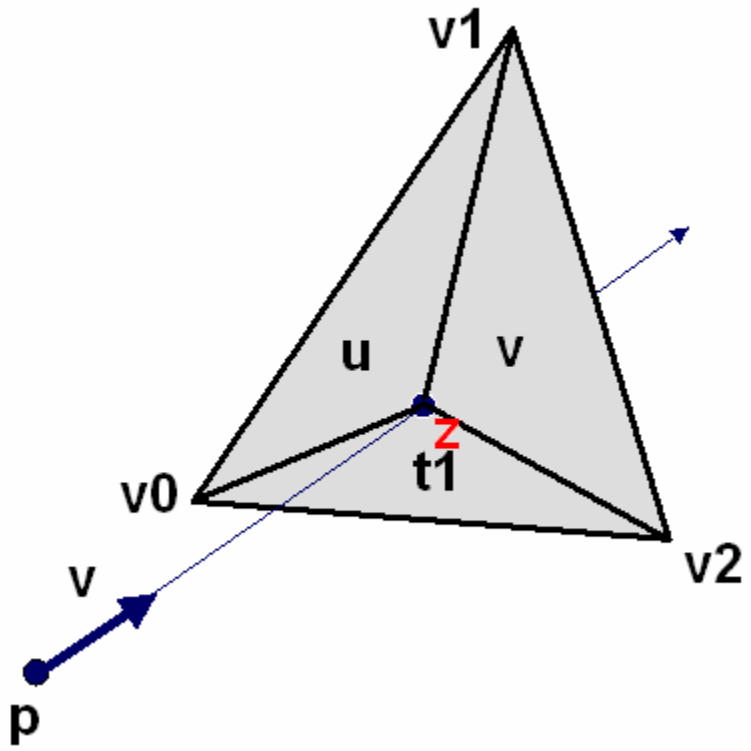
- **Простой вариант**
  - $Ax + By + Cz + D = 0$
  - Найти t
    - $x = p.x + v.x \cdot t$
    - $y = p.y + v.y \cdot t$
    - $z = p.z + v.z \cdot t$



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

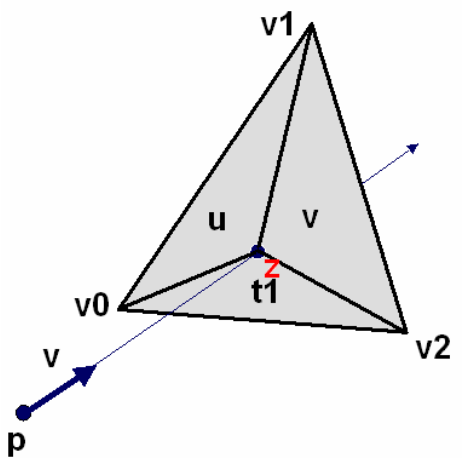
Пересечение луча и треугольника

- **Простой вариант**
  - *t* известно
    - *z = p + v*t*
    - *S = cross(v1-v0, v2-v0)*
    - *u = cross(v1-z, v0-z)*
    - *v = cross(v1-z, v2-z)*
    - *t1 = cross(v2-z, v0-z)*
  - $|u + v + t1 - S| < \varepsilon$



Пересечение луча и треугольника

- **Оптимизированный вариант**
  - Барицентрические координаты
    - $u := u/S,\ v := v/S,\ t1 := t1/S$
    - $t1 = 1 - u - v$



$$z(u,v) = (1-u-v)v1 + uv2 + vv0$$

$$z(t) = p + td$$

$$p + td = (1-u-v)v1 + uv2 + vv0$$

  - 3 уравнения, 3 неизвестных

Пересечение луча и треугольника

- **Оптимизированный вариант**

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

$$E1 = v1 - v0$$

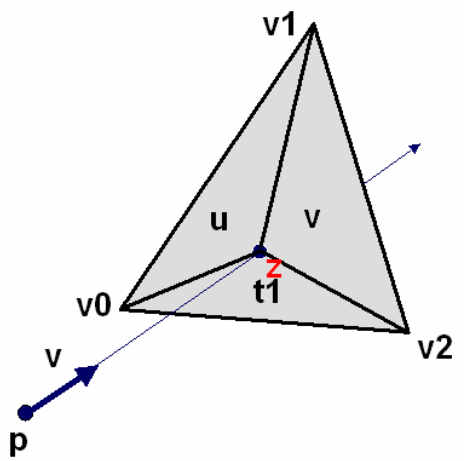
$$E2 = v2 - v0$$

$$T = p - v0$$

$$P = cross(D, E2)$$

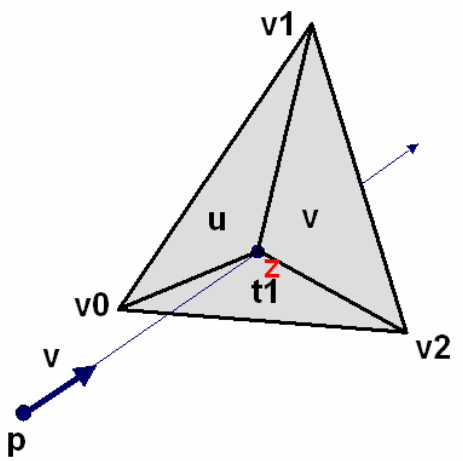
$$Q = cross(T, E1)$$

$$D = v$$



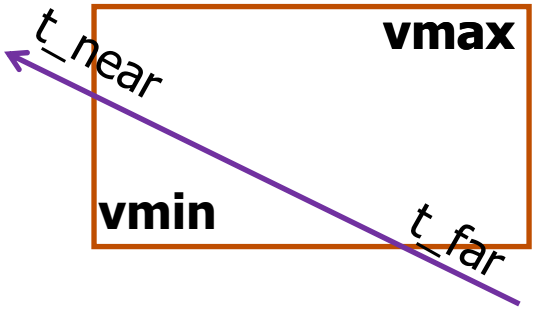
Пересечение луча и треугольника

- **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

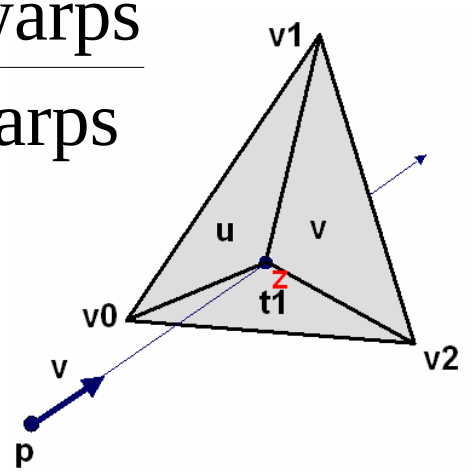


- **Сфера**
  - ~13 mad + sqrtf == 52 + 32 = 84 такта
  - меньше ветвлений
  - Иерархия из сфер не лучше иерархии из боксов

Multiprocessor Occupancy

- Регистры

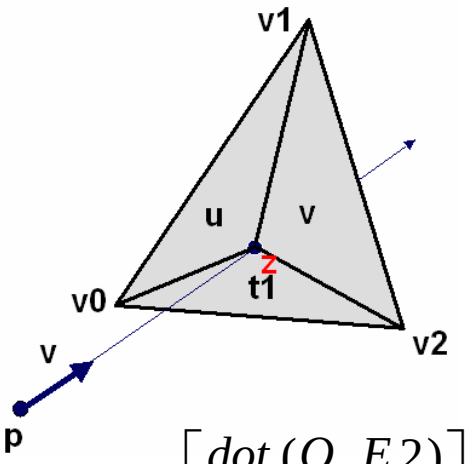
$$\text{occupancy} = \frac{\text{active warps}}{\text{max warps}}$$



  - 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!**



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

$$E1 = v1 - v0$$

$$E2 = v2 - v0$$

$$T = p - v0$$

$$P = cross(D, E2)$$

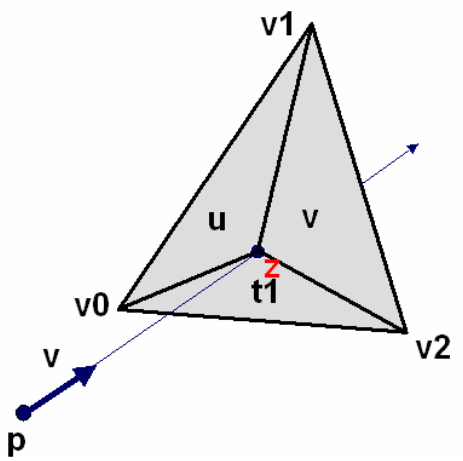
$$Q = cross(T, E1)$$

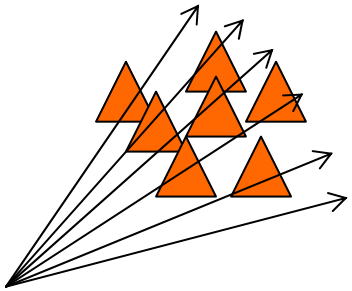
$$D = v$$

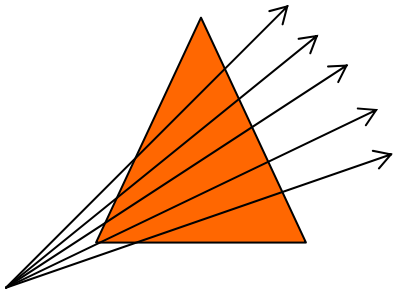
Пересечение луча и треугольника

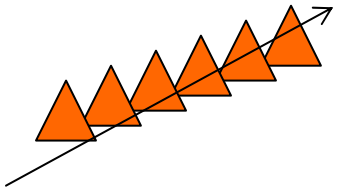
- Организация сетки - блоки 8x8
- Что общее для нитей блока?

1. Свой луч, свой треугольник
2. Свой луч, общий треугольник
3. Общий луч, свой треугольник



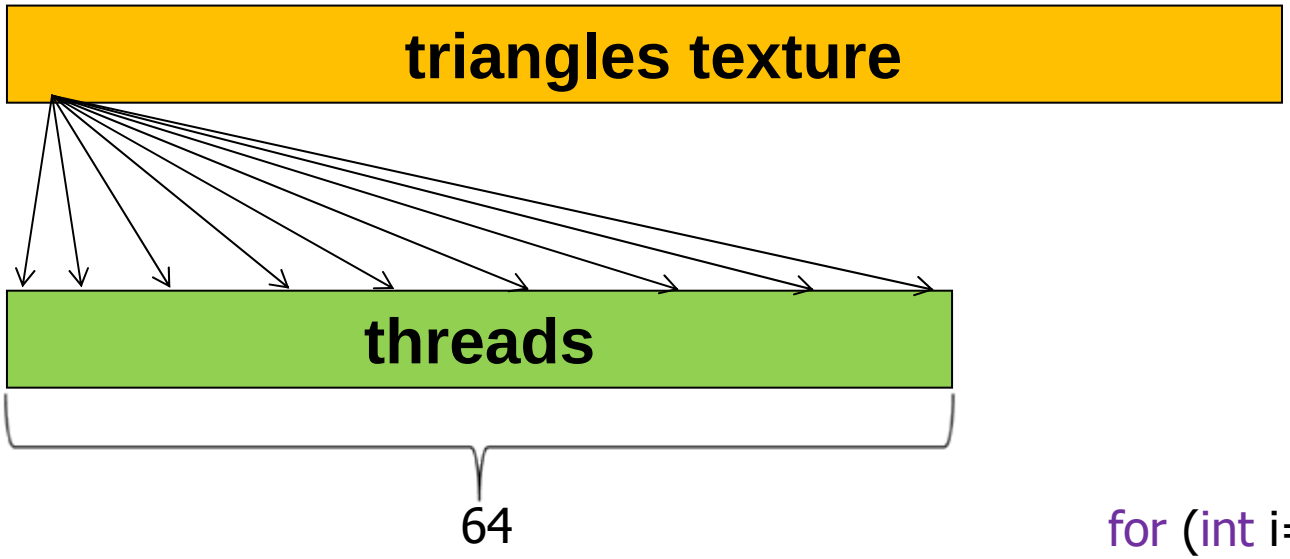


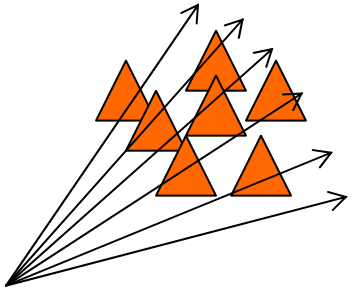




Пересечение луча и треугольника

- Свой луч, свой треугольник

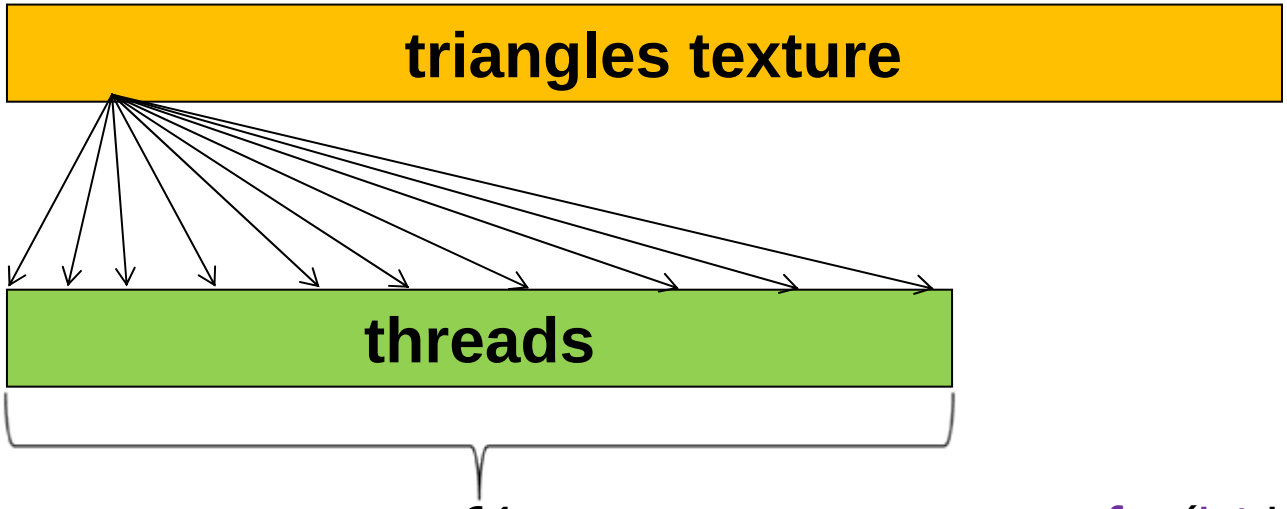


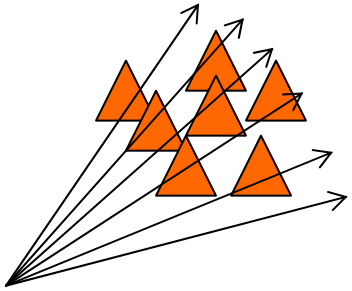


```
for (int i=0;i<triNum;i++)
{
   (A,B,C) = tex1Dfetch(tex,i);
   // intersection code
}
```

Пересечение луча и треугольника

- Свой луч, свой треугольник

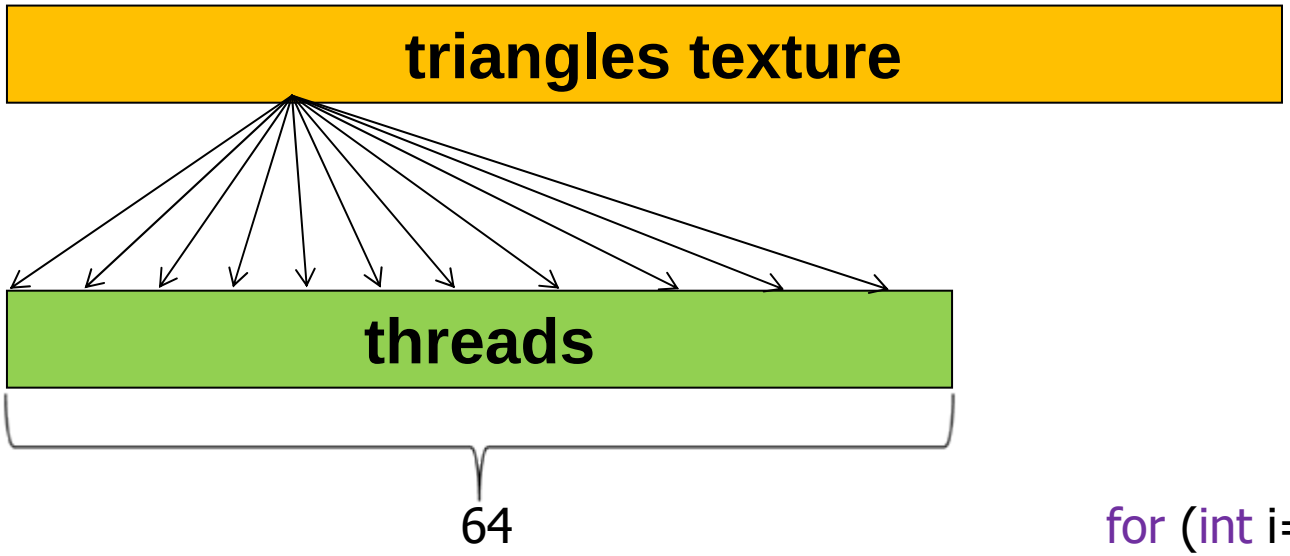


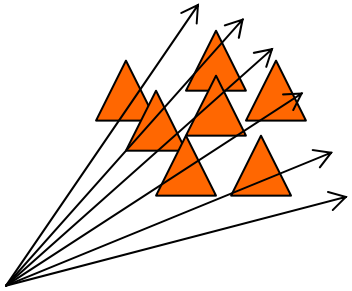


```
for (int i=0;i<triNum;i++)
{
   (A,B,C) = tex1Dfetch(tex,i);
   // intersection code
}
```

Пересечение луча и треугольника

- Свой луч, свой треугольник

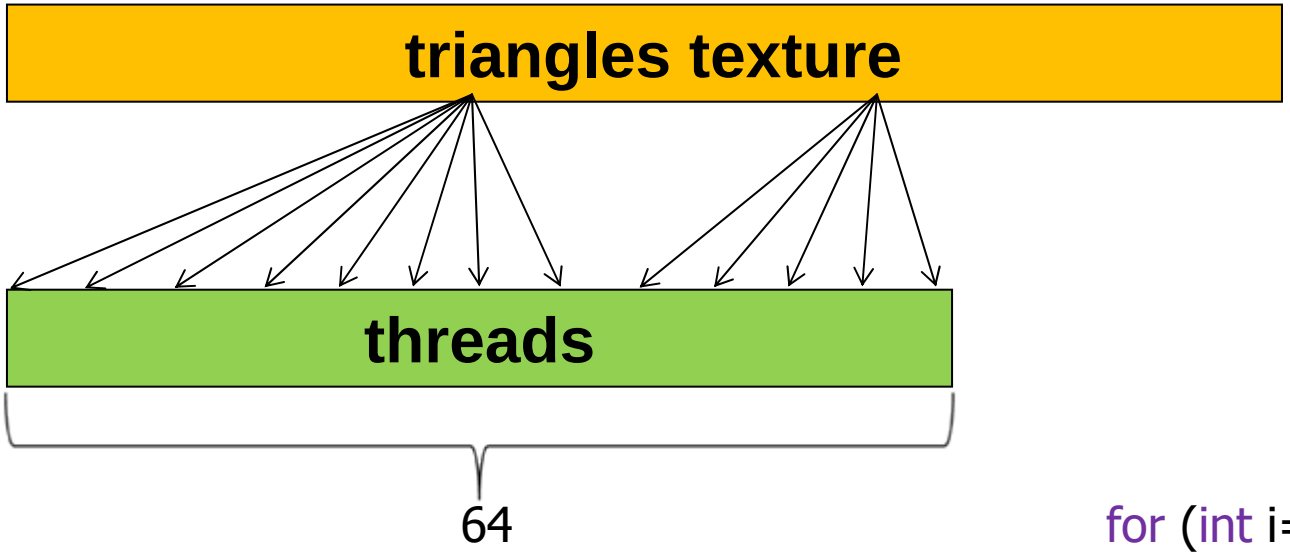


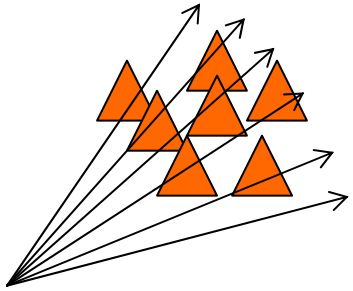


```
for (int i=0;i<triNum;i++)
{
   (A,B,C) = tex1Dfetch(tex,i);
   // intersection code
}
```

Пересечение луча и треугольника

- Свой луч, свой треугольник

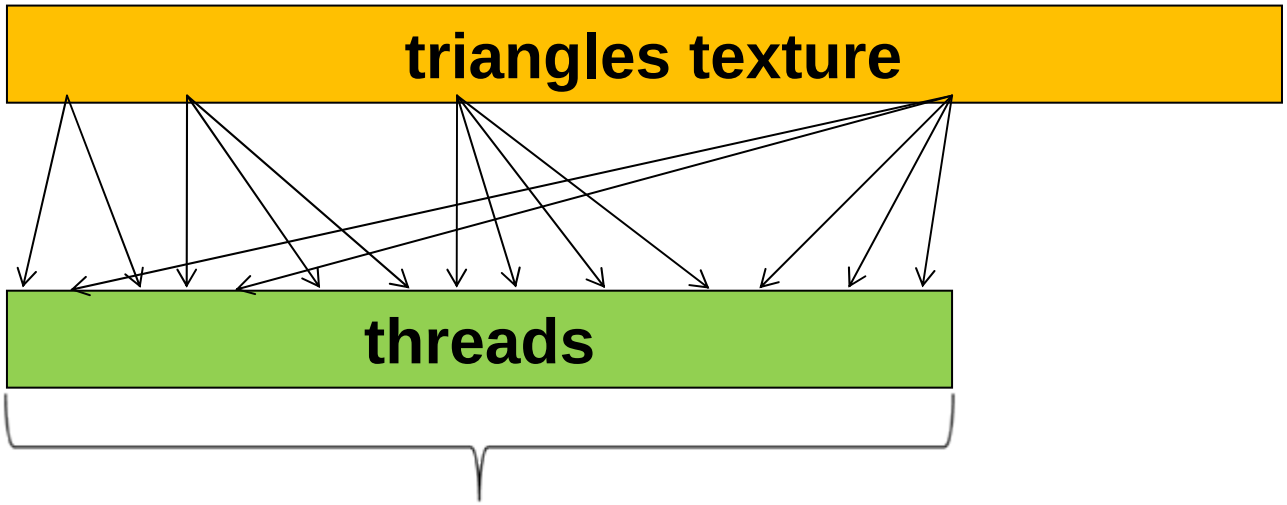


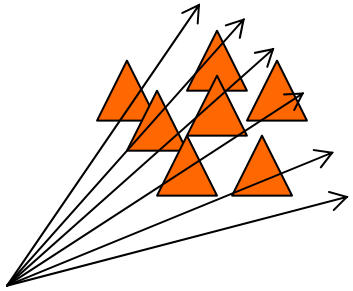


```
for (int i=0;i<triNum;i++)
{
   (A,B,C) = tex1Dfetch(tex,i);
   // intersection code
}
```

Пересечение луча и треугольника

- Свой луч, свой треугольник



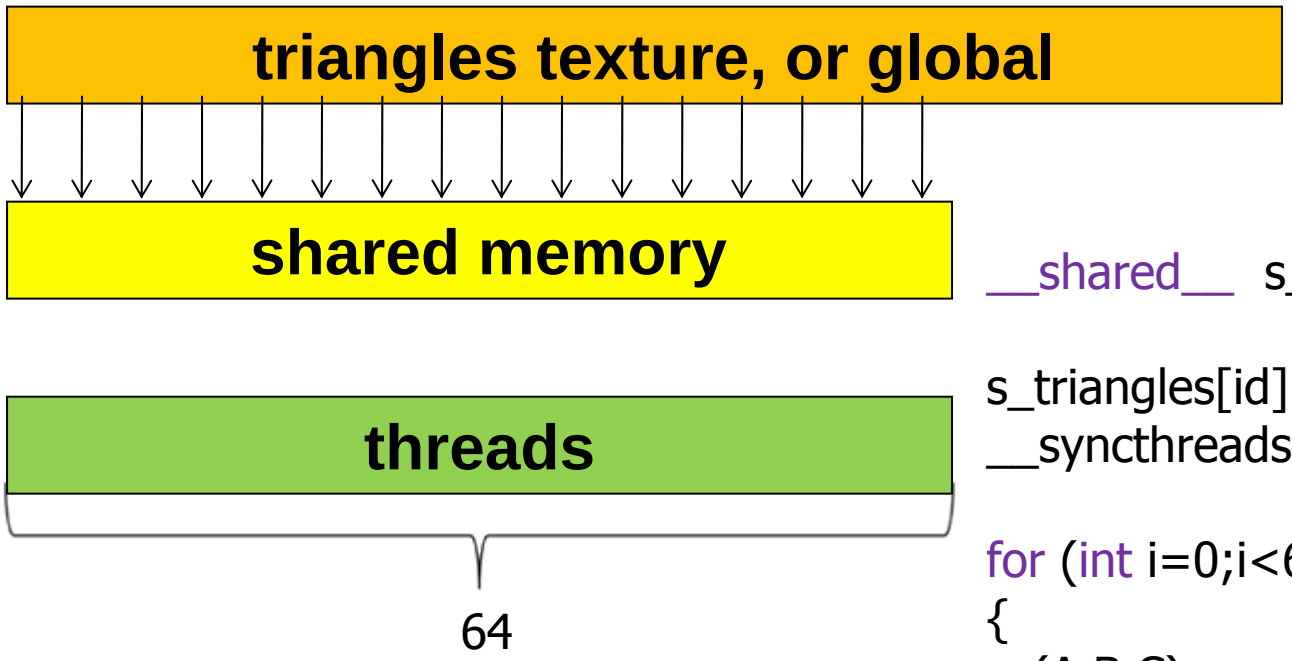


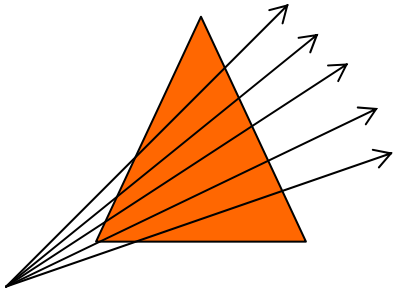
```
for (int i=0;i<triNum;i++)
{
   (A,B,C) = tex1Dfetch(tex, i+offset);
   // intersection code
}
```

Ядро занимает 32 регистра

Пересечение луча и треугольника

- Свой луч, общий треугольник

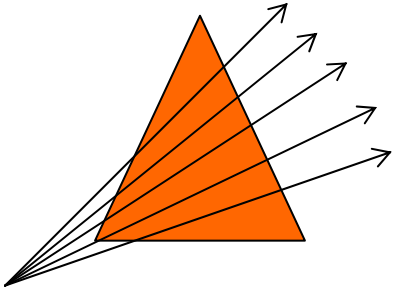


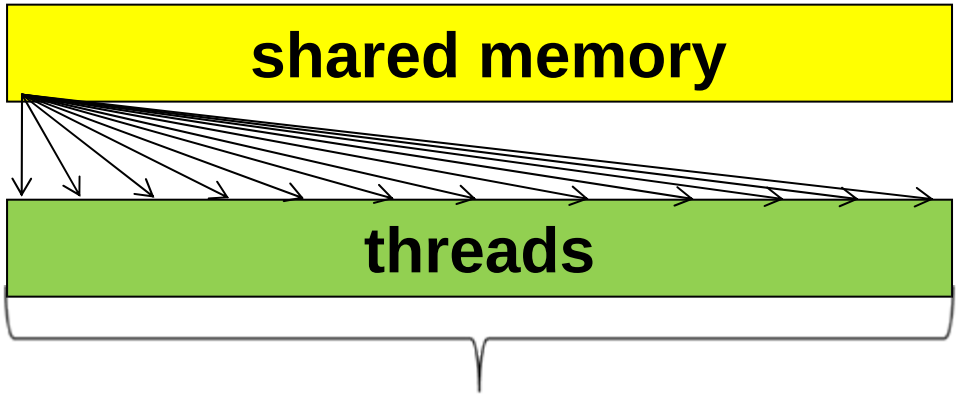


```
__shared__  s_triangles[64];

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

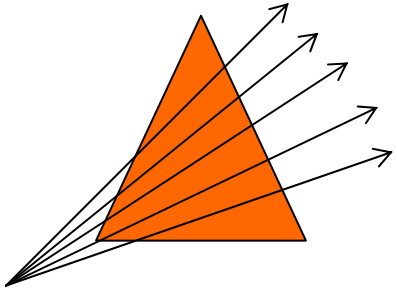


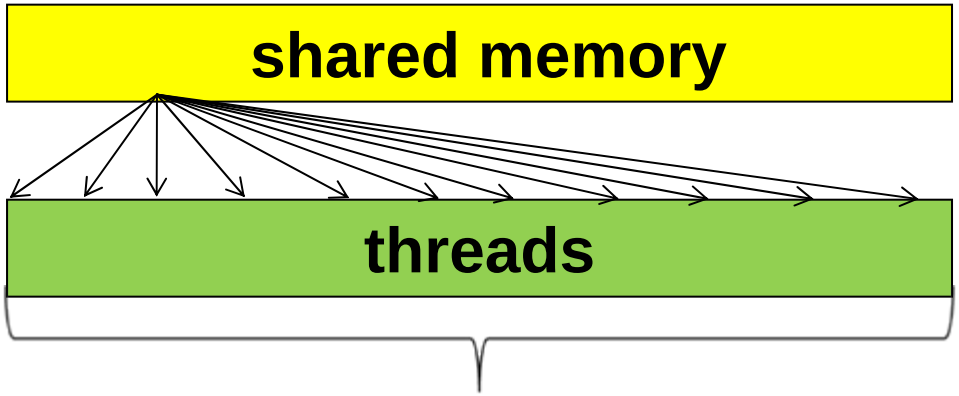



```
__shared__  s_triangles[64];

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



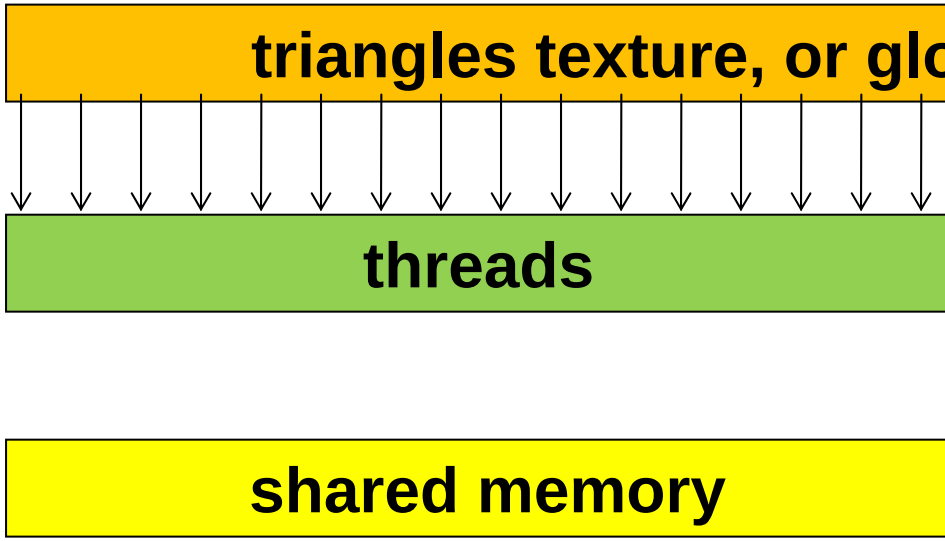


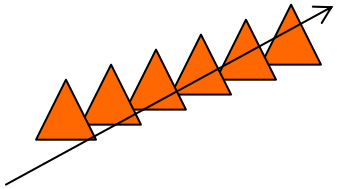

```
__shared__  s_triangles[64];

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

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

Ядро занимает 20 регистров

Пересечение луча и треугольника

- Общий луч, свой треугольник

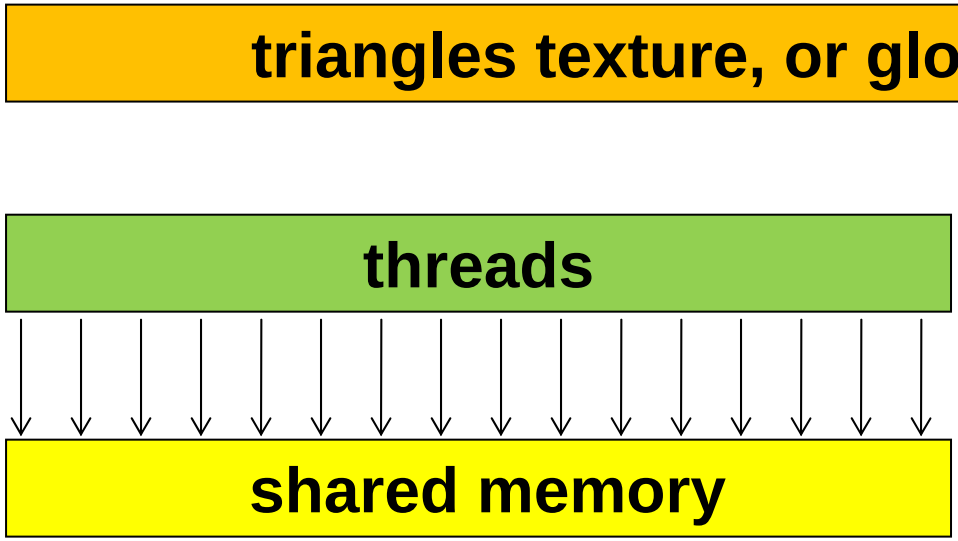


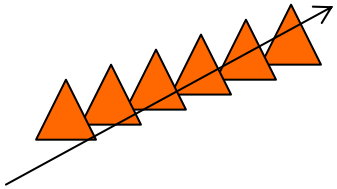


```
__shared__ Hit  s_tResult[64];
__shared__ Ray ray;

(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



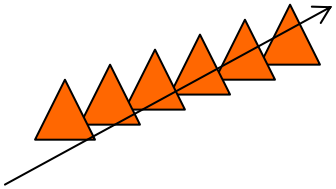


```
__shared__ Hit  s_tResult[64];
__shared__ Ray ray;

(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__ 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
```

shared memory

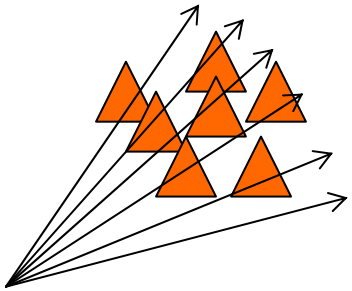
ReduceMin

Ядро занимает 21 регистр

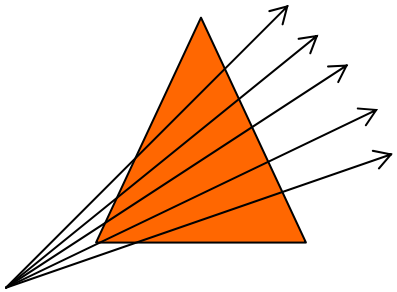
Пересечение луча и треугольника

- Организация сетки - блоки 8x8
- Что общее для нитей блока?

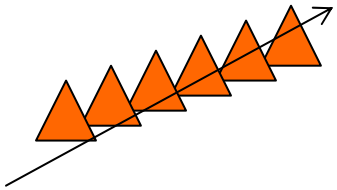
1. Свой луч, свой треугольник (100%)
2. Свой луч, общий треугольник (130%)
3. Общий луч, свой треугольник (40-50%)



32 регистра



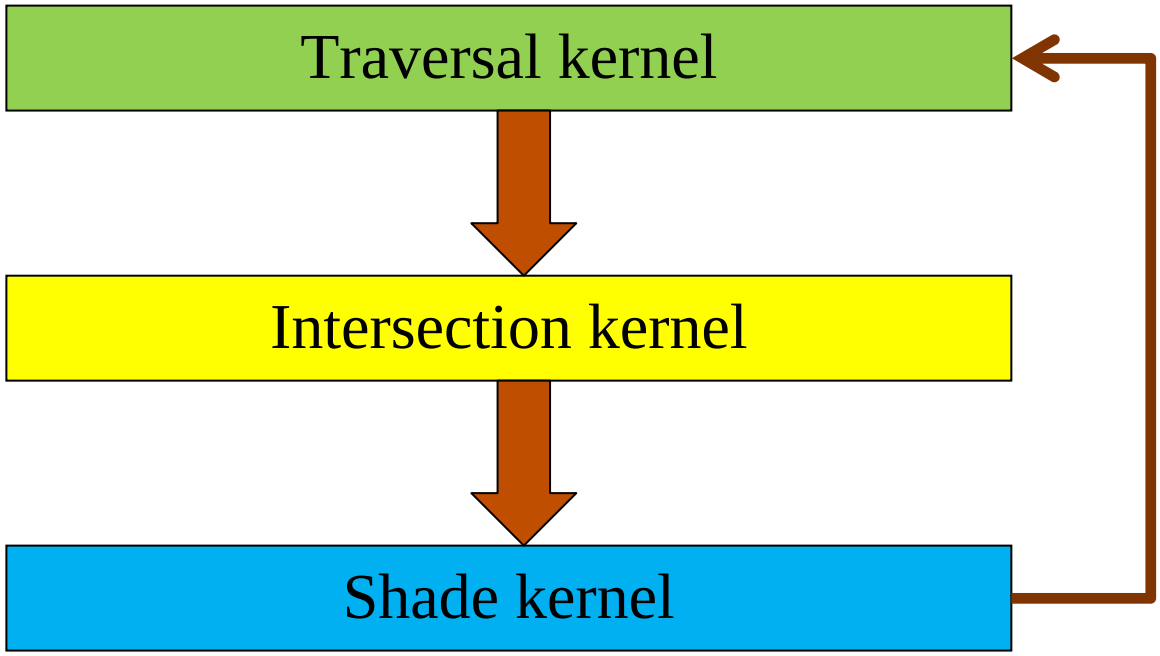
20 регистров



21 регистр

Архитектура рейтрейсера

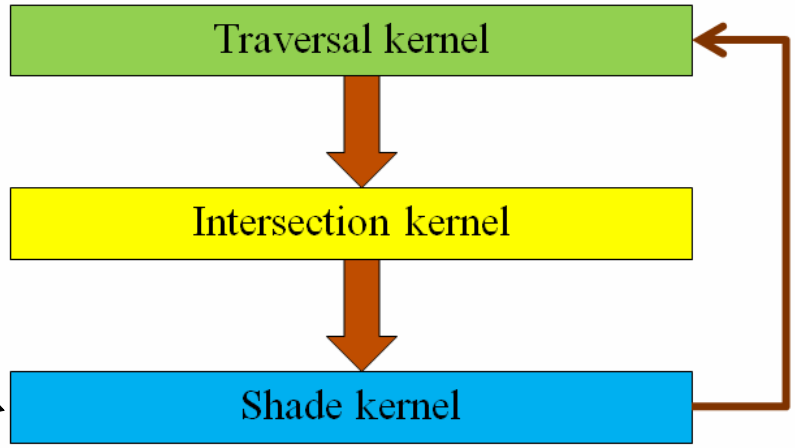
- Ядро пересечений – 32 регистра
- Нужно разбить алгоритм трассировки на несколько ядер



Архитектура рейтрейсера

- Traversal kernel – блоки 16x4
- Как хранить геометрию?

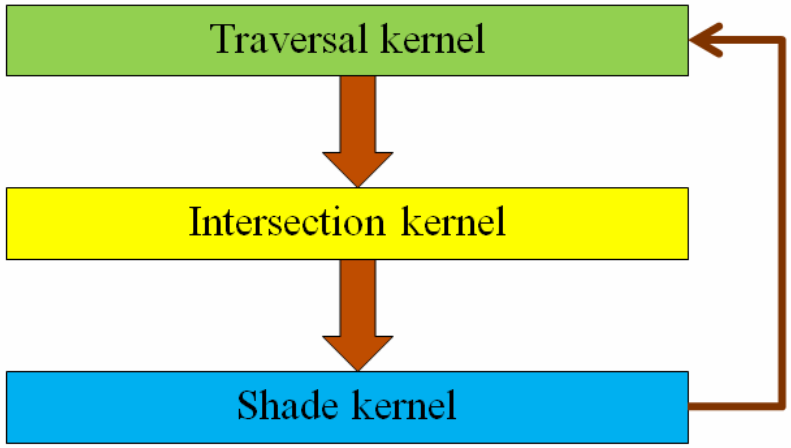
```
struct Vertex
{
    float3 pos[3];
    float3 norm[3];
    float2 texCoord;
    uint materialIndex;
};

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



Архитектура рейтрейсера

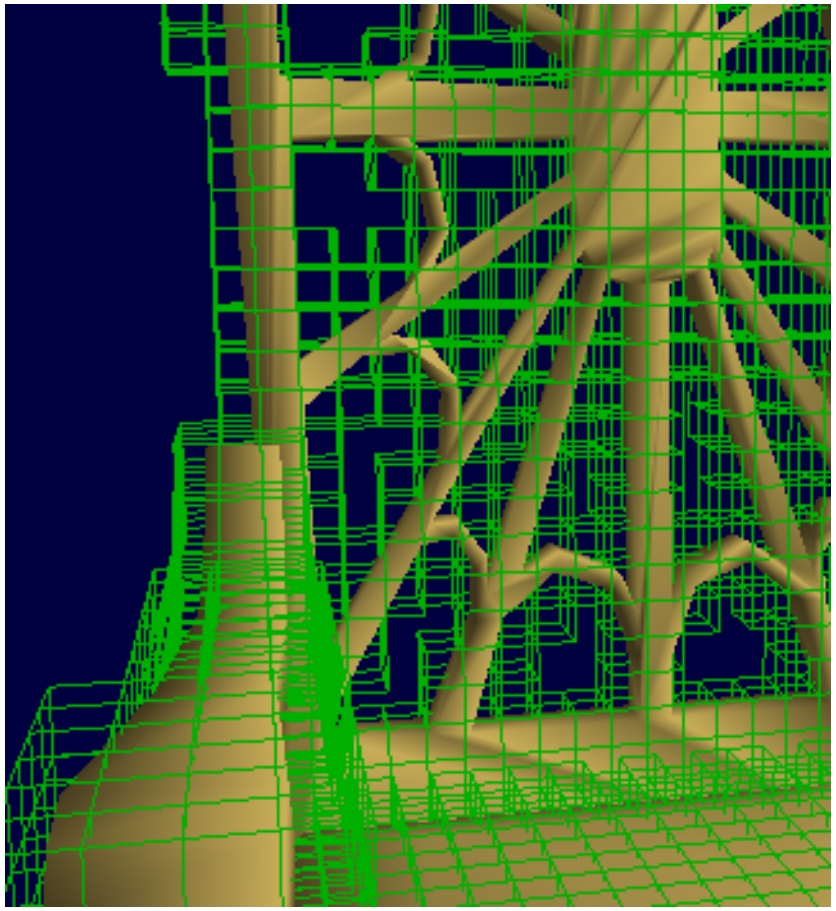
- Traversal kernel – блоки 16x4
- Дублирование геометрии

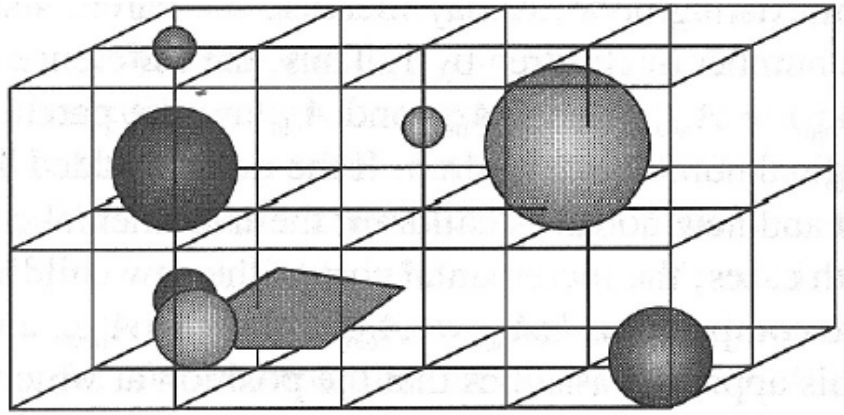
```
struct Triangle
{
   vec3f v[3];
   unsigned int selfIndex;
}; // 40

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

// выборки по float2
```

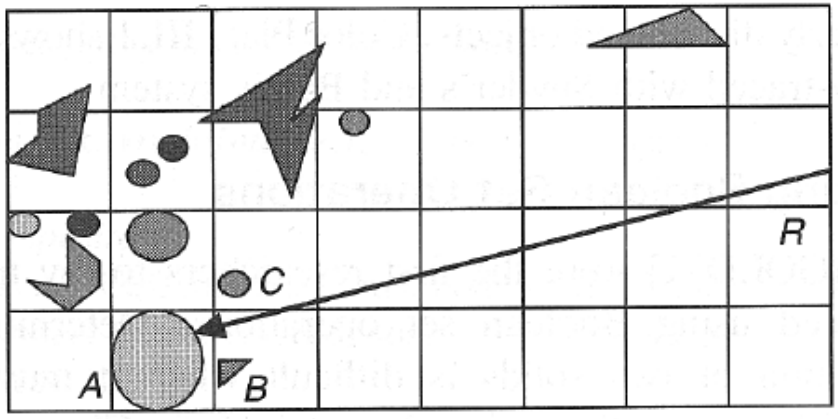



```
struct Hit
{
    float t;
    unsigned int objectIdAndType;
};
```

Регулярная сетка



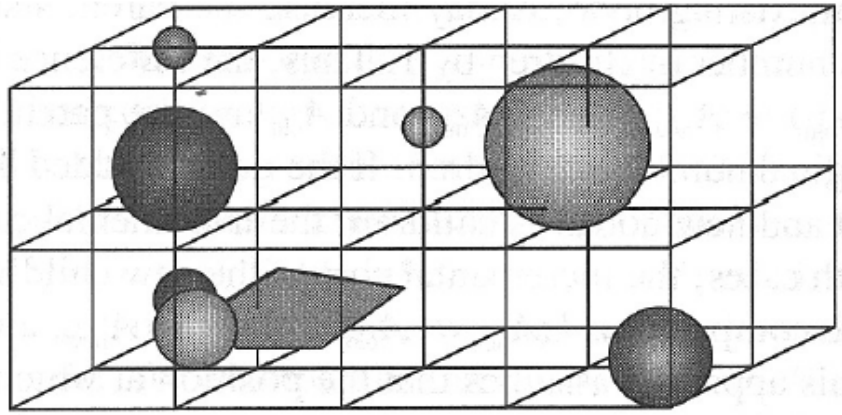


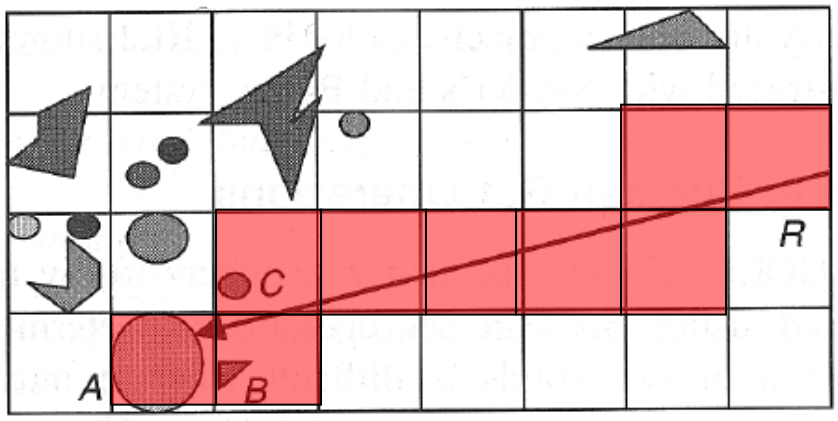


Регулярная сетка

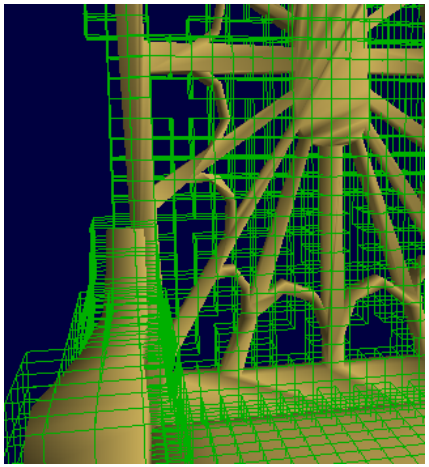
- Регулярная сетка

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





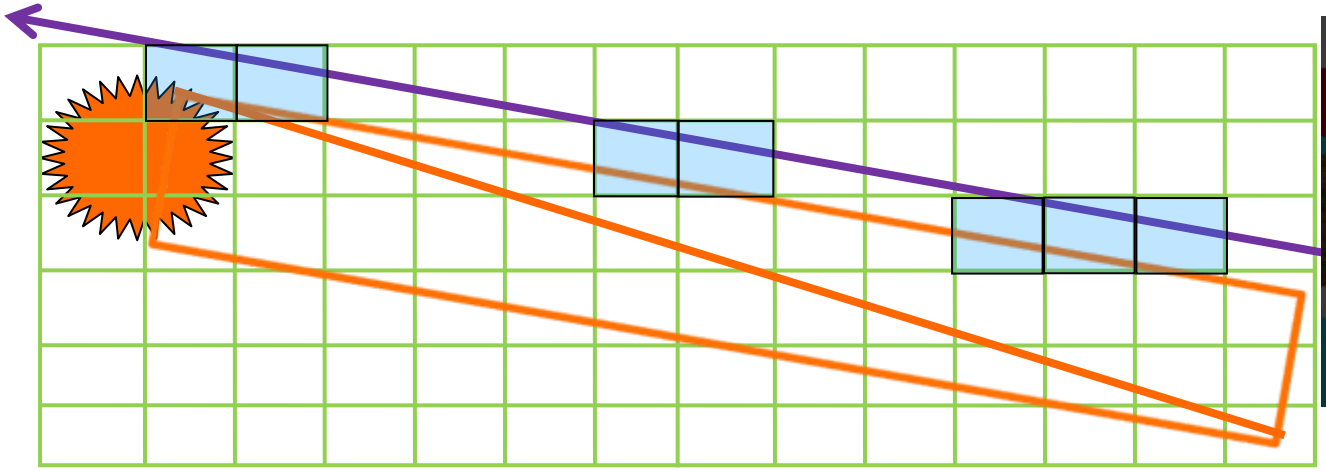
Регулярная сетка

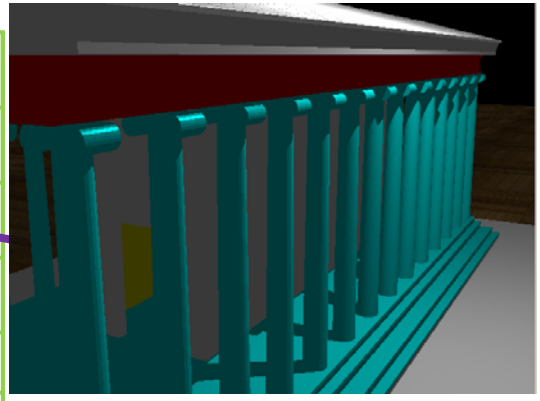


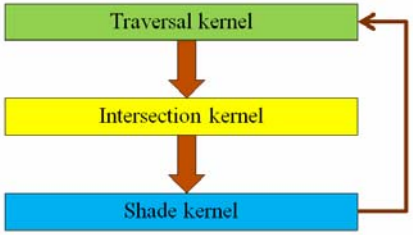
- Преимущества
  - Просто и быстро строится
  - Простой алгоритм траверса
- Недостатки
  - Плохо справляется с пустым пространством
  - Требует много памяти
  - Много повторных пересечений – **отвратительно** разбивает геометрию
- Только для небольших сцен (1-50K)

Регулярная сетка

- Почему сетка плохо разбивает геометрию?



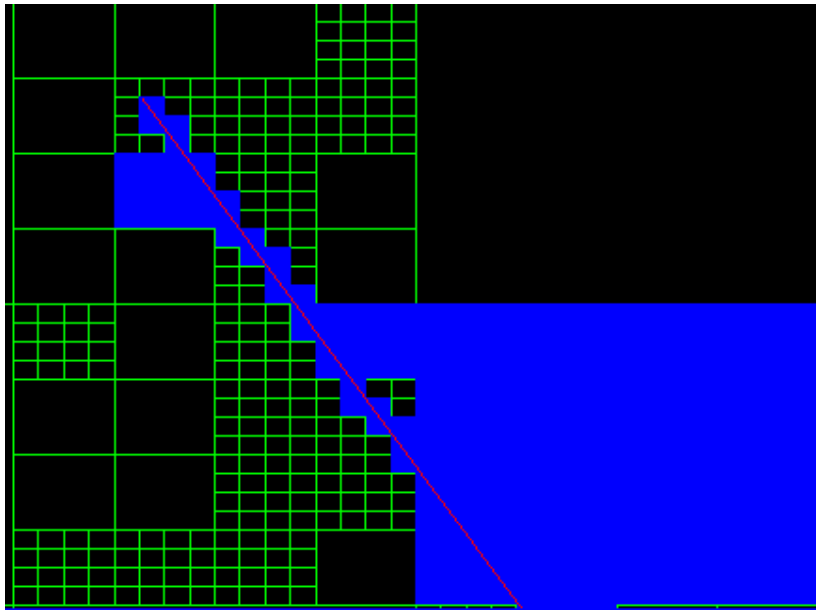


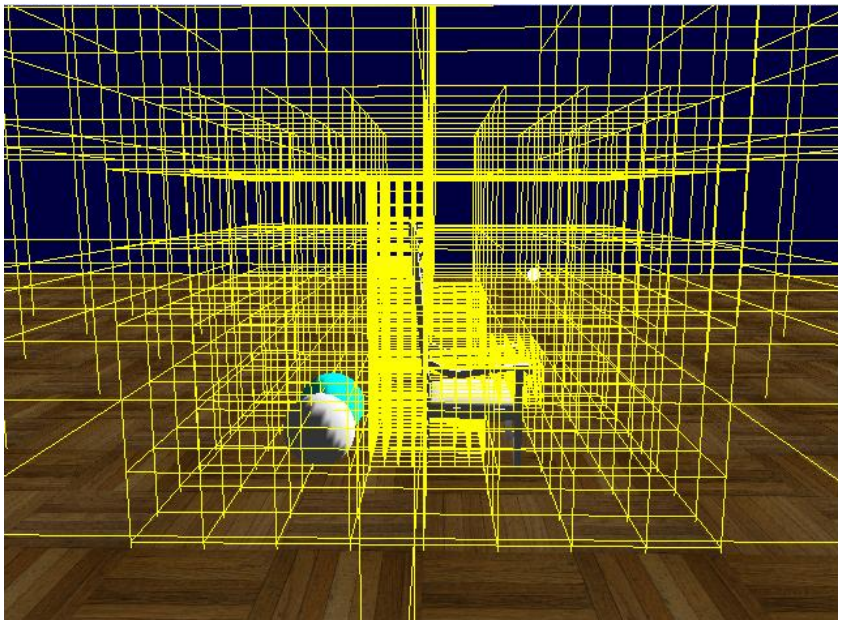


- Перебрали 15 вокселей
- 7 раз посчитали пересечение с одним и тем же треугольником!

Иерархическая сетка

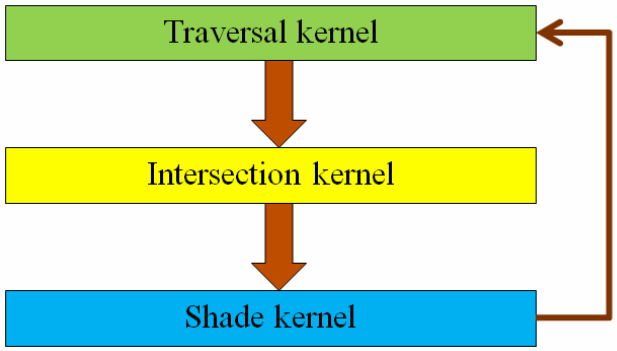
- Небольшое число вокселей
- Рекурсивно разбиваем воксели в местах с плотной геометрией

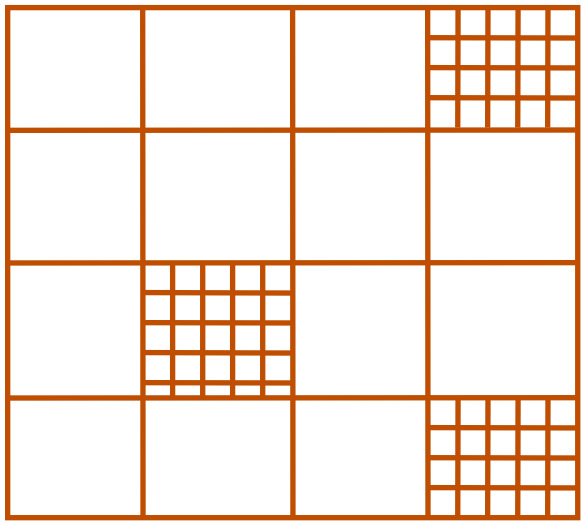


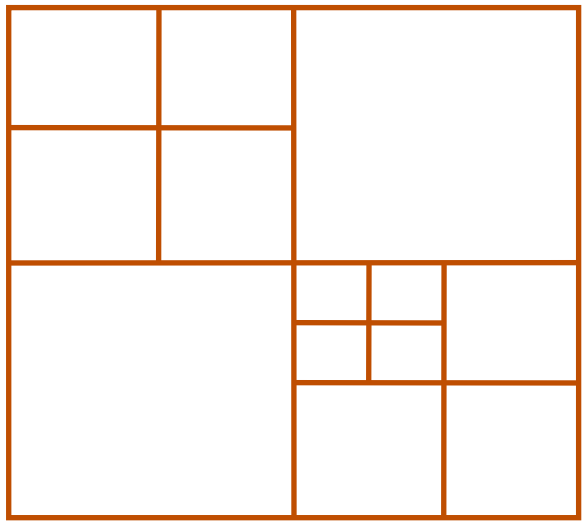


Что дает иерархическая сетка?

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

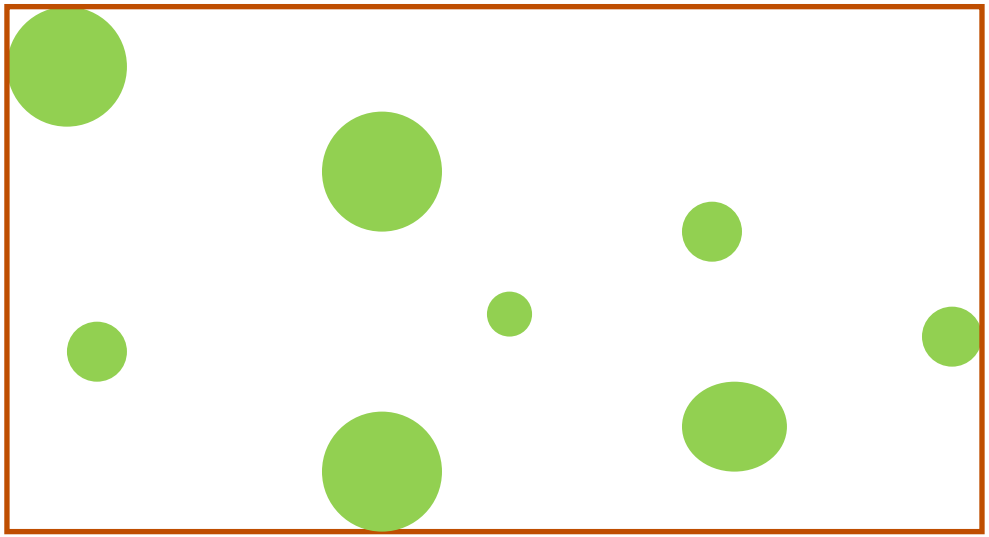






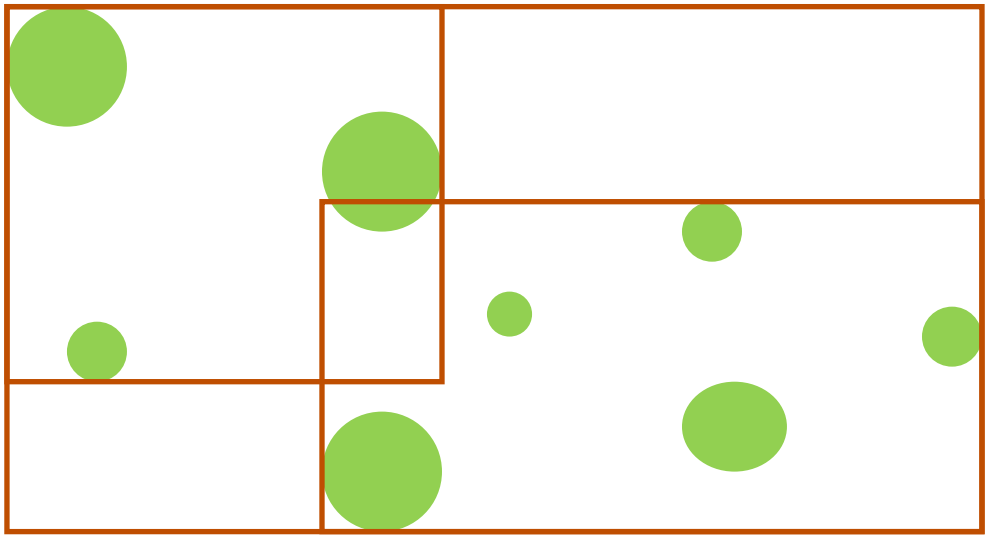
BVH деревья

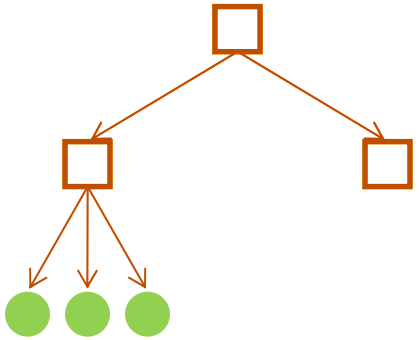
- Bounding Volume Hierarchy



BVH деревья

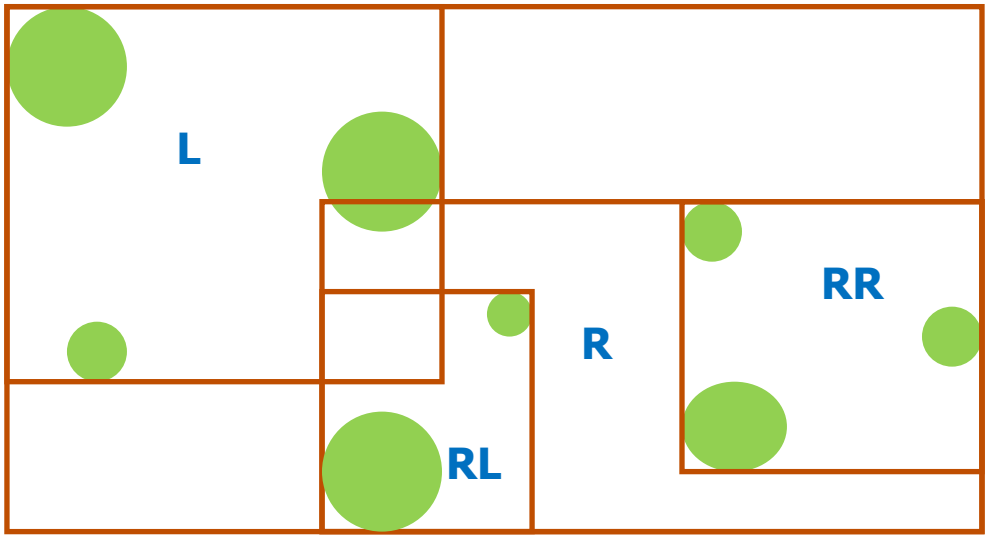
- Bounding Volume Hierarchy

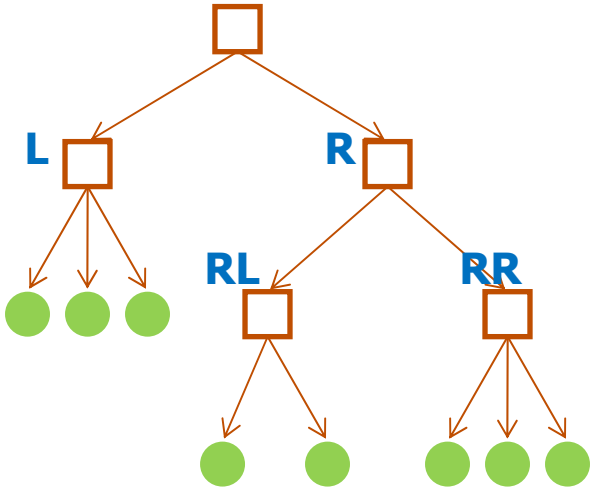




BVH деревья

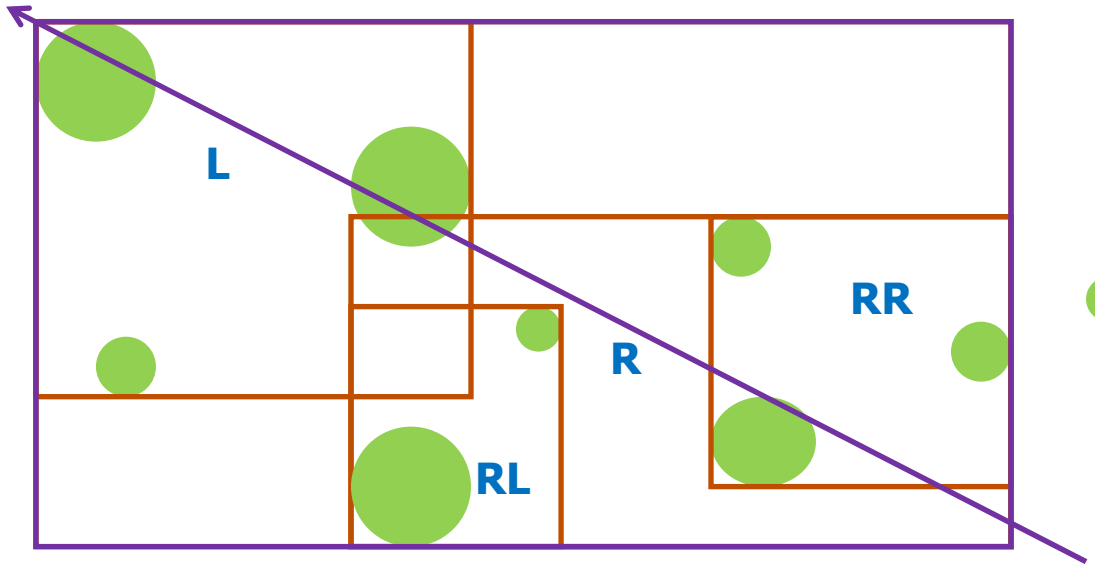
- Bounding Volume Hierarchy

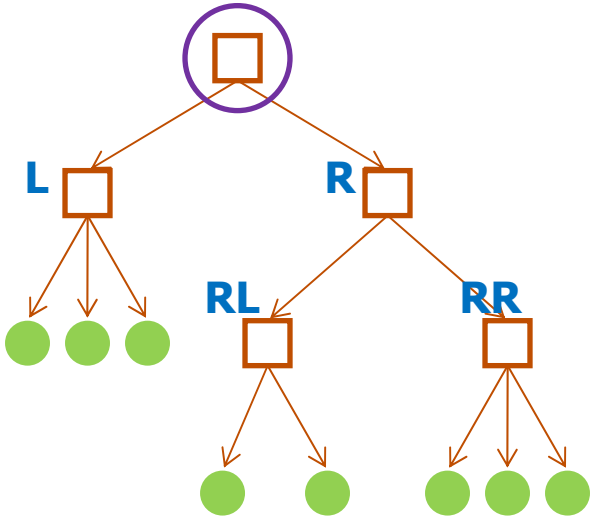




BVH деревья

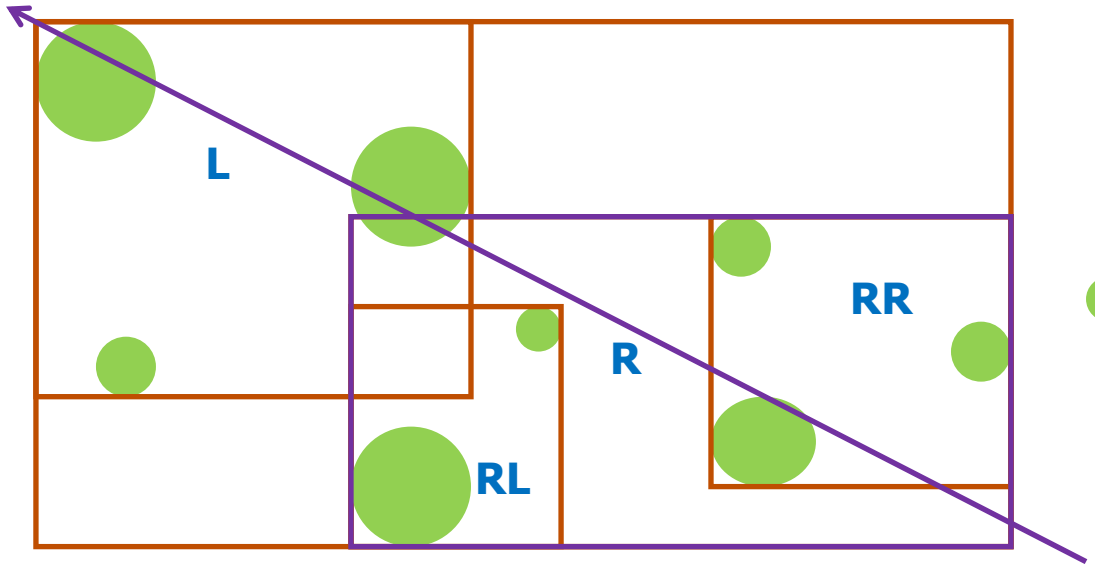
- Траверс на CPU

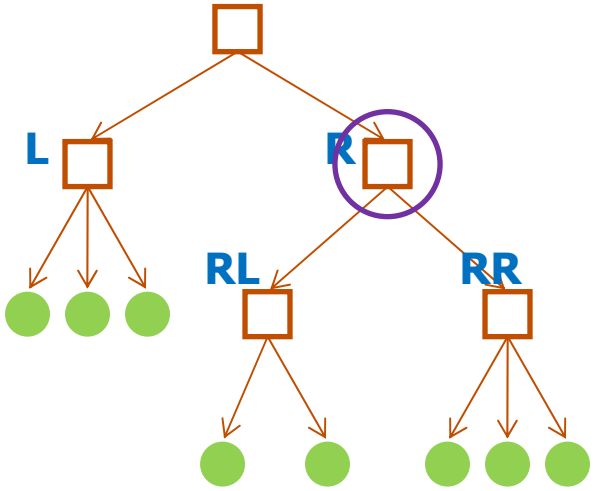




BVH деревья

- Траверс на CPU

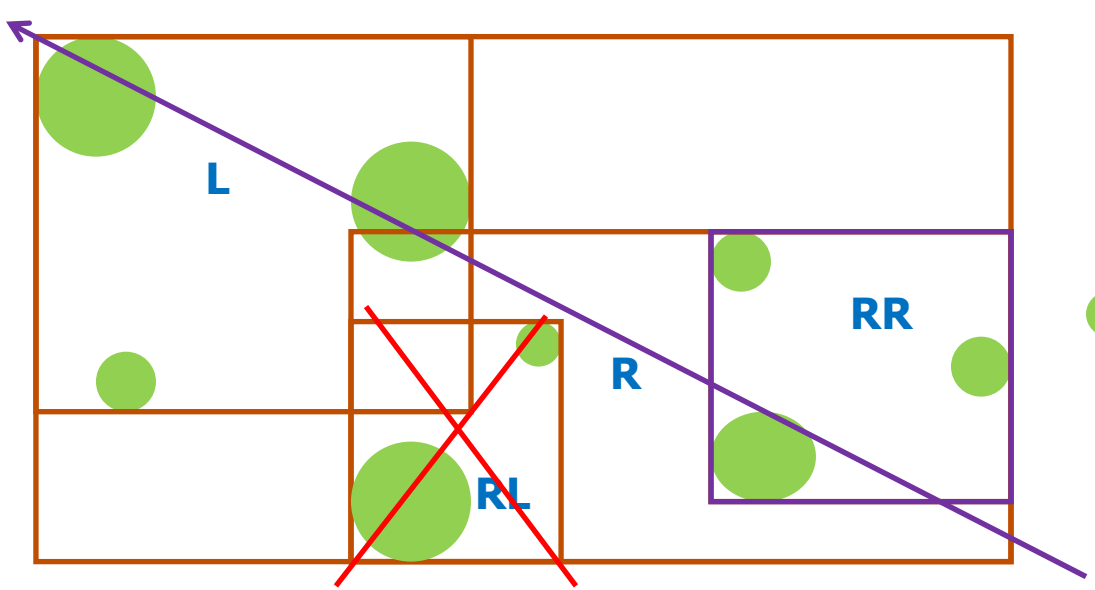


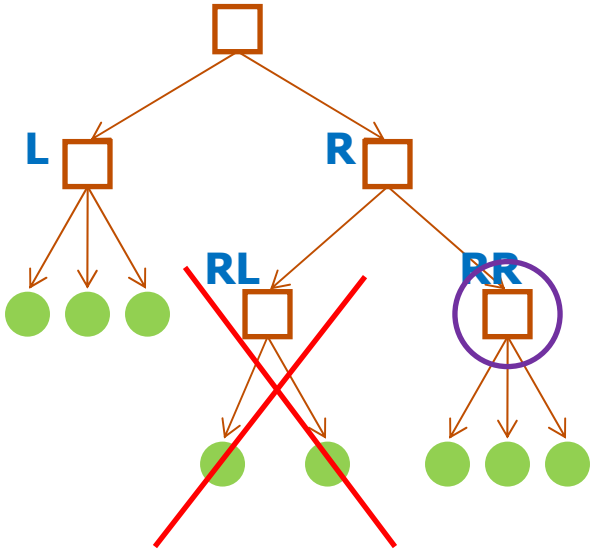


Стек: L

BVH деревья

- Траверс на CPU

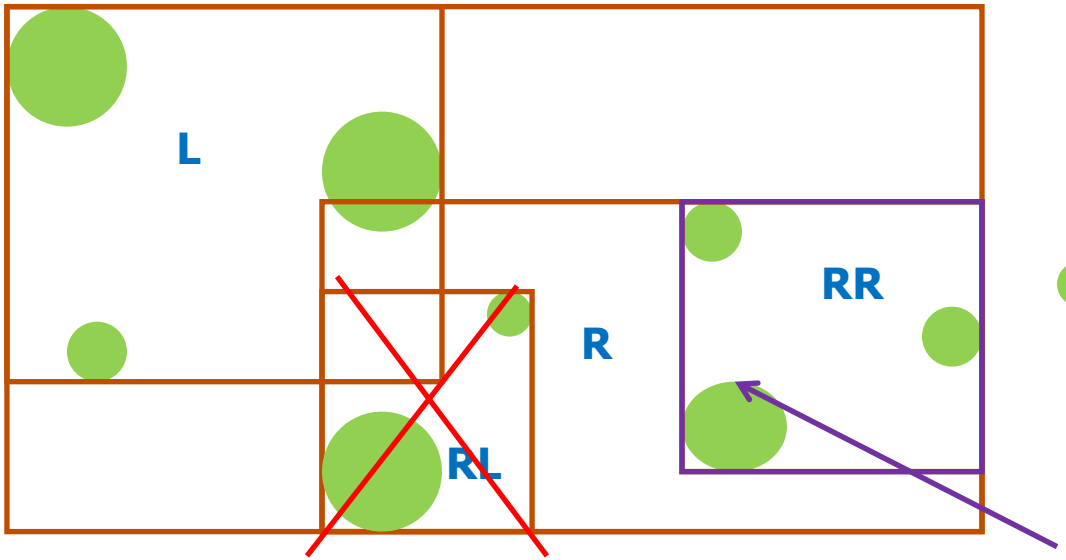


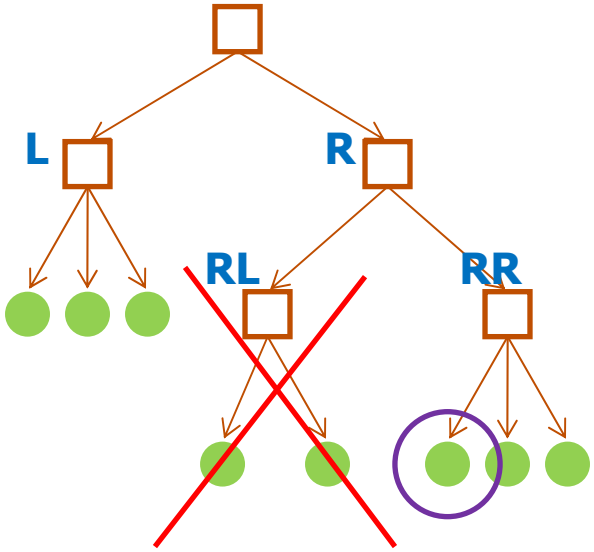


Стек: L

BVH деревья

- Траверс на CPU

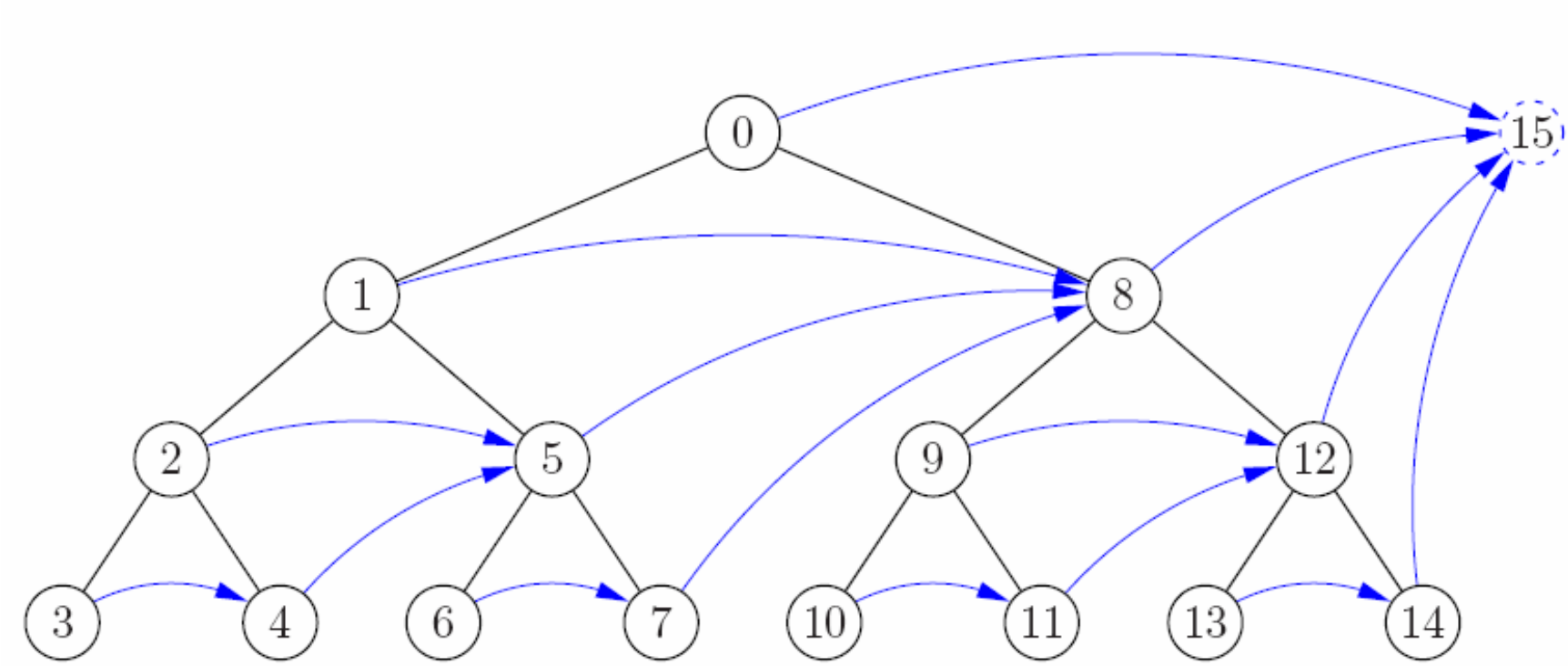




Стек: L

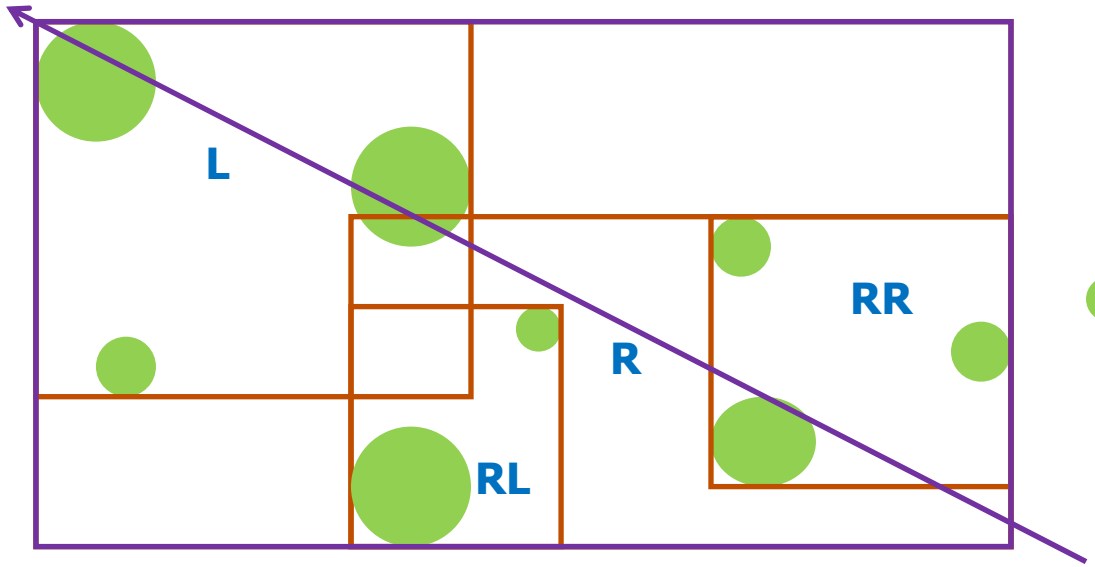
BVH деревья

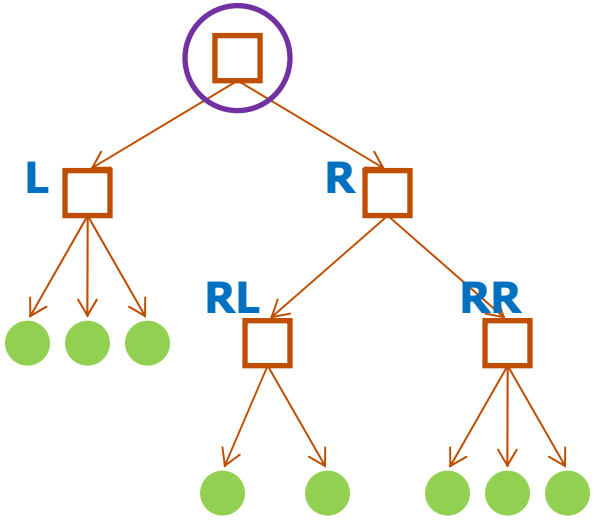
- Траверс на CPU



BVH деревья

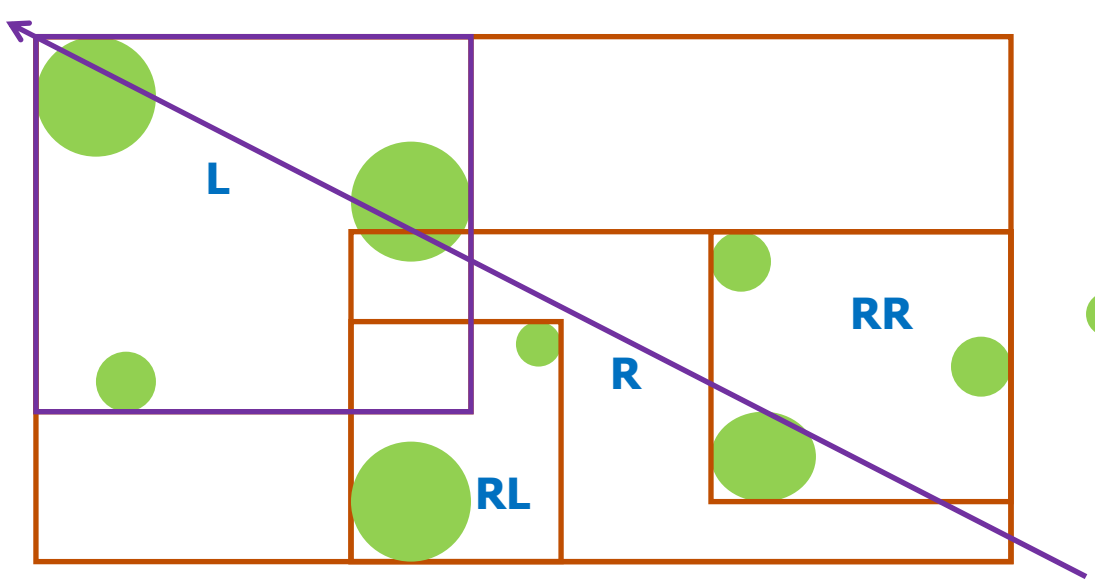
- Траверс на CPU

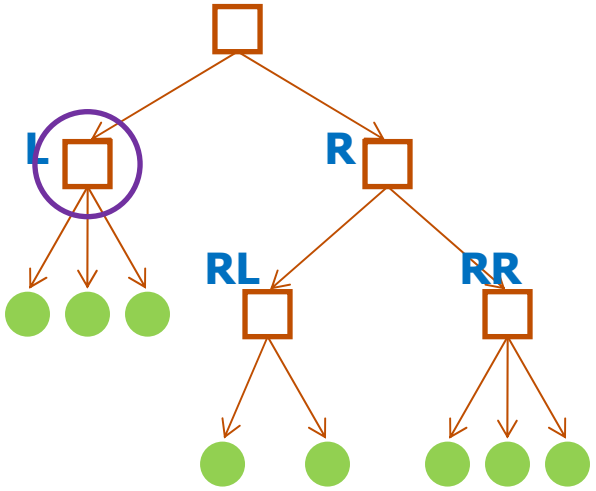




BVH деревья

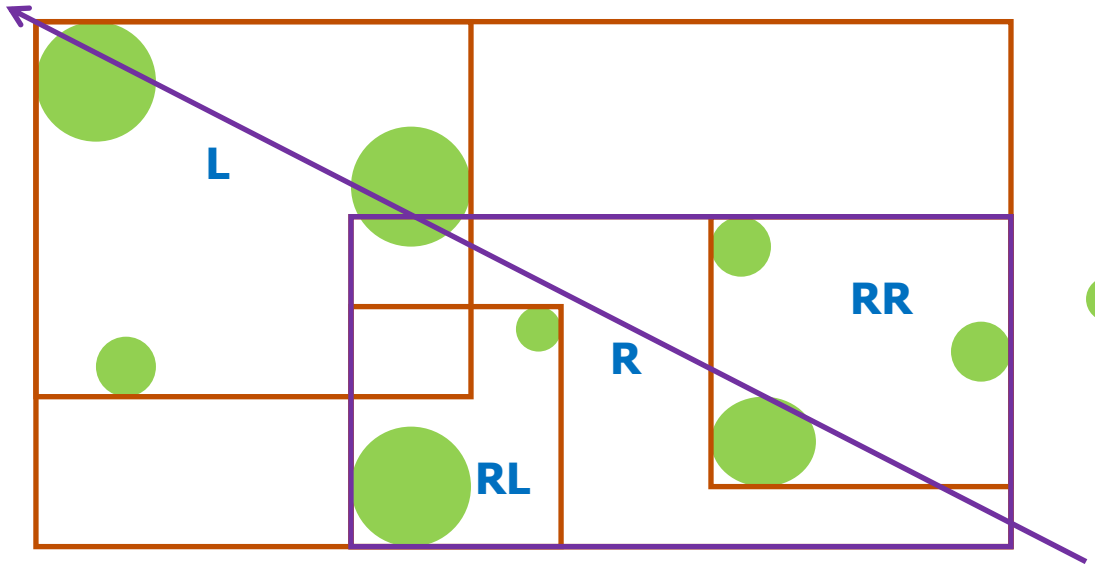
- Траверс на CPU

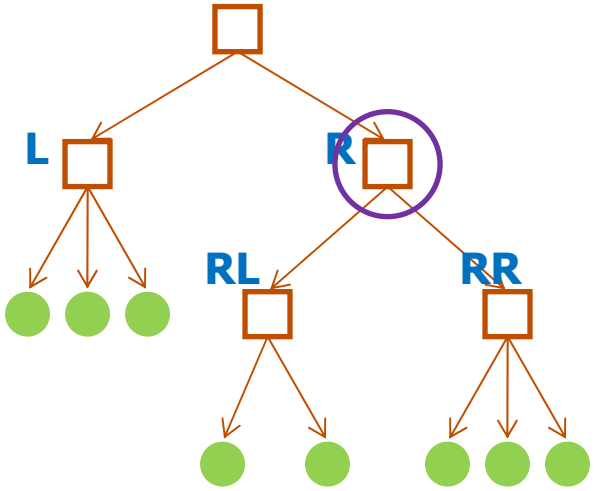




BVH деревья

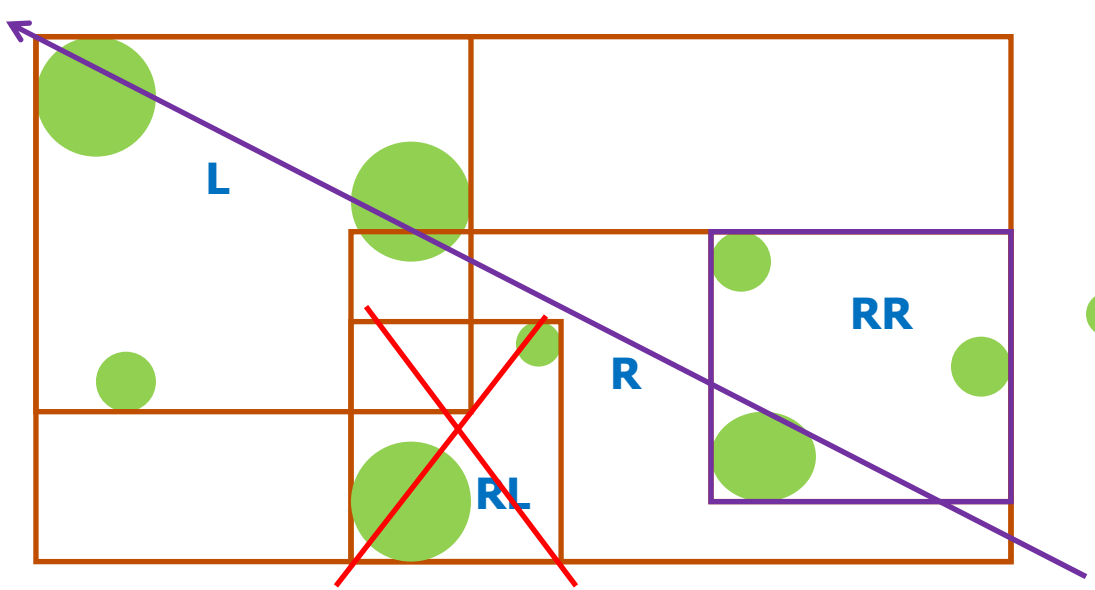
- Траверс на CPU

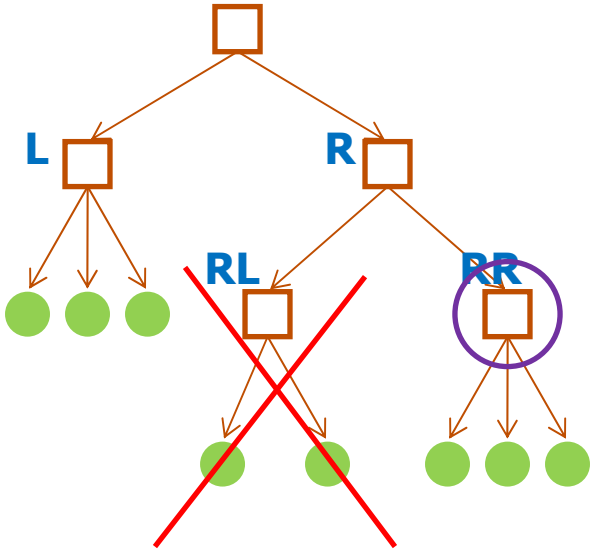




BVH деревья

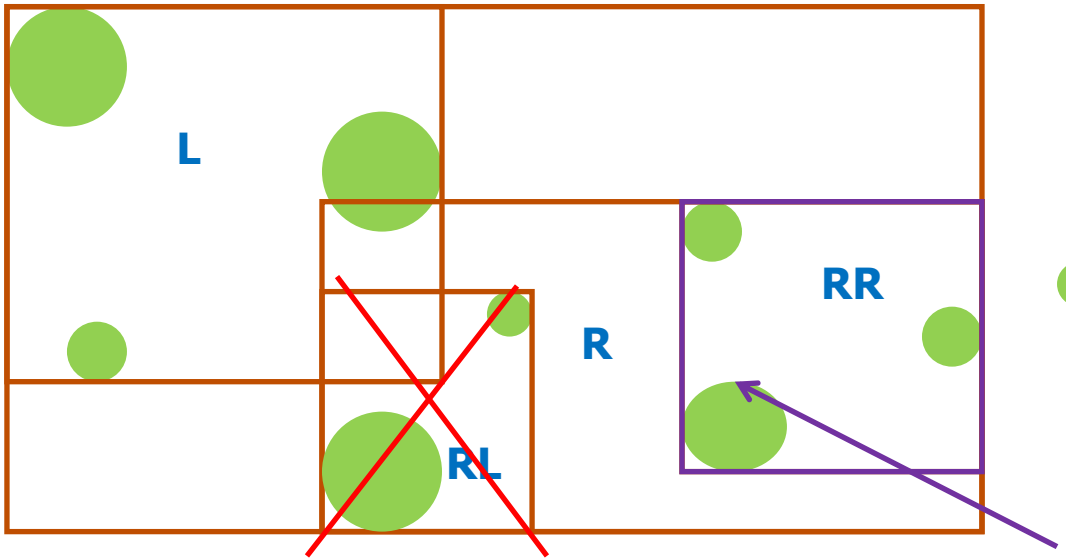
- Траверс на CPU

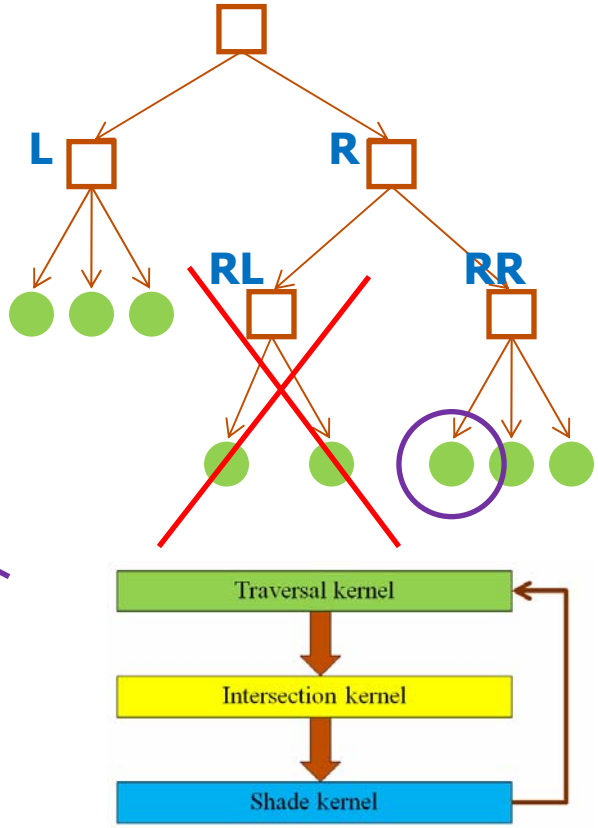




BVH деревья

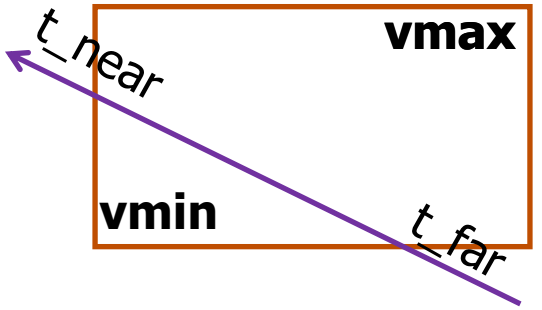
- Траверс на CPU

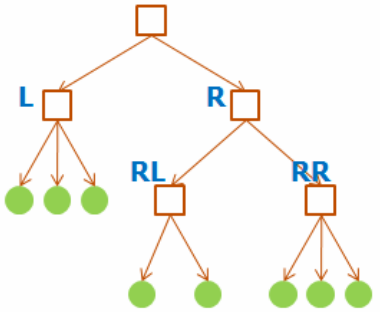




BVH деревья

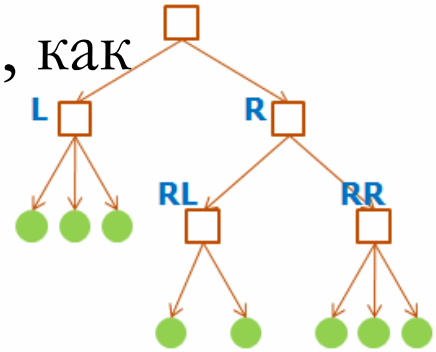
- **Как делать на CUDA?**
  - Луч – 6 регистров
  - Бокс – 6 регистров
  - t_near, t_far, - 2
  - nodeOffset, leftOffset, tid – 3
- **Пересечение луча с боксом**
  - Минимум по всем 6 плоскостям
    - (vmin[0] + rInv.pos[0]) * rInv.dir[0];



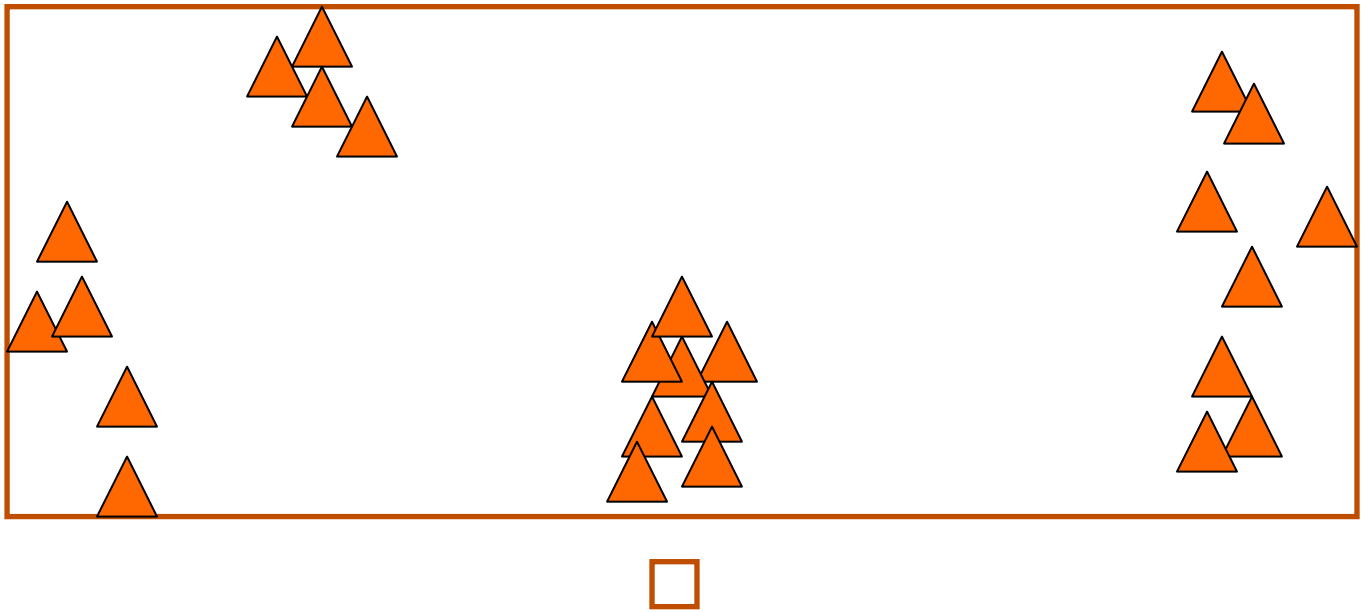


BVH деревья

- **Как делать на CUDA?**
  - 24 mad-а покрывают латентность текстурной памяти
- **Стек на локальной памяти**
  - Локальная память это не так медленно, как может показаться
- **Бесстековый алгоритм**
  - Перебираем массив всегда строго слева направо

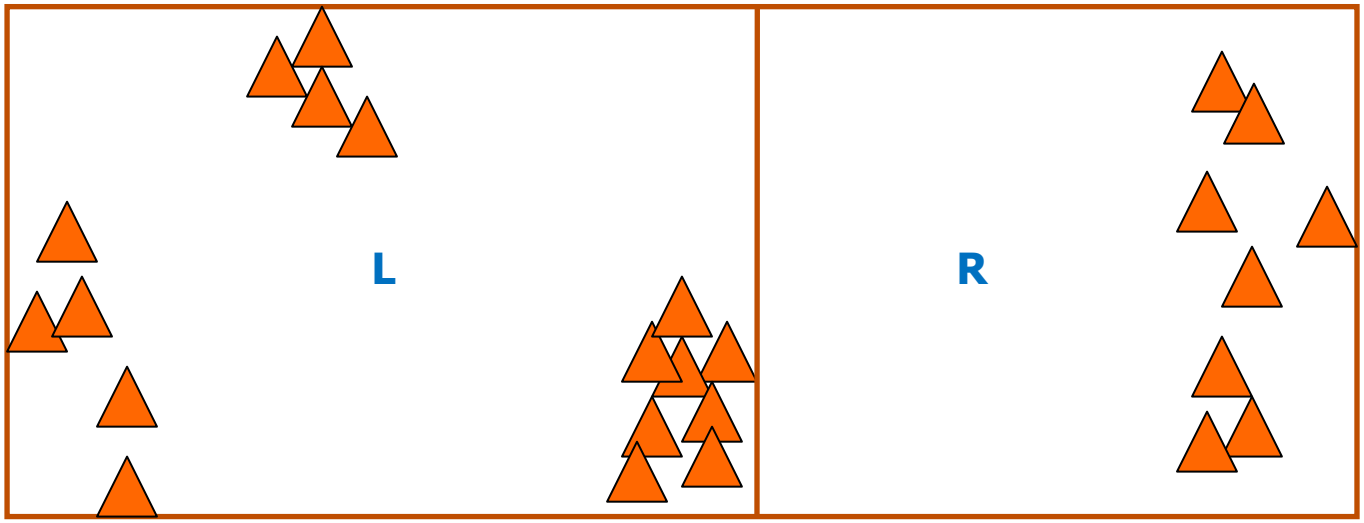


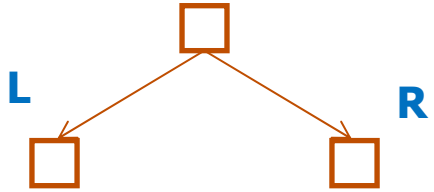
kd-деревья



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

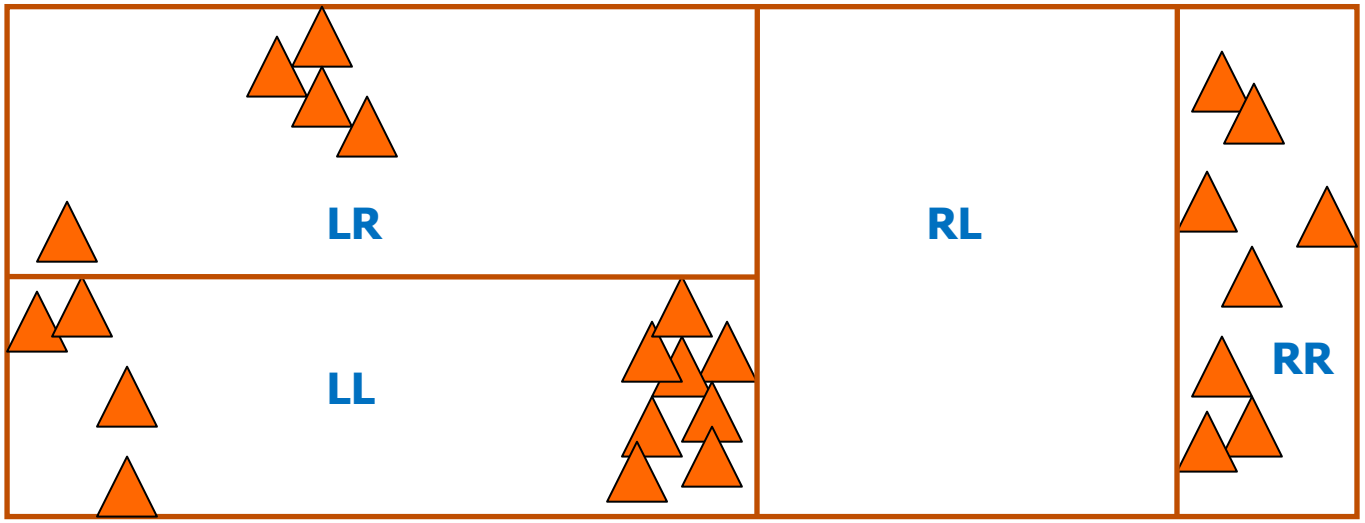
kd-деревья



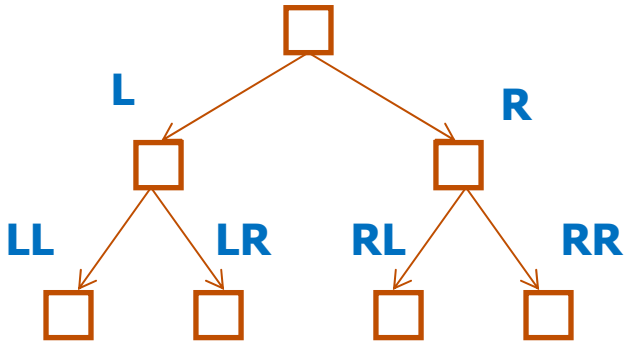



```
struct KdTreeNode
{
    float split;
    uint leftOffset: 29;
    uint splitAxis: 2;
    uint leaf: 1;
};
```

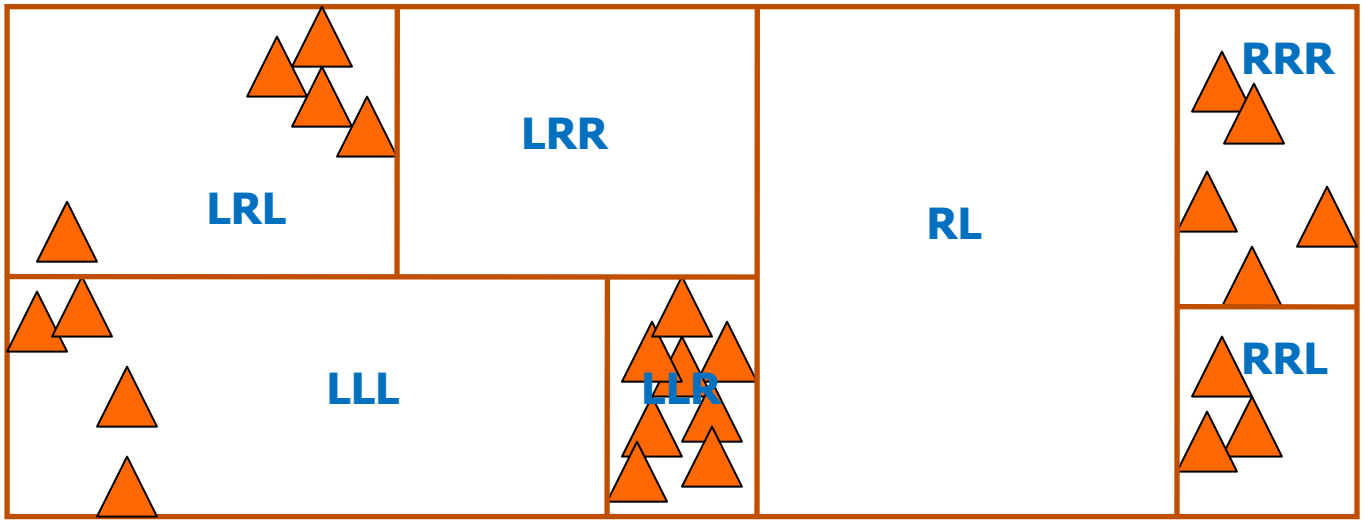
kd-деревья



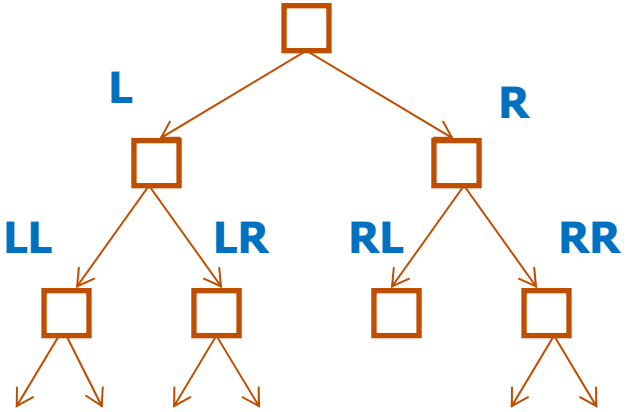

```
struct KdTreeNode
{
    float split;
    uint leftOffset: 29;
    uint splitAxis: 2;
    uint leaf: 1;
};
```



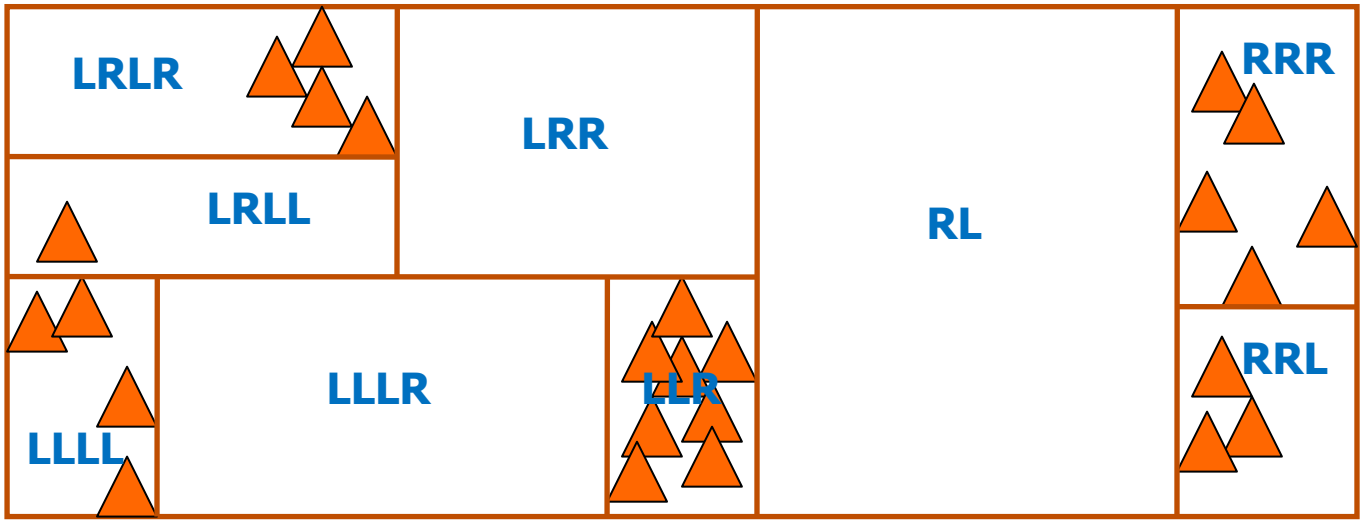
kd-деревья



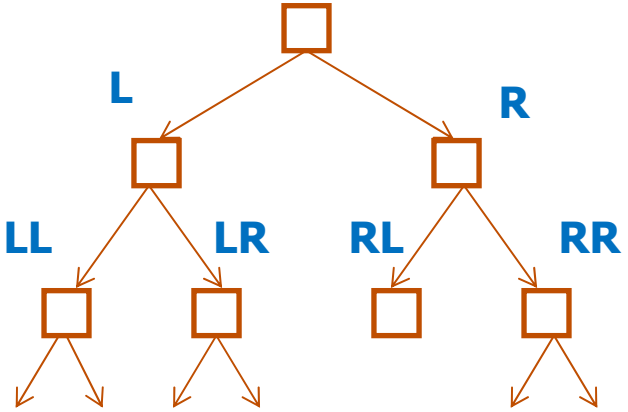

```
struct KdTreeNode
{
    float split;
    uint leftOffset: 29;
    uint splitAxis: 2;
    uint leaf: 1;
};
```



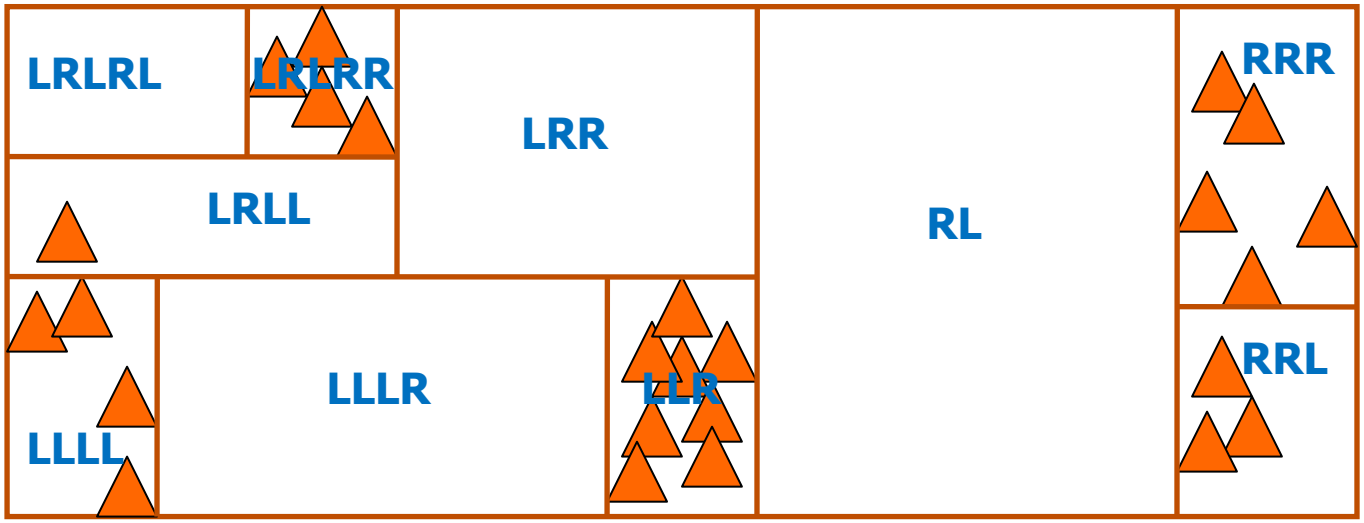
kd-деревья



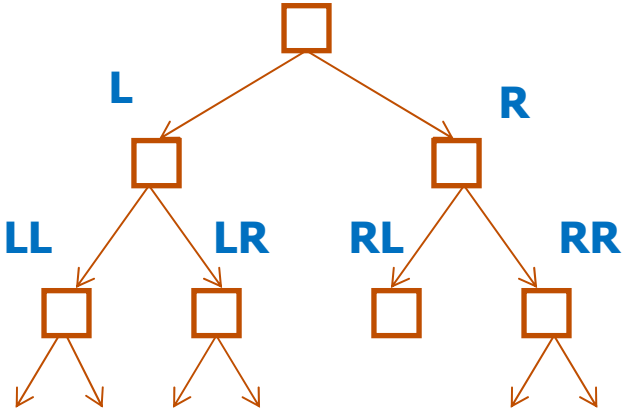

```
struct KdTreeNode
{
    float split;
    uint leftOffset: 29;
    uint splitAxis: 2;
    uint leaf: 1;
};
```



kd-деревья

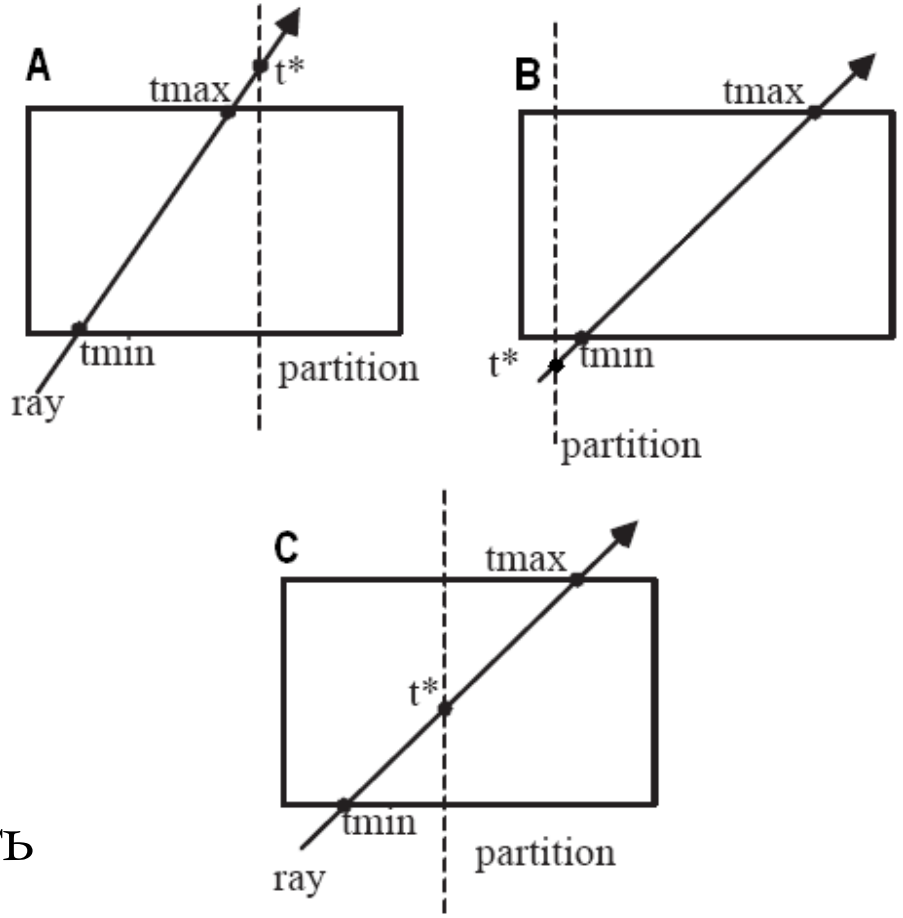



```
struct KdTreeNode
{
    float split;
    uint leftOffset: 29;
    uint splitAxis: 2;
    uint leaf: 1;
};
```



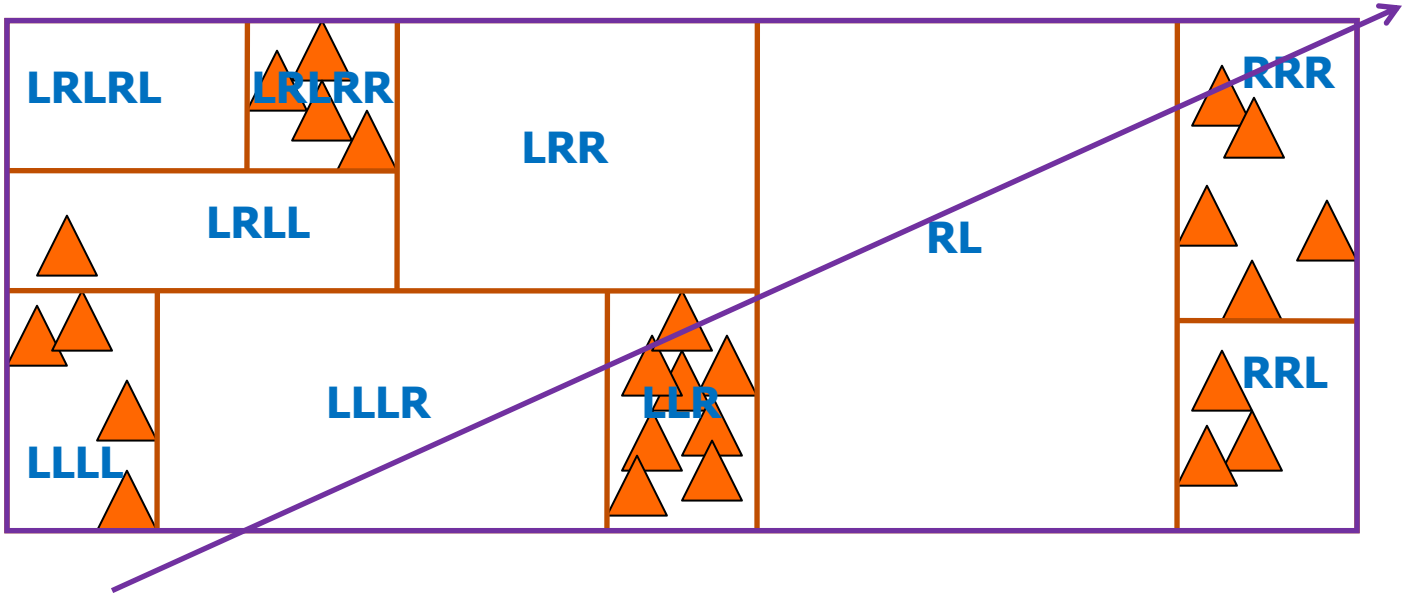
kd-деревья

- **Алгоритм траверса**
- **Регистры – 13 min:**
  - луч - 6
  - t, tmin, tmax – 3
  - node – 2
  - tid, stack_top – 2
  - На практике удалось уложиться в 19
  - Стек: локальная память



kd-деревья

- Алгоритм траверса

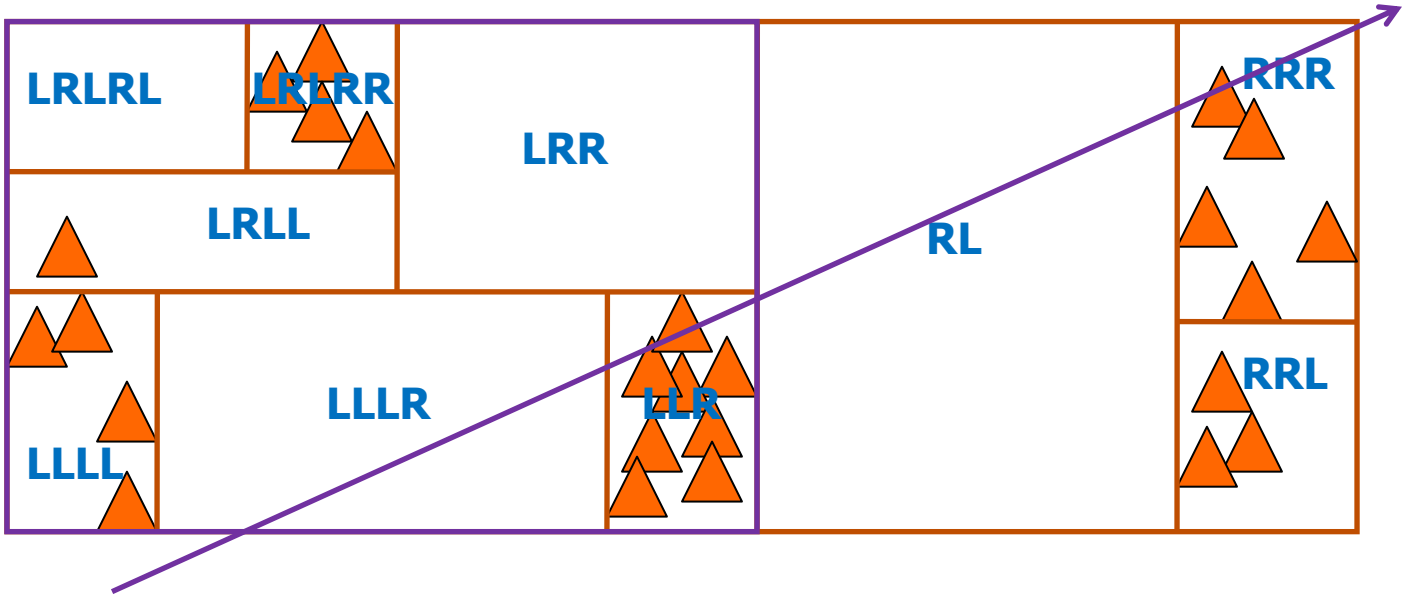


Стек:

Текущий узел:

kd-деревья

- Алгоритм траверса

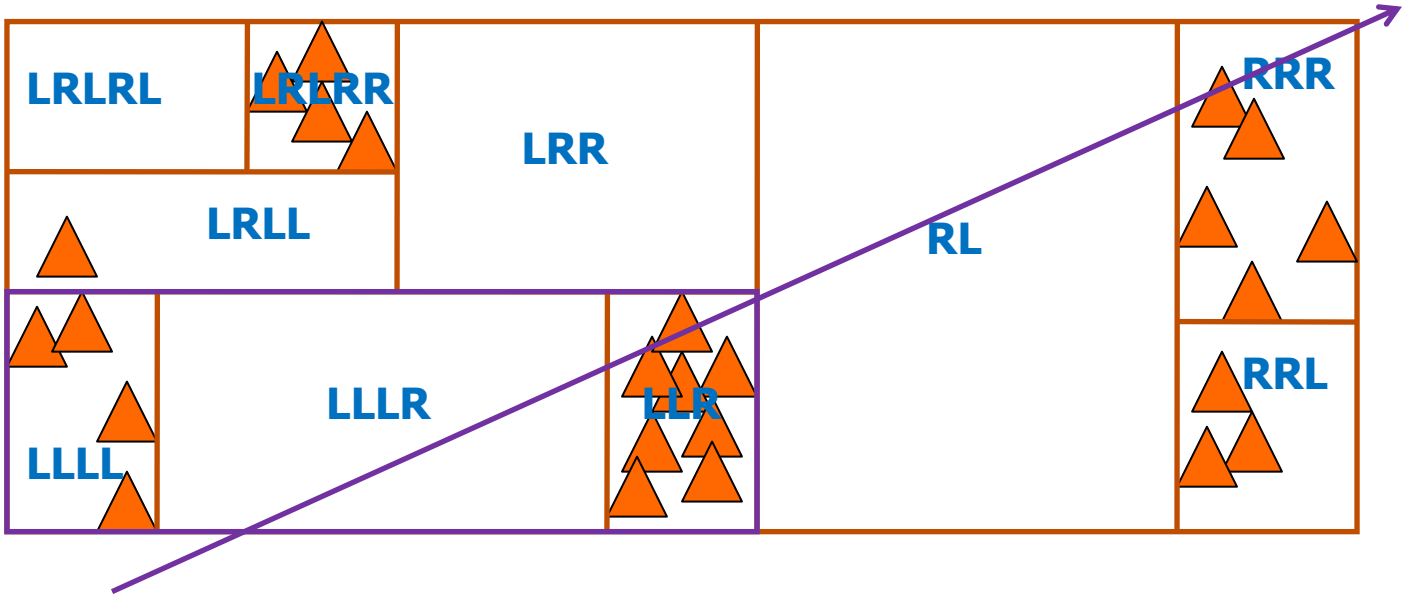


Стек: **R**

Текущий узел: **L**

kd-деревья

- Алгоритм траверса

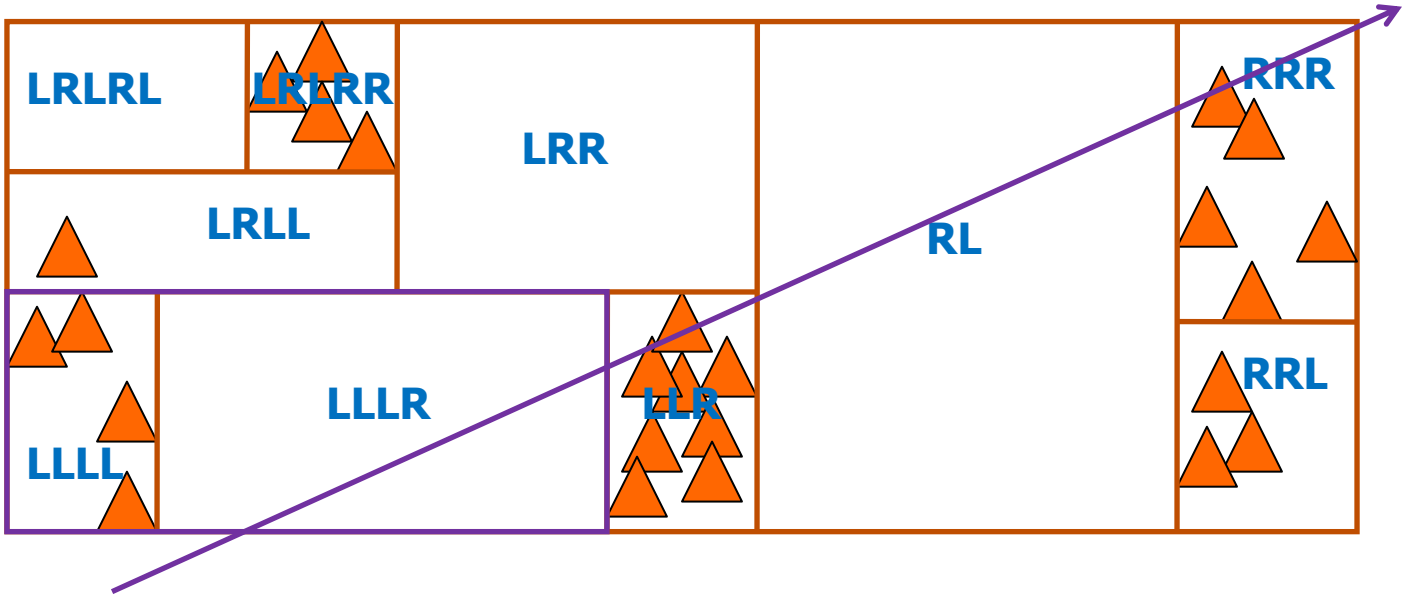


Стек: **R**

Текущий узел: **LL**

kd-деревья

- Алгоритм траверса

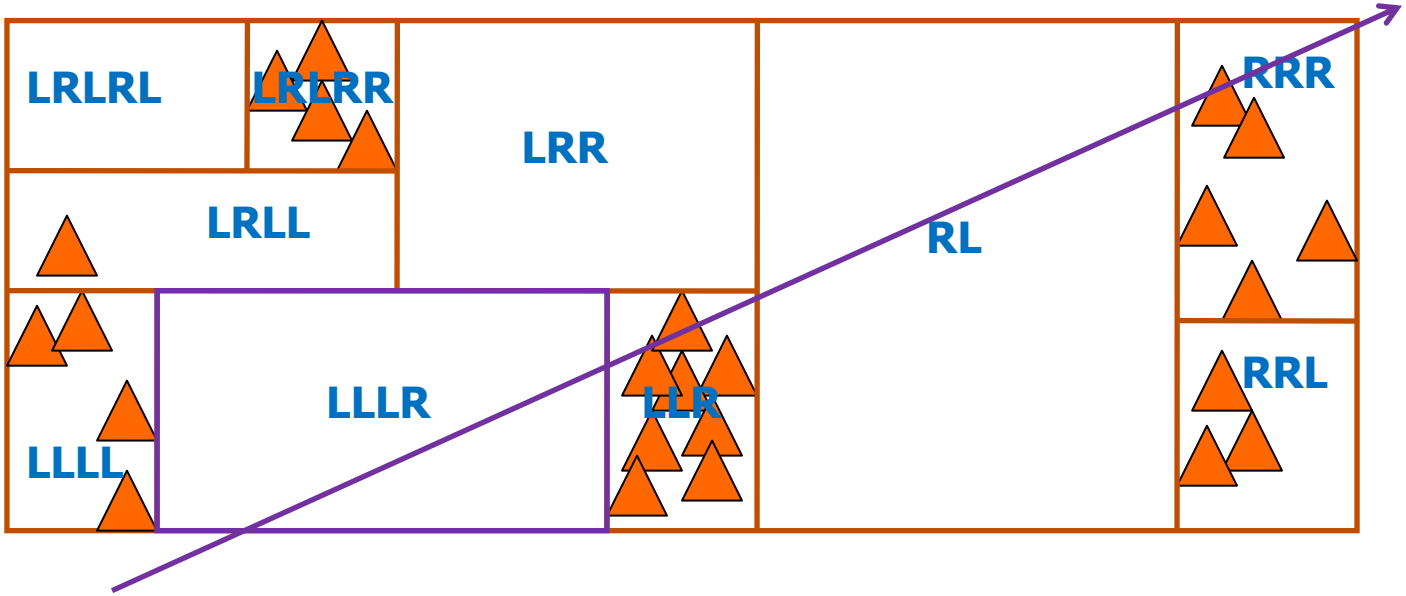


Стек: **LLR, R**

Текущий узел: **LLL**

kd-деревья

- Алгоритм траверса

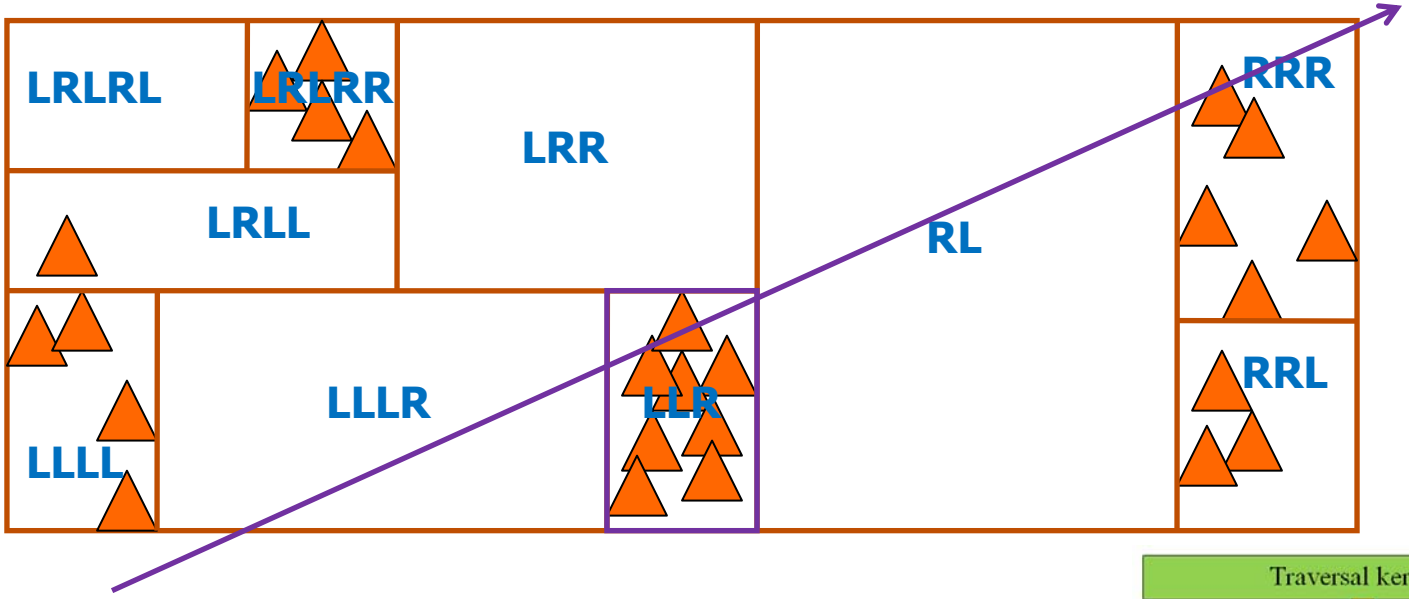


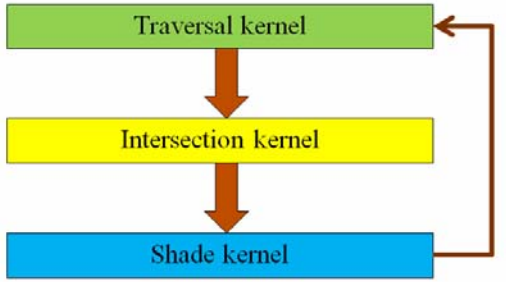
Стек: **LLR, R**

Текущий узел: **LLLR**

kd-деревья

- Алгоритм траверса





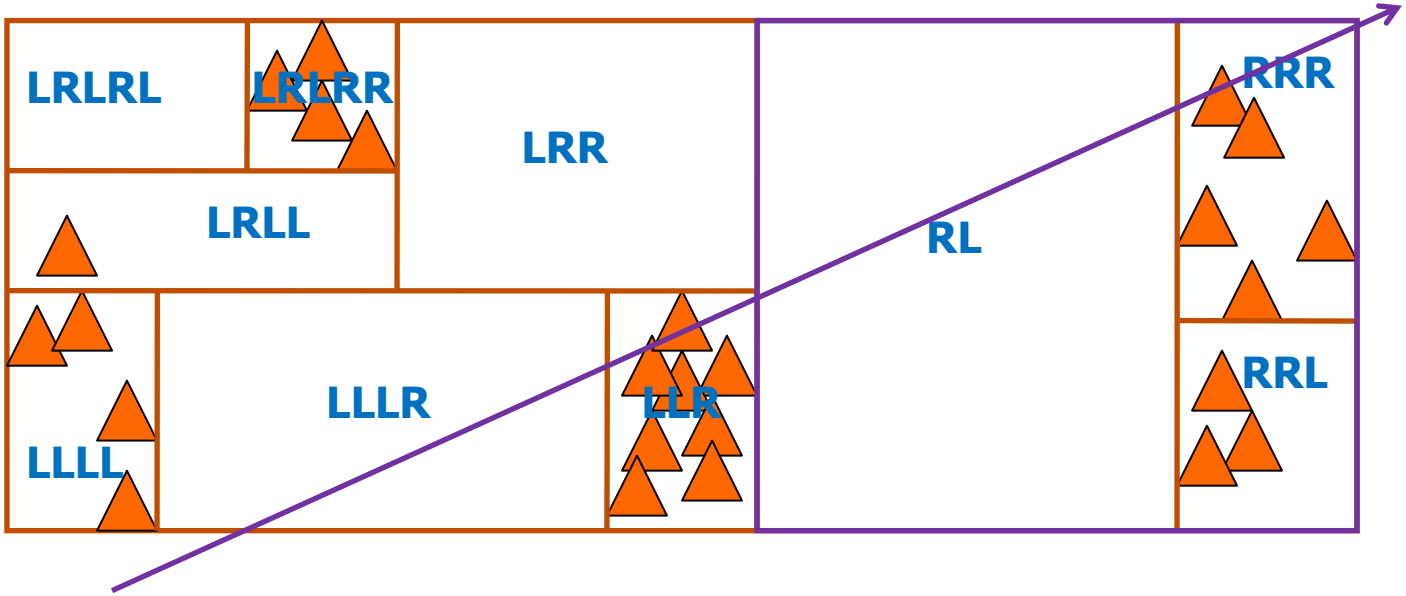
Стек: **R**

Текущий узел: **LLR**

Можно было бы остановиться!

kd-деревья

- Алгоритм траверса

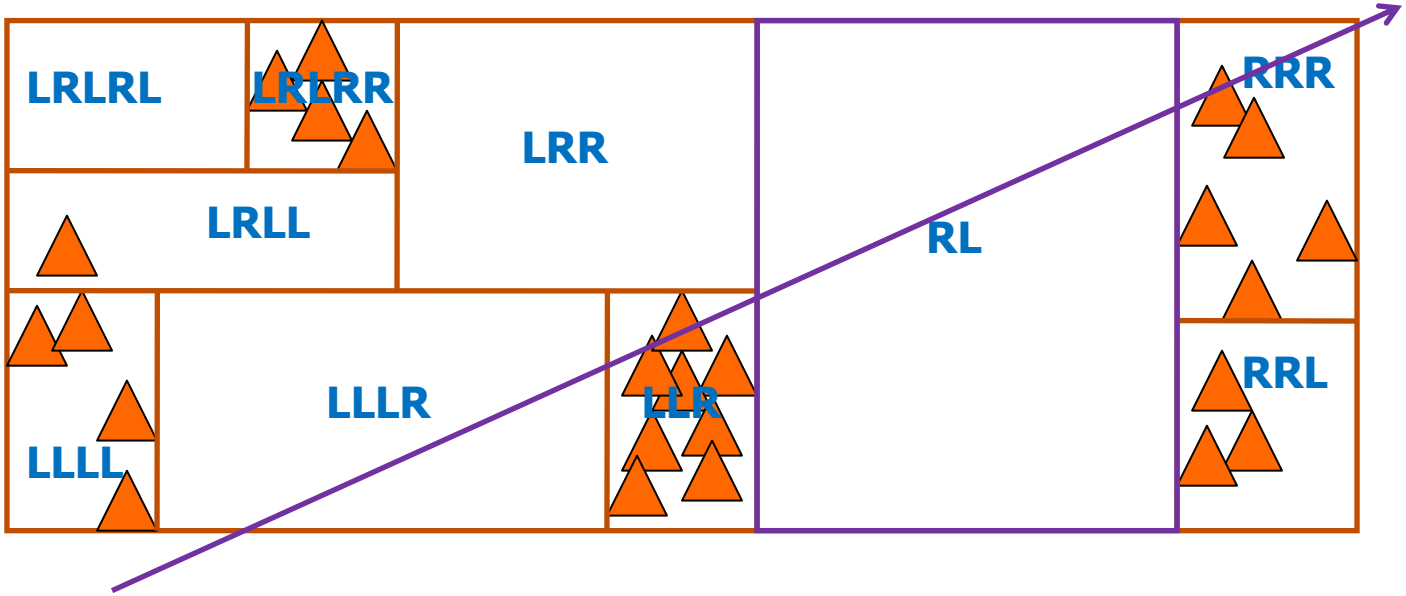


Стек:

Текущий узел: **R**

kd-деревья

- Алгоритм траверса

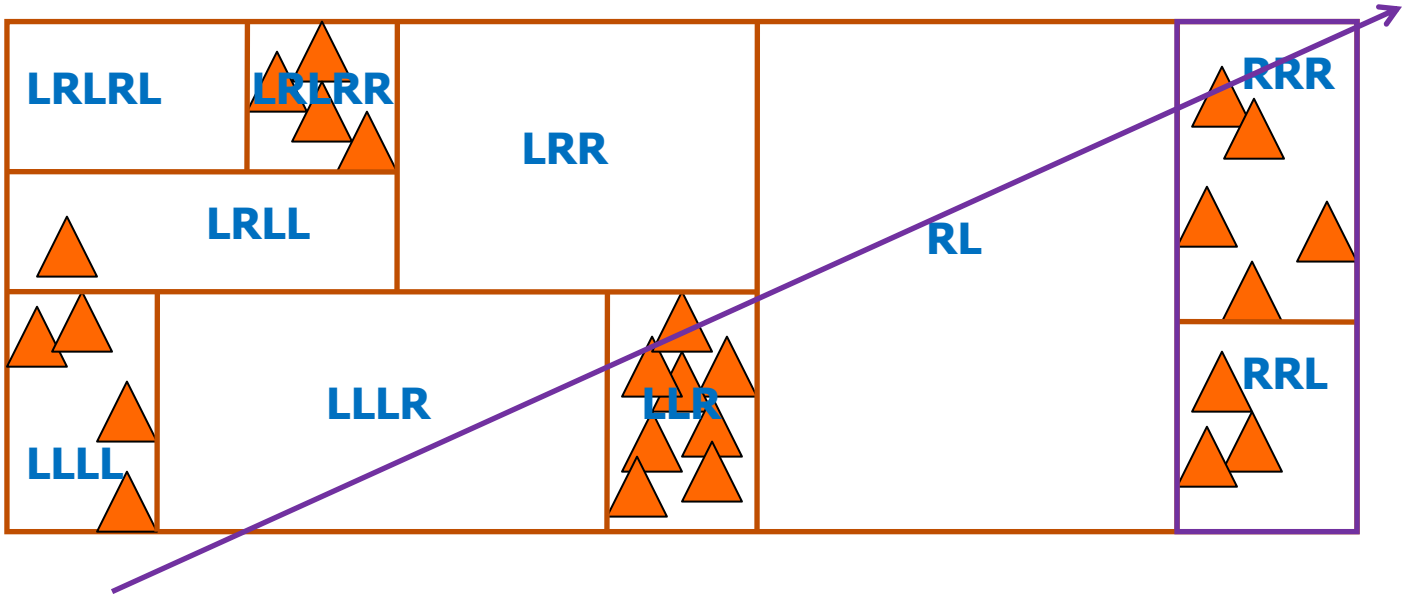


Стек: **RR**

Текущий узел: **RL**

kd-деревья

- Алгоритм траверса

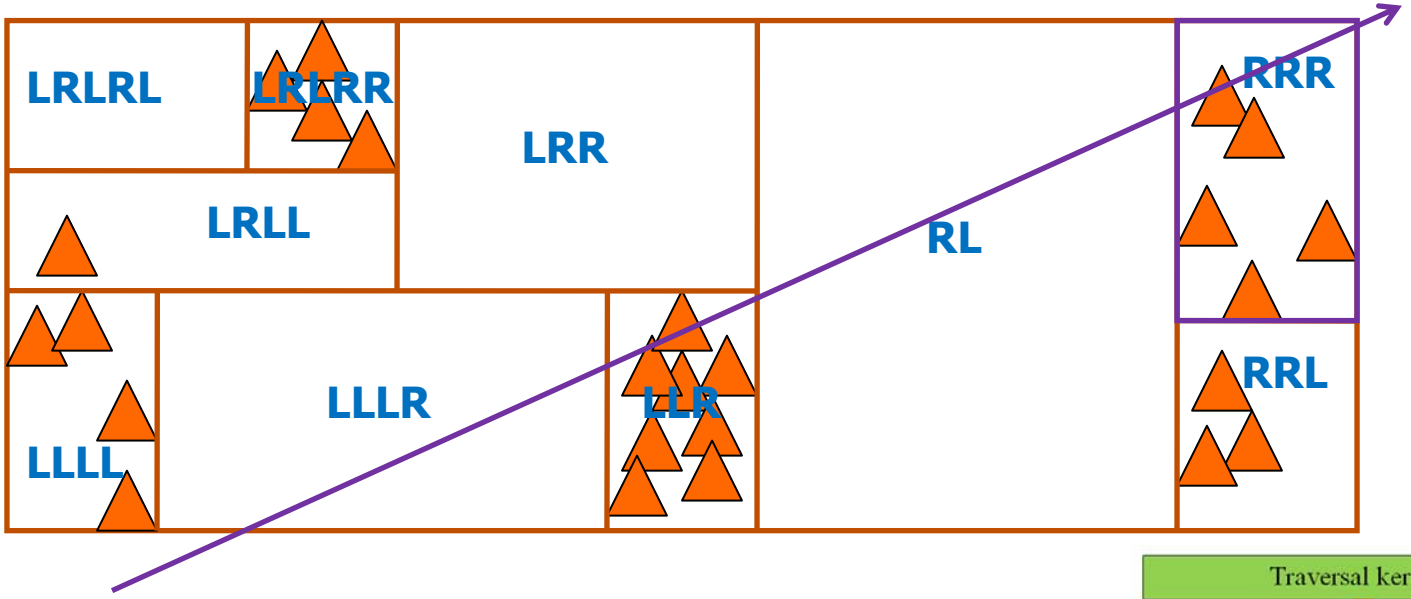


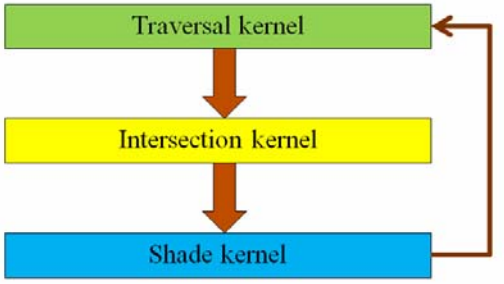
Стек:

Текущий узел: **RR**

kd-деревья

- Алгоритм траверса



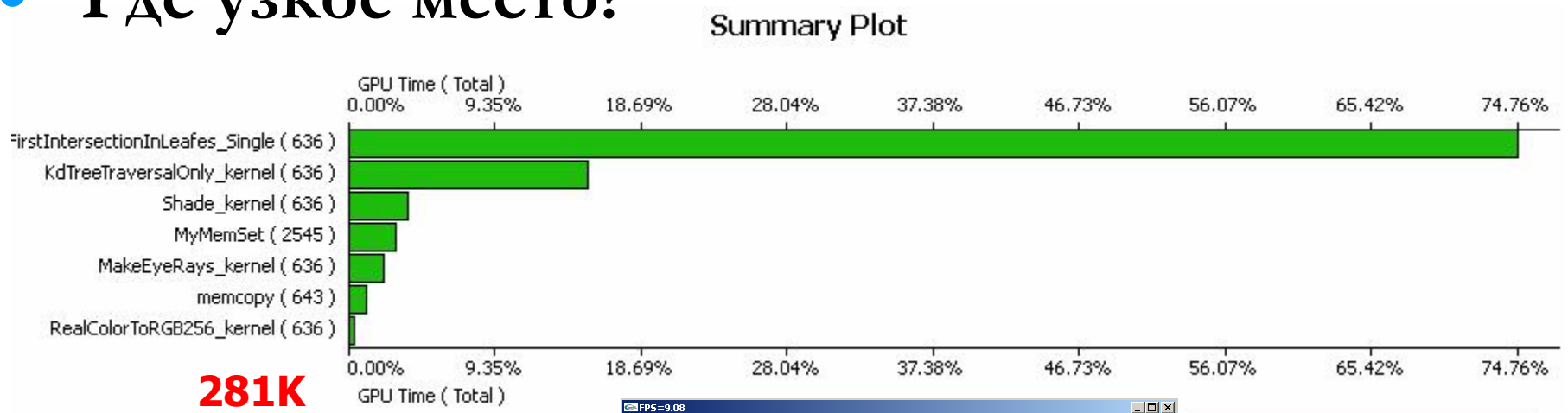


Стек:

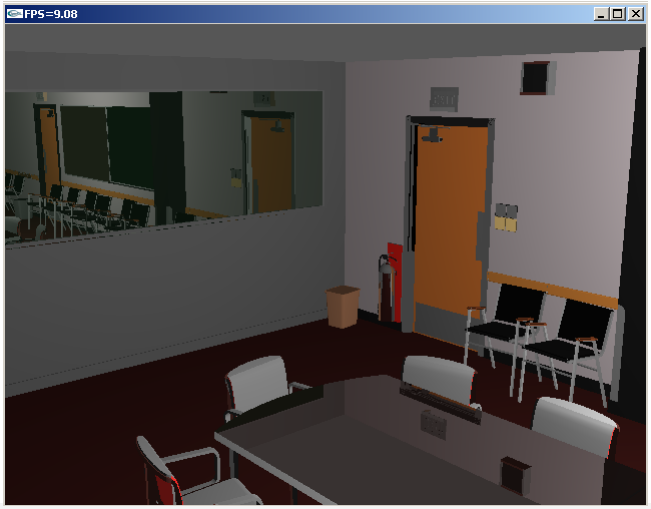
Текущий узел: **RRR** Конец, результат: **LLR, RRR**

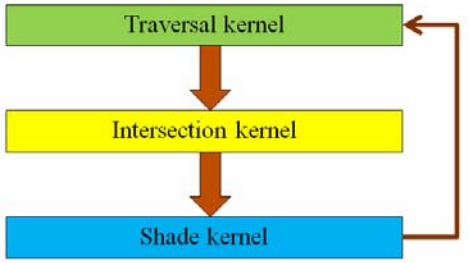
kd-деревья

- Где узкое место?



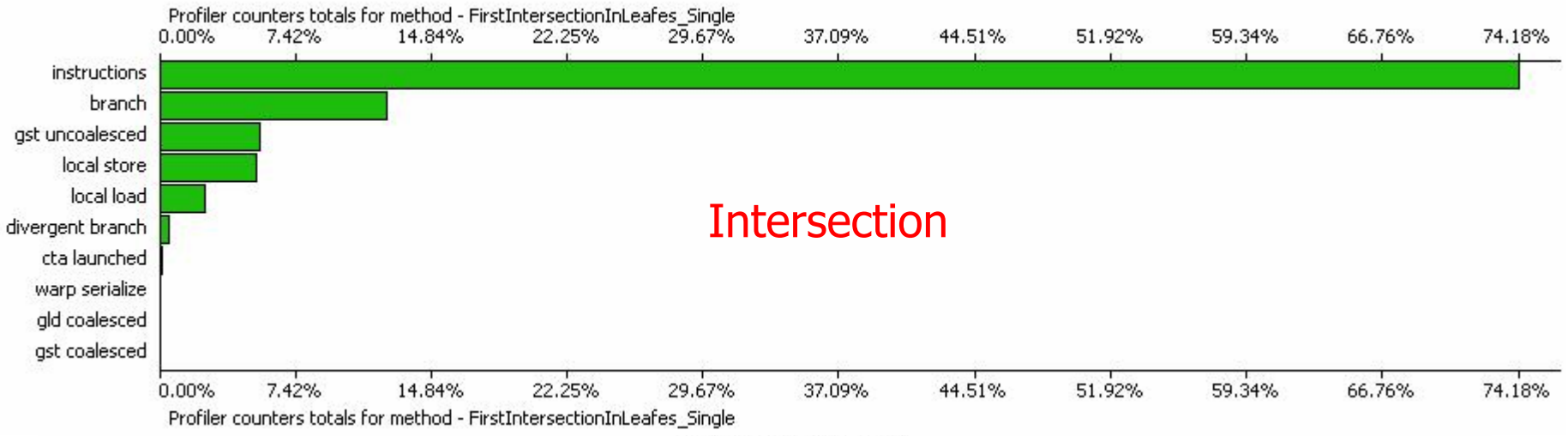
281K

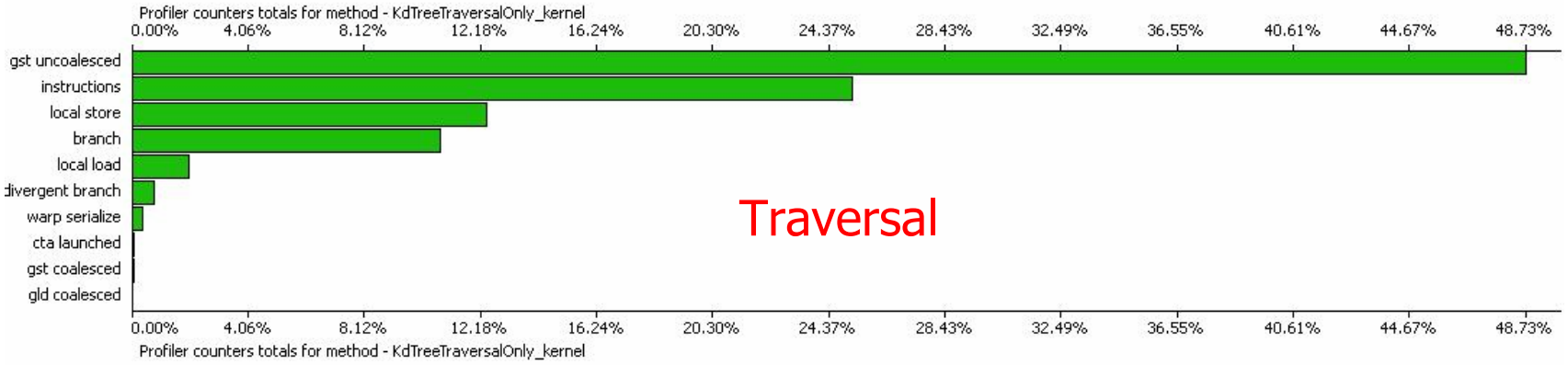




kd-деревья

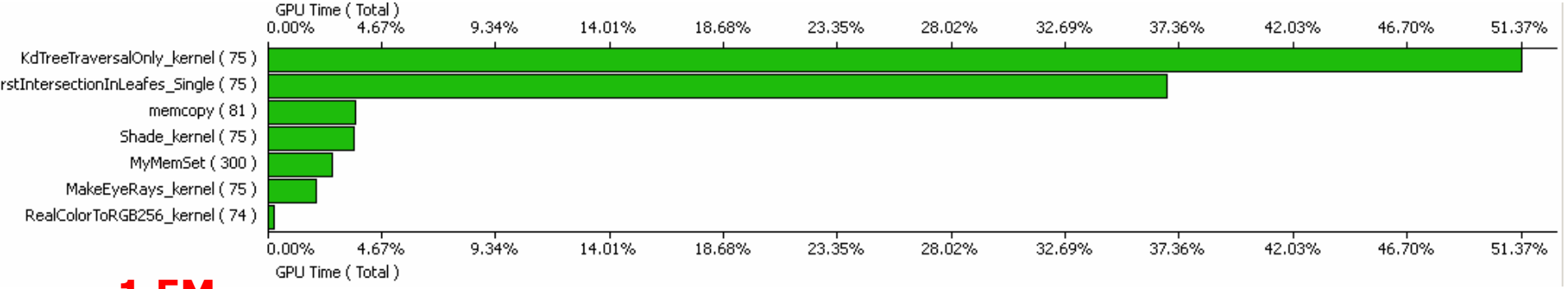
- Conference Room (281K)



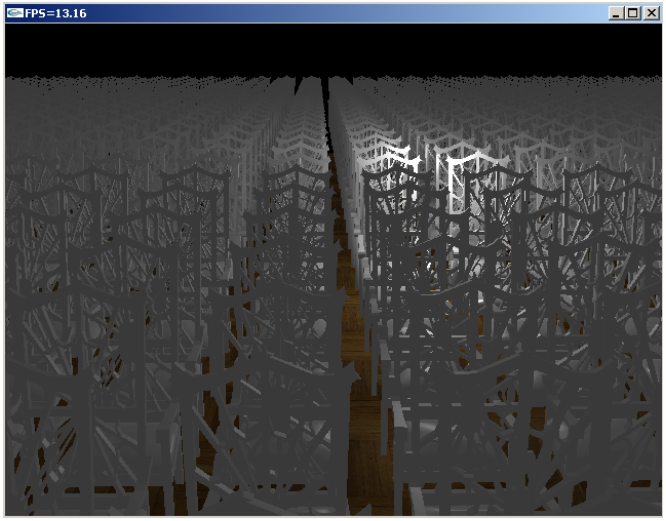


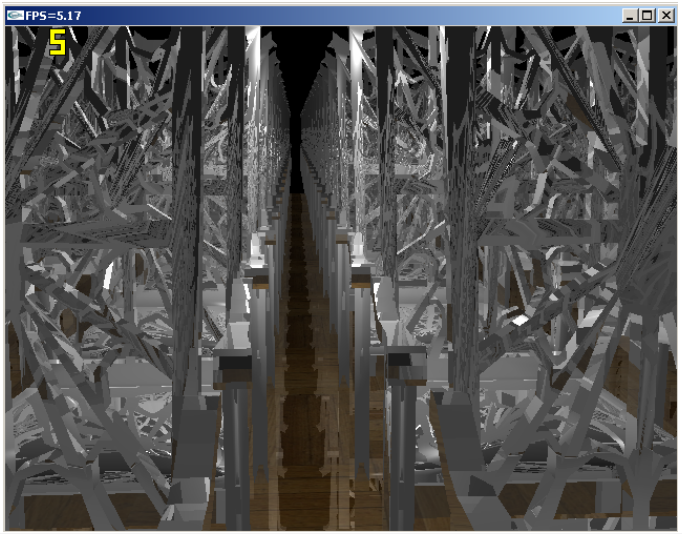
kd-деревья

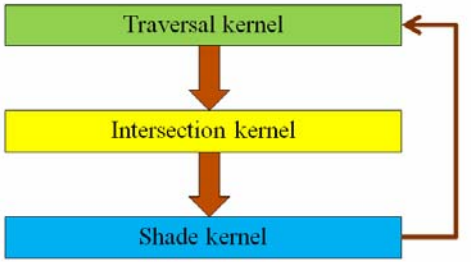
- Где узкое место?



1.5M

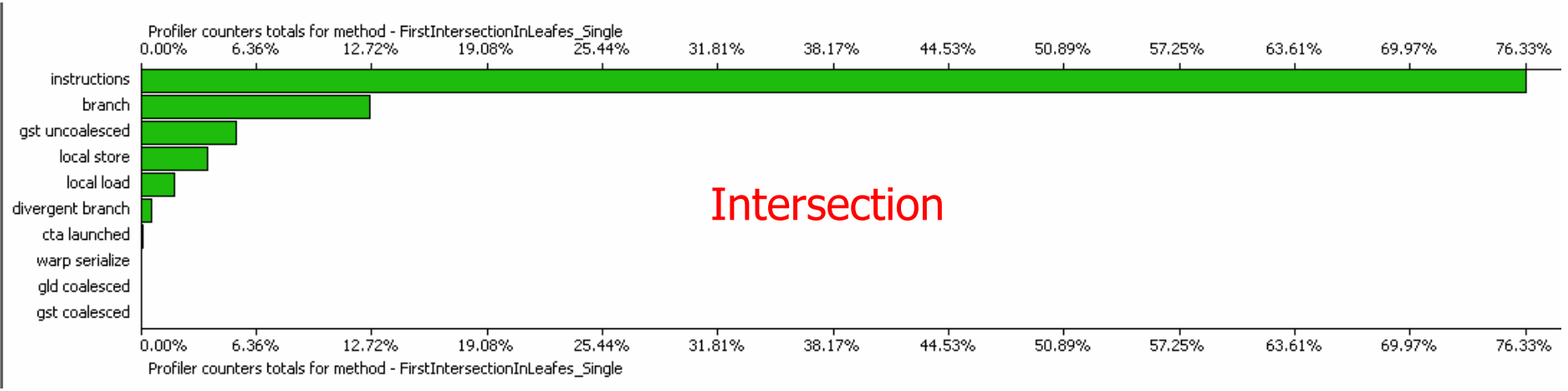


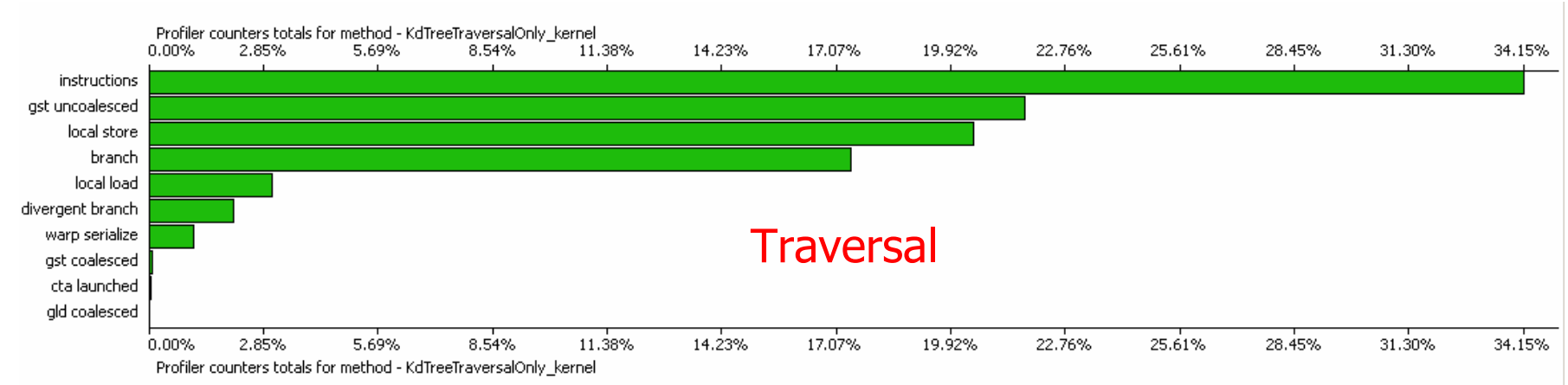




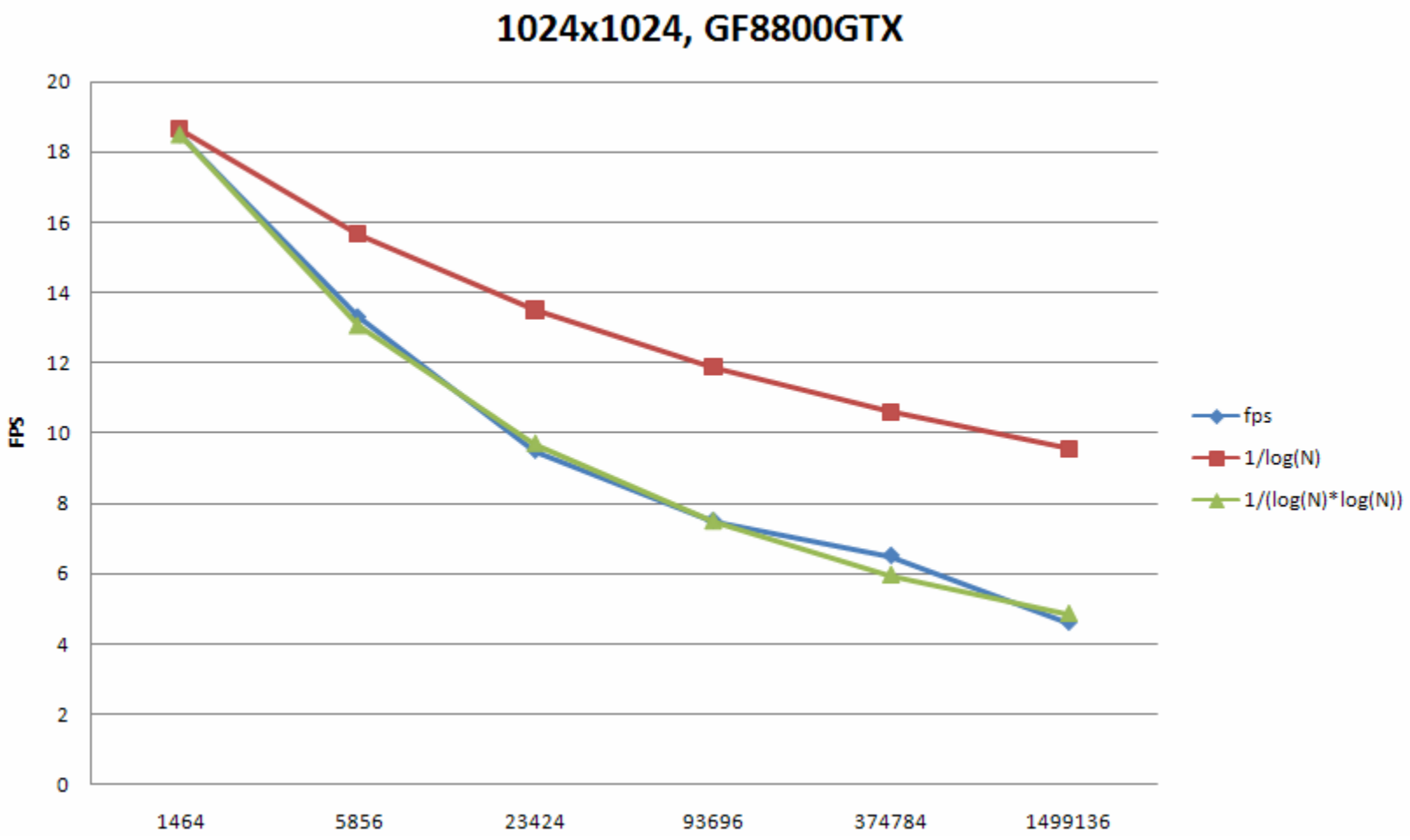
kd-деревья

- Стулья (1.5M)





Производительность



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