



# ТРАССИРОВКА ЛУЧЕЙ НА CUDA

Докладчик:

Фролов В.А. (ВМиК МГУ, G&M Lab; Nvidia)

Научный руководитель:

Игнатенко А.В. (ВМиК МГУ, G&M Lab)

Лекторы:

Боресков А.В. (ВМиК МГУ)

Харламов А.А. (ВМиК МГУ, Nvidia)



# План

- ⌘ OpenGL interoperability
- ⌘ RT – что, зачем и как?
- ⌘ Пересечение луча и треугольника
- ⌘ Ускоряющие структуры
- ⌘ Организация трассировки лучей
- ⌘ Ray marching
- ⌘ Задание



# OpenGL interop

## ⌘ OpenGL 2.0

☑ VBO, PBO

## ⌘ OpenGL 3.0

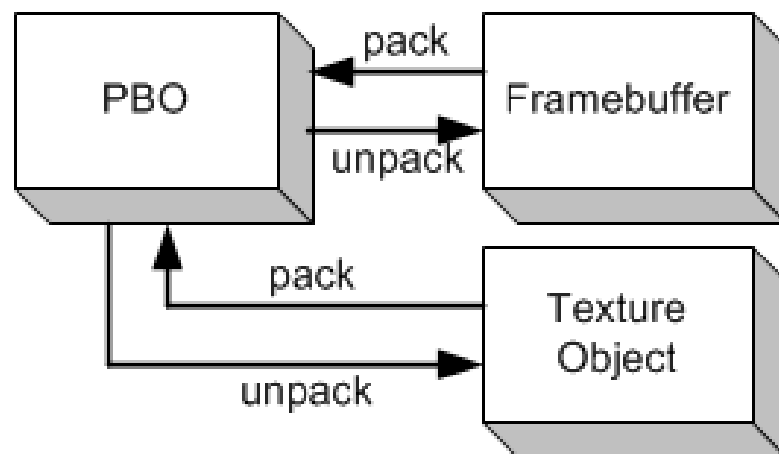
☑ VBO, PBO

☑ texture

☑ renderbuffer

## ⌘ OpenGL 4.0

☑ ?





# OpenGL 2.0 interop

## ⌘ Pack

⌘ `glReadPixels (...)`

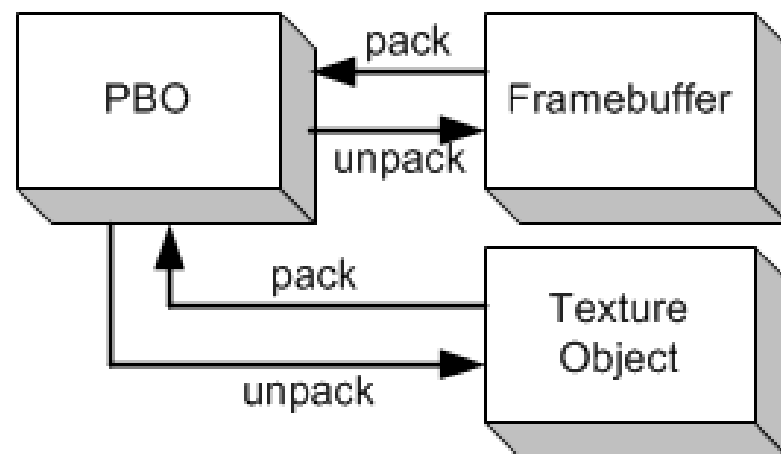
⌘ `glGetTexImage (...)`

## ⌘ Unpack

⌘ `glDrawPixels (...)`

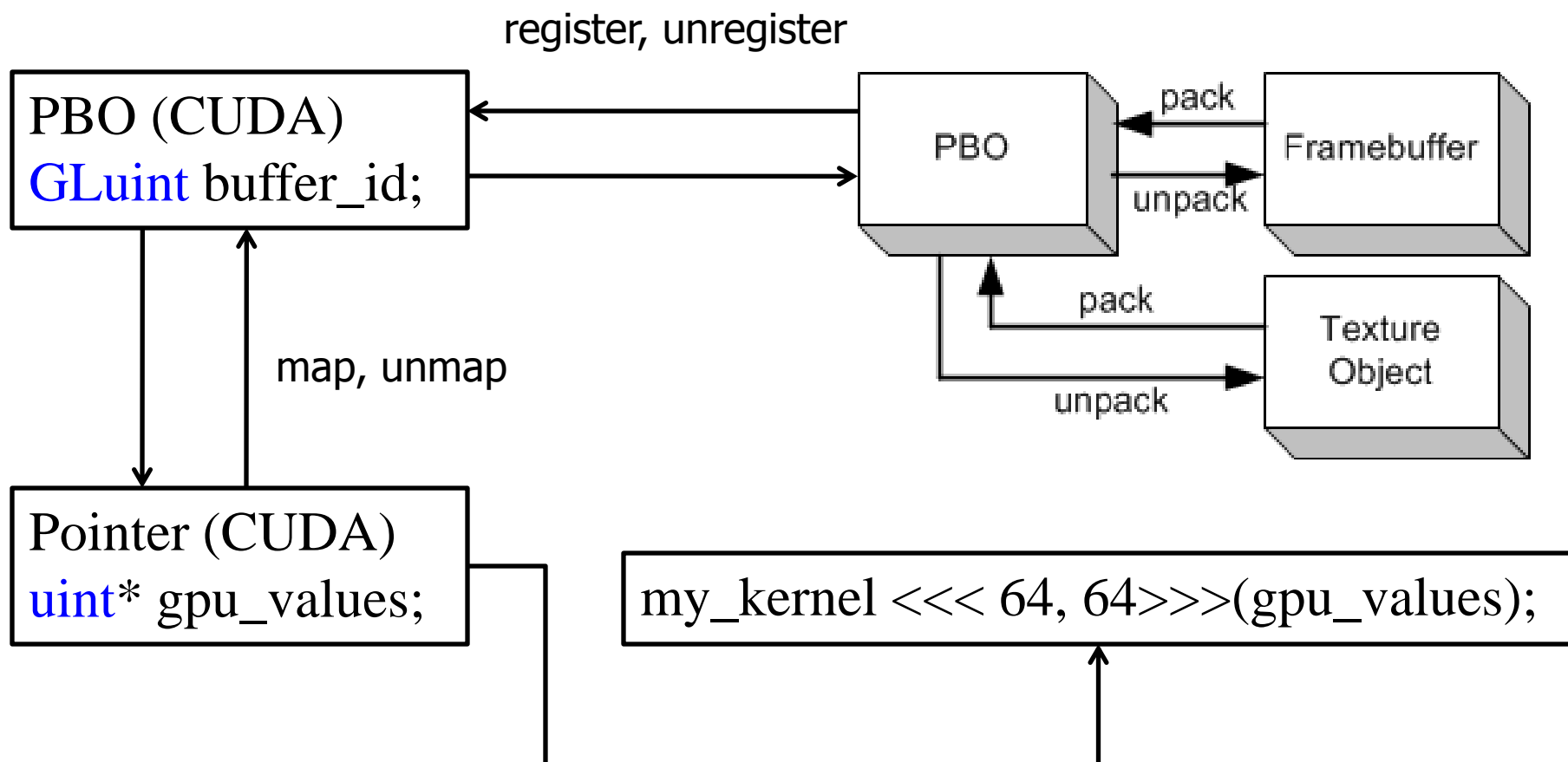
⌘ `glTexImage2D (...)`

⌘ `glTexSubImage2D (...)`





# OpenGL 2.0 interop





# OpenGL 2.0 interop

```
glGenBuffers(1, &pixelBuffer);  
glBindBuffer (GL_PIXEL_UNPACK_BUFFER, pixelBuffer);  
glBufferData(GL_PIXEL_UNPACK_BUFFER, w*h*sizeof(int),  
             screen_buffer, GL_STATIC_DRAW);  
  
cudaGLRegisterBufferObject (pixelBuffer);  
cudaGLMapBufferObject ((void*)&my_pointer, pixelBuffer);  
  
my_kernel<<<64, 64>>>(my_pointer);  
  
cudaGLUnmapBufferObject(pixelBuffer);  
cudaGLUnregisterBufferObject(pixelBuffer);  
glBindTexture(GL_TEXTURE_2D, tex_id);  
glTexImage2D (... , 0, ..., w, h, 0, ..., ..., (GLvoid*)0);
```

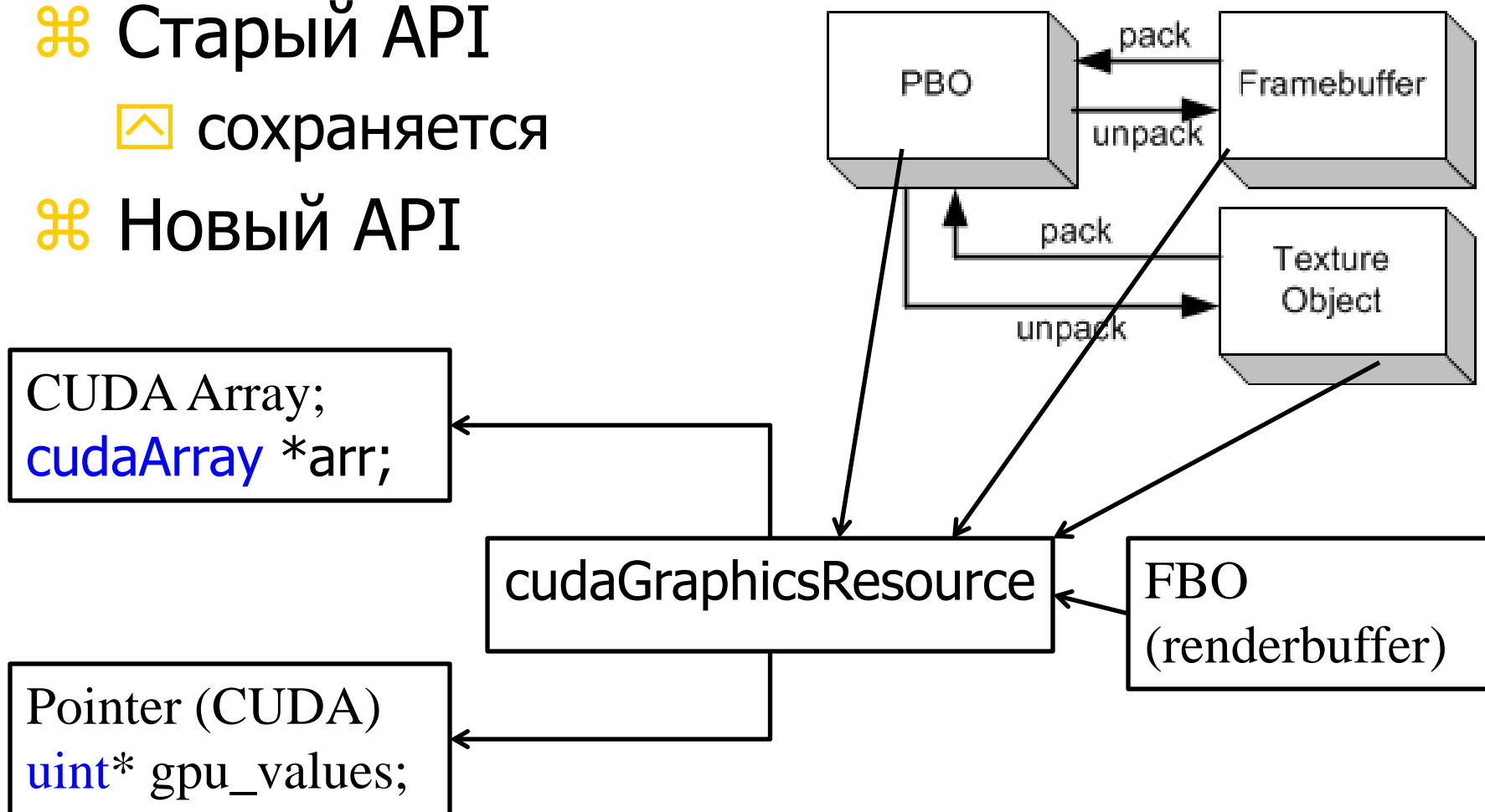


# OpenGL 3.0 interop

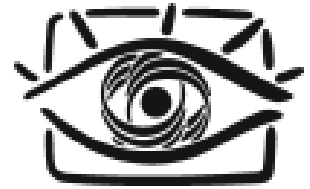
⌘ Старый API

☑ сохраняется

⌘ Новый API



# OpenGL 3.0 interop



```
GLuint tex_id;  
glGenTextures(1, & tex_id);  
...  
struct cudaGraphicsResource *tex_resource;  
cudaGraphicsGLRegisterImage (&tex_resource, tex_id,  
                             GL_TEXTURE_2D, cudaGraphicsMapFlagsReadOnly);  
cudaGraphicsMapResources (1, & tex_resource, 0);  
  
cudaArray *array_ptr;  
cudaGraphicsSubResourceGetMappedArray (&array_ptr, tex_resource, 0, 0);  
  
cudaMemcpyToArray(array_ptr, 0, 0, dest, size, cudaMemcpyDeviceToDevice);  
  
cudaGraphicsUnmapResources(1, & tex_resource, 0);
```





# 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

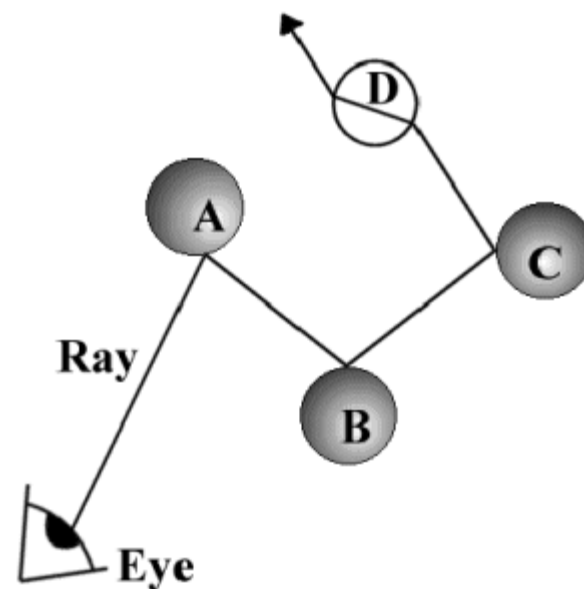
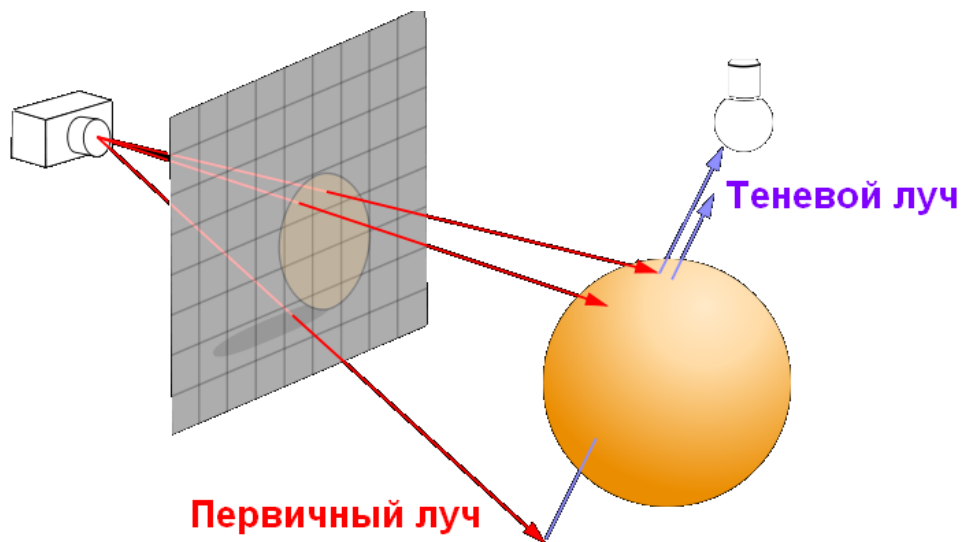
- ☑ Обратная трассировка лучей
- ☑ Растеризация + обратная трассировка лучей

# Обратная трассировка лучей



## ⌘ Алгоритм

☑ Первичные, теневые, отраженные лучи



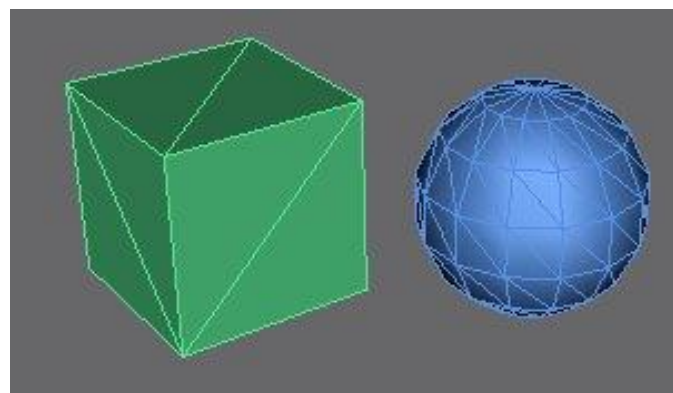
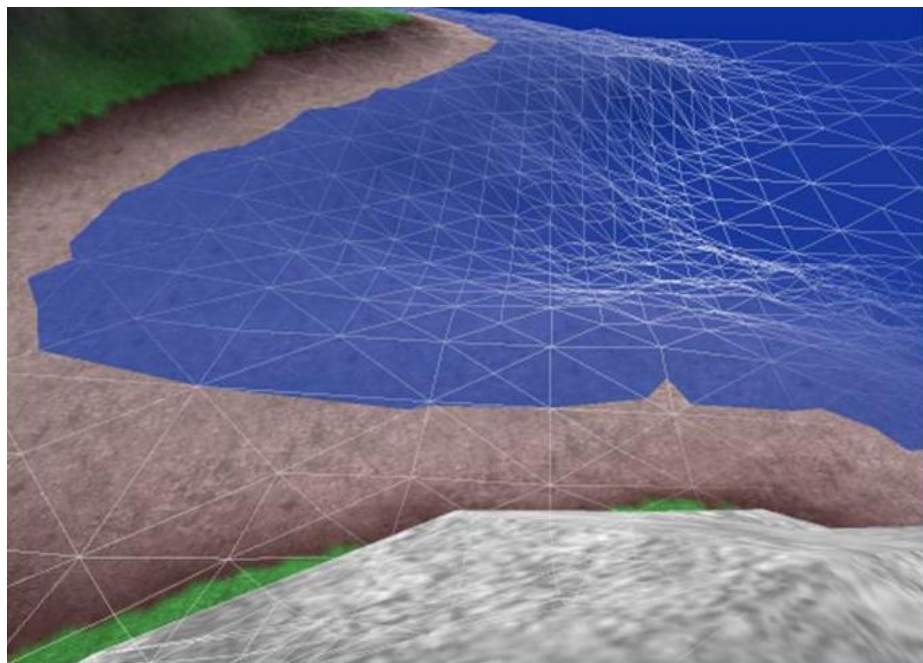
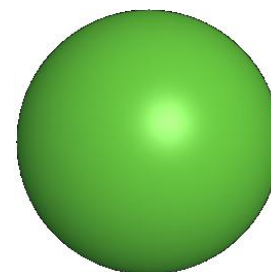


# Ray Tracing

⌘ Представление 3D объектов

☑ Аналитическое

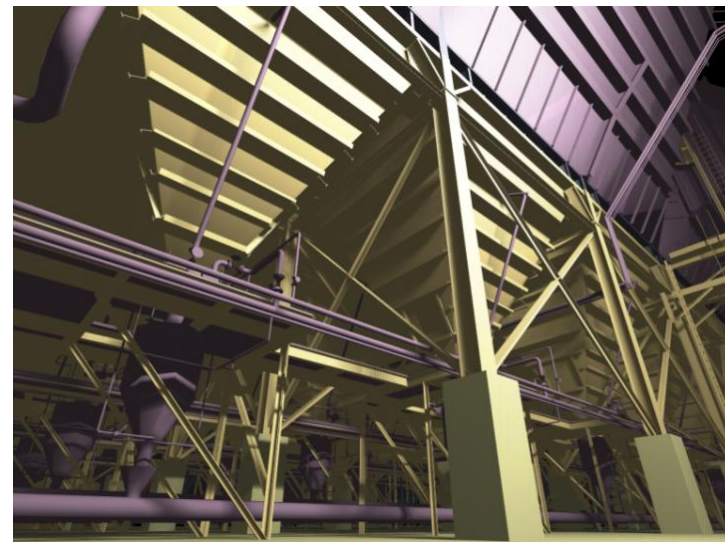
☑ Меши из треугольников





# 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

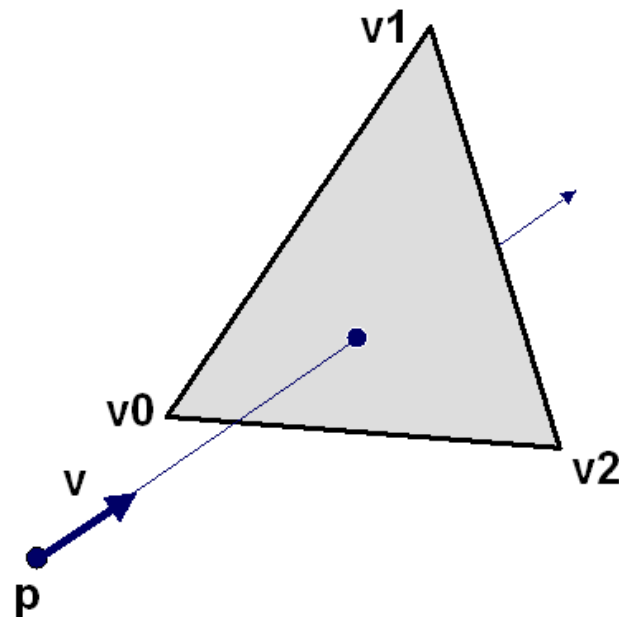
☒  $Ax + By + Cz + D = 0$

☒ 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

☒  $z = p + v * t$

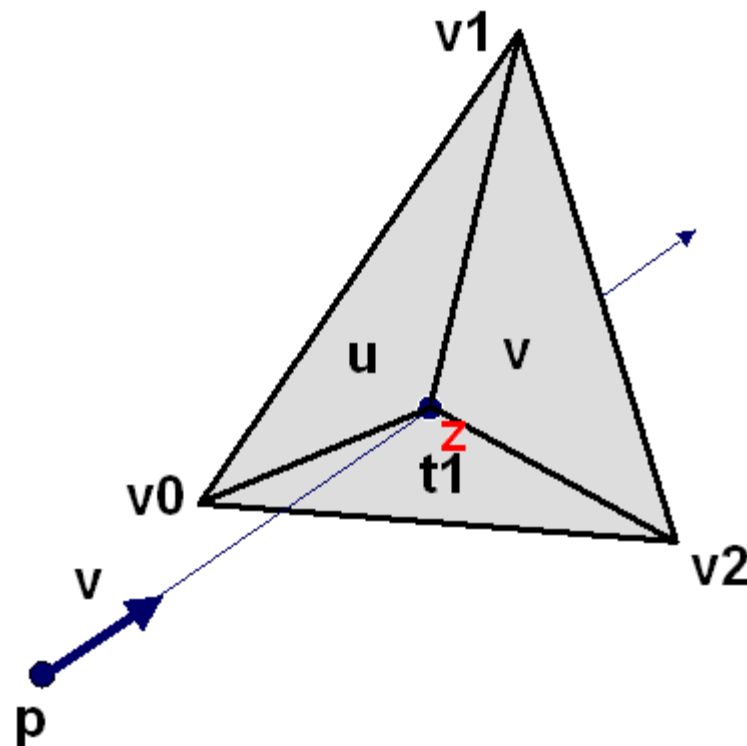
☒  $S = \text{cross}(v1-v0, v2-v0)$

☒  $u = \text{cross}(v1-z, v0-z)$

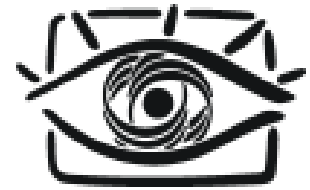
☒  $v = \text{cross}(v1-z, v2-z)$

☒  $t1 = \text{cross}(v2-z, v0-z)$

☐  $|u + v + t1 - S| < \varepsilon$



# Пересечение луча и треугольника



## ⌘ Оптимизированный вариант

☑ Барицентрические координаты

☒  $u := u/S, v := v/S, t1 := t1/S$

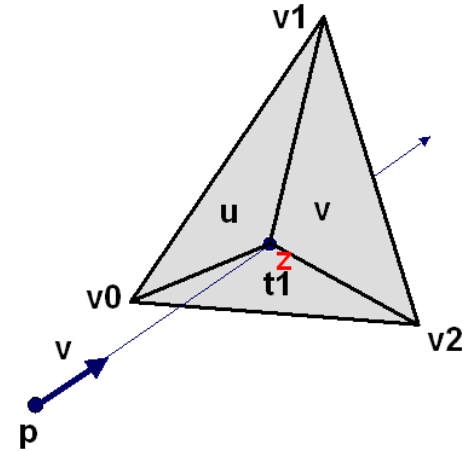
☒  $t1 = 1 - u - v$

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

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

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

☑ 3 уравнения, 3 неизвестных



# Пересечение луча и треугольника



## ⌘ Оптимизированный вариант

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

$$E1 = v1 - v0$$

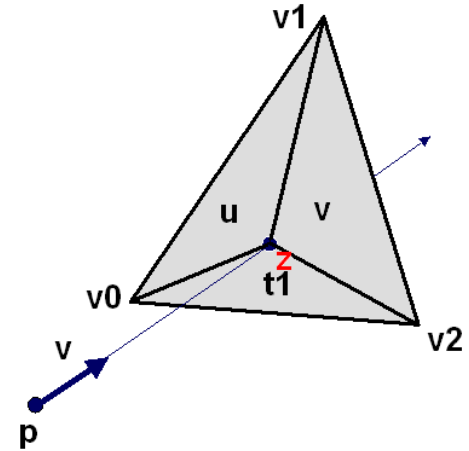
$$E2 = v2 - v0$$

$$T = p - v0$$

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

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

$$D = v$$



# Пересечение луча и треугольника



## ⌘ Простой вариант

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

☒ 248-404 тактов

## ⌘ Оптимизированный вариант

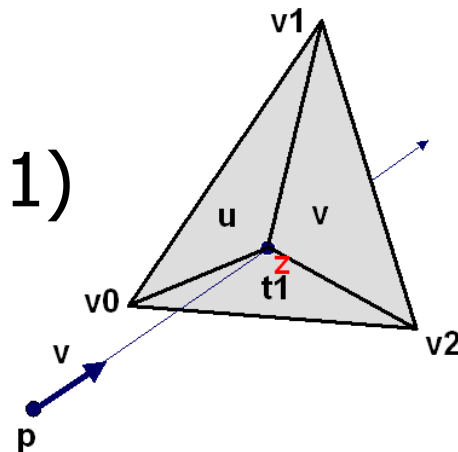
☑ Операции ( $*$  : 23,  $+/-$  : 24,  $/$  : 1)

☒ 132-224 такта

## ⌘ Как считали нижнюю оценку?

☑ использование mad вместо mul и add

☑  $4 * (N\_mul + |N\_add - N\_mul|)$





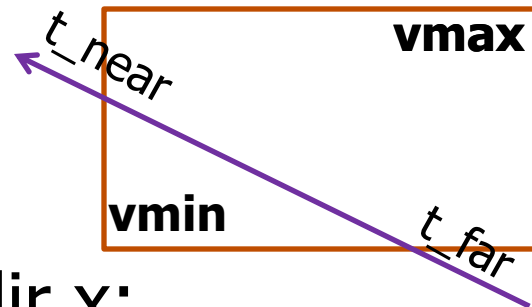
# Другие примитивы

⌘ 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

# Осцирансы



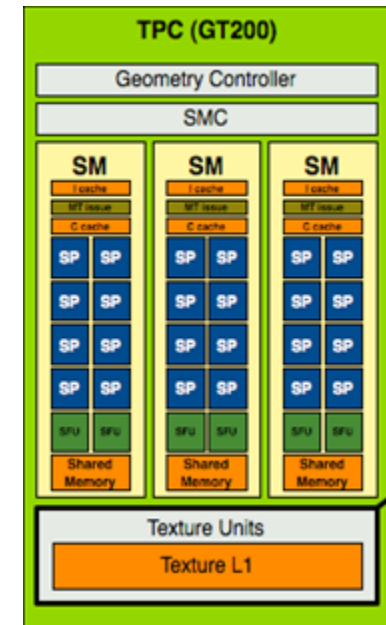
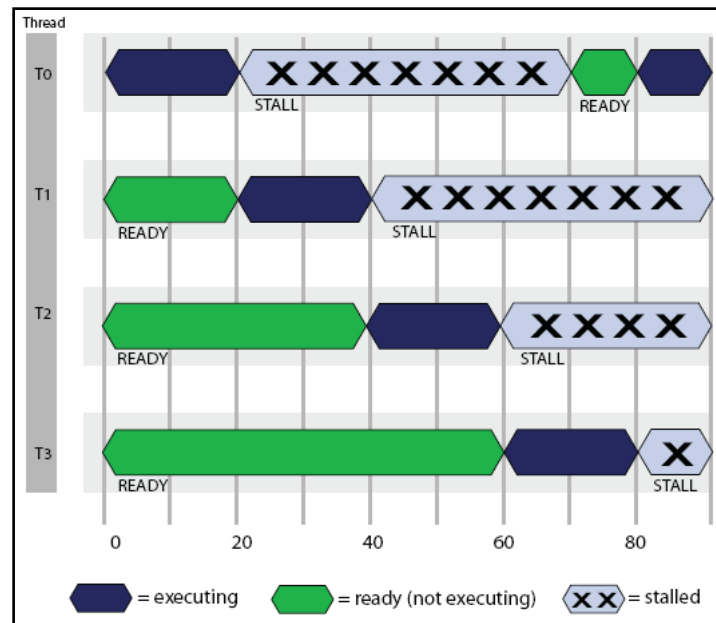
⌘ Очень быстрое переключение

⌘ Влияет:

⌘ Регистры

⌘ shared память

⌘ Размер блока



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



# Осцирансу

## ⌘ 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?

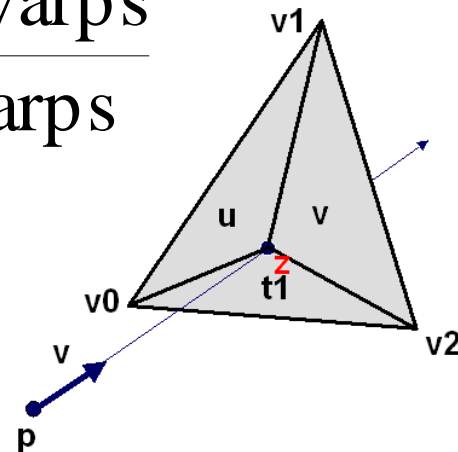
☒ рег ≤ 40: 3 блока, 6 warp-ов активны

☒ 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



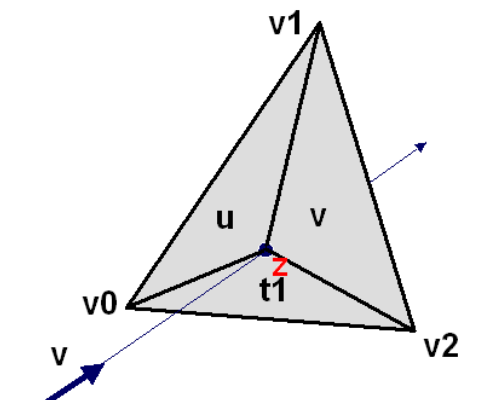
# Пересечение луча и треугольника



## ⌘ 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



## ⌘ Unit test

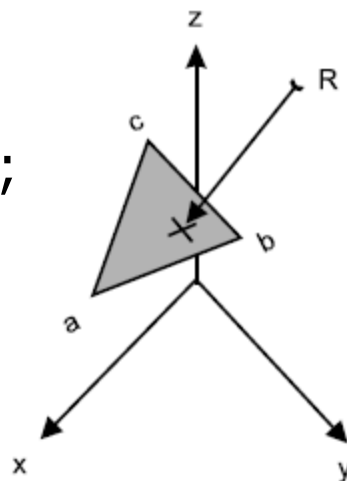
```
float3 o = mul3x4(m, origin);
```

```
float3 d = mul3x3(m, dir);
```

```
float t = -o.z/d.z;
```

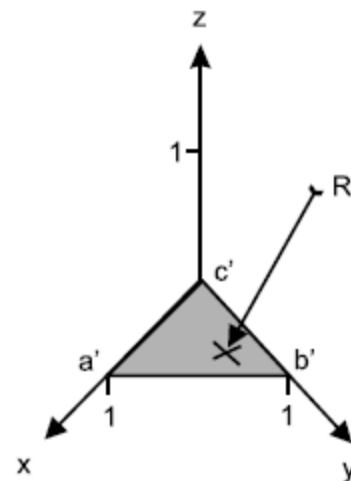
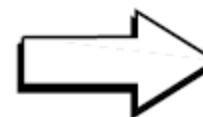
```
float u = o.x + t*d.x;
```

```
float v = o.y + t*d.y;
```



(a) world coordinate space

affine triangle  
transformation



(b) unit triangle space

$$T_{\Delta}^{-1} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = A \quad T_{\Delta}^{-1} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = B$$
$$T_{\Delta}^{-1} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = C \quad T_{\Delta}^{-1} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = N$$

# Пересечение луча и треугольника



## ⌘ Unit test

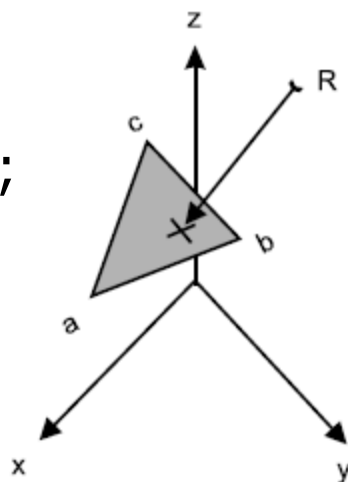
```
float3 o = mul3x4(m, origin);
```

```
float3 d = mul3x3(m, dir);
```

```
float t = -o.z/d.z;
```

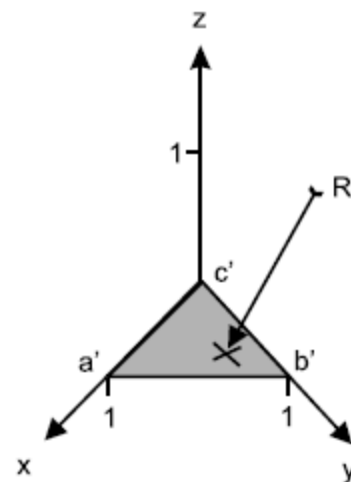
```
float u = o.x + t*d.x;
```

```
float v = o.y + t*d.y;
```



(a) world coordinate space

affine triangle  
transformation



(b) unit triangle space

Где m это  $T_{\Delta}$

$$T_{\Delta}^{-1}(X) = \begin{pmatrix} A_x - C_x & B_x - C_x & N_x - C_x \\ A_y - C_y & B_y - C_y & N_y - C_y \\ A_z - C_z & B_z - C_z & N_z - C_z \end{pmatrix} \cdot X + \begin{pmatrix} C_x \\ C_y \\ C_z \end{pmatrix}$$

# Пересечение луча и треугольника



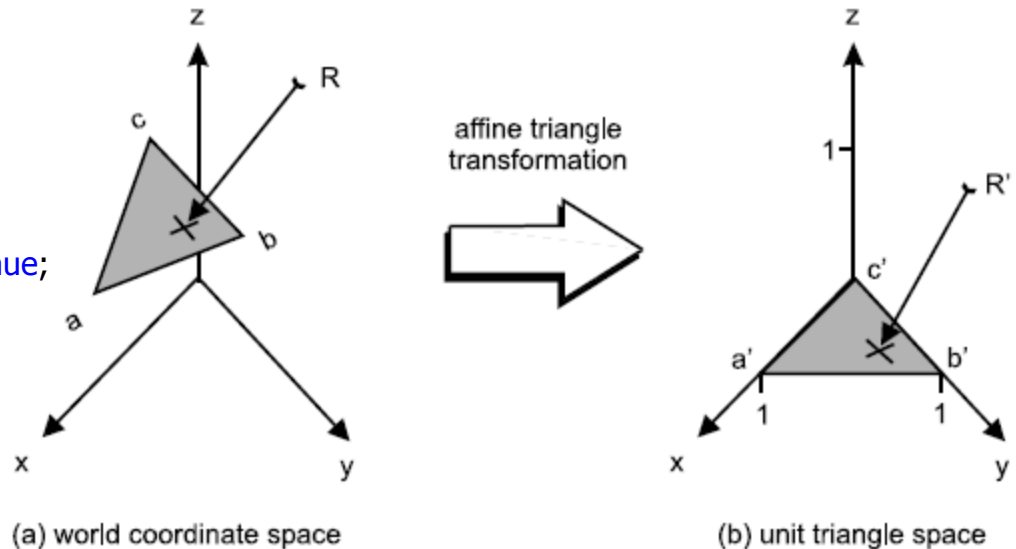
## ⌘ Unit test

```
float4 row2 = tex1Dfetch(tex, triAddress+0);  
(oz,dz) = ...  
float t = -oz/dz;  
if (t <= EPSILON_E8M || t >= tTriangle) continue;
```

```
float4 row0 = tex1Dfetch(tex, triAddress+1);  
(ox, dx) = ...  
float u = ox + t*dx;  
if (u < 0.f) continue;
```

```
float4 row1 = tex1Dfetch(tex, triAddress+2);  
(oy dy) = ...  
float v = oy + t*dy;
```

```
if(v >= 0 && (u+v) <= 1.f) {  
    tTriangle = t;  
    tIndex = currIndex;  
}
```



Операции (\*: 20, + : 20, / : 1)

- ➡ 116-196 тактов
- ➡ 21 регистр на весь kernel
- ➡ 22 + поиск в kd дереве
- ➡ 25 + persistent threads



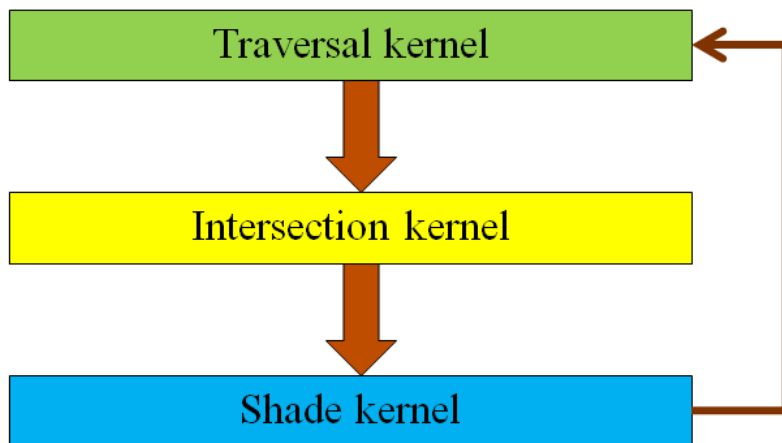
# Экономия регистров

## ⌘ Два подхода к экономии регистров

☐ uber kernel



☐ separate kernel



```
__global__ void Uber_kernel()  
{
```

```
START:
```

```
    if (state == TRAVERSE)
```

```
    {
```

```
        // .....
```

```
        if (node.leaf())
```

```
        {
```

```
            // save necessary data
```

```
            state = INTERSECTION;
```

```
            goto START;
```

```
        }
```

```
    }
```

```
    else if (state == INTERSECTION)
```

```
    {
```

```
        // ...
```

```
    }
```

```
    else if (state == SHADE)
```

```
    {
```

```
        //...
```

```
    }
```

```
}
```



# Экономия регистров

## Uber kernel

- ⌘ Рекурсия (+)
- ⌘ Локальность данных (+)
- ⌘ Непрерывное выполнение (+)
- ⌘ Трудно профилировать (-)
- ⌘ Трудно оптимизировать (-)
- ⌘ Самая сложная часть лимитирует все ядро (-)
- ⌘ Нельзя пересортировывать данные (-)

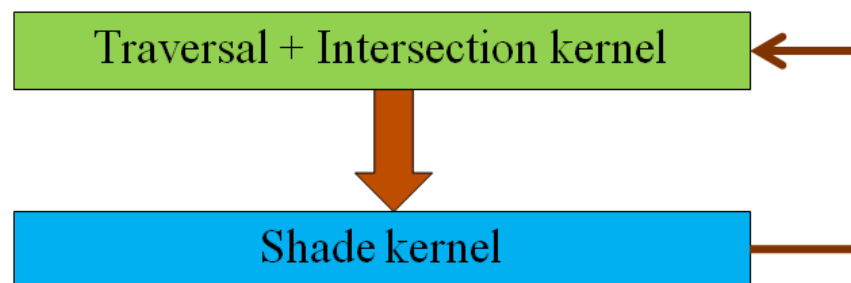
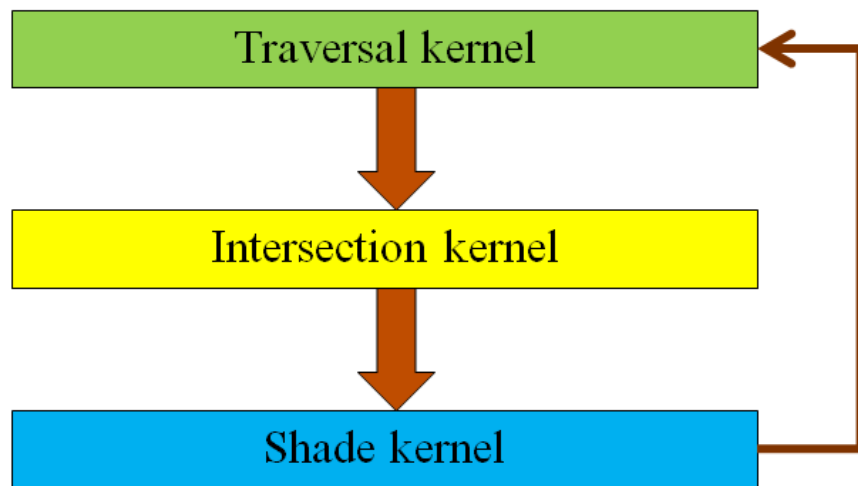
## Separate kernel

- Нет рекурсии (-)
- Обращения к DRAM (-)
- Остановка ядер (-)
- Легко профилировать и оптимизировать (+)
- Гибкость (+)
- Проще интегрировать с CPU реализациями (+)



# Архитектура

- ⌘ Ядро пересечений – 24-32 регистра
- ⌘ Но это только пересечения
- ⌘ Нужно разбить алгоритм трассировки на несколько ядер





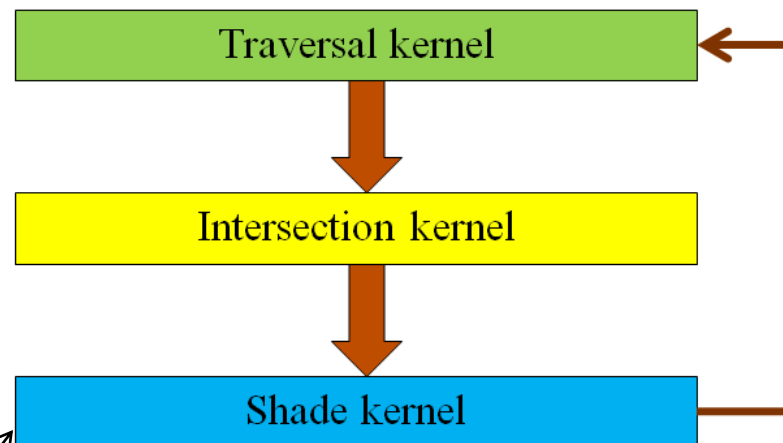
# Архитектура

⌘ Как хранить геометрию?

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

```
struct Triangle {  
    float4 v[3];  
    // store index in v[2].w  
};
```

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



Vertex array:



Index array:





The diagram illustrates the data flow for triangle intersection using a kd tree. It consists of four main components arranged vertically:

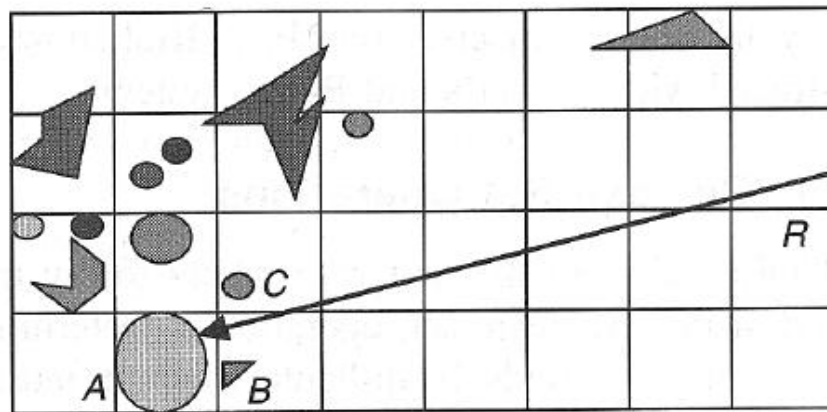
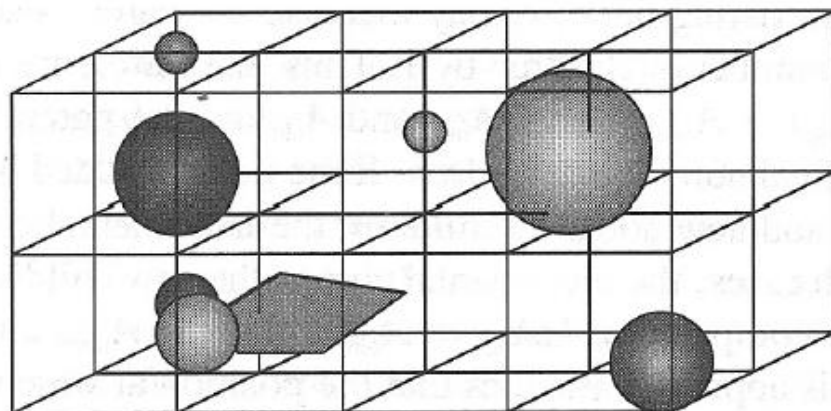
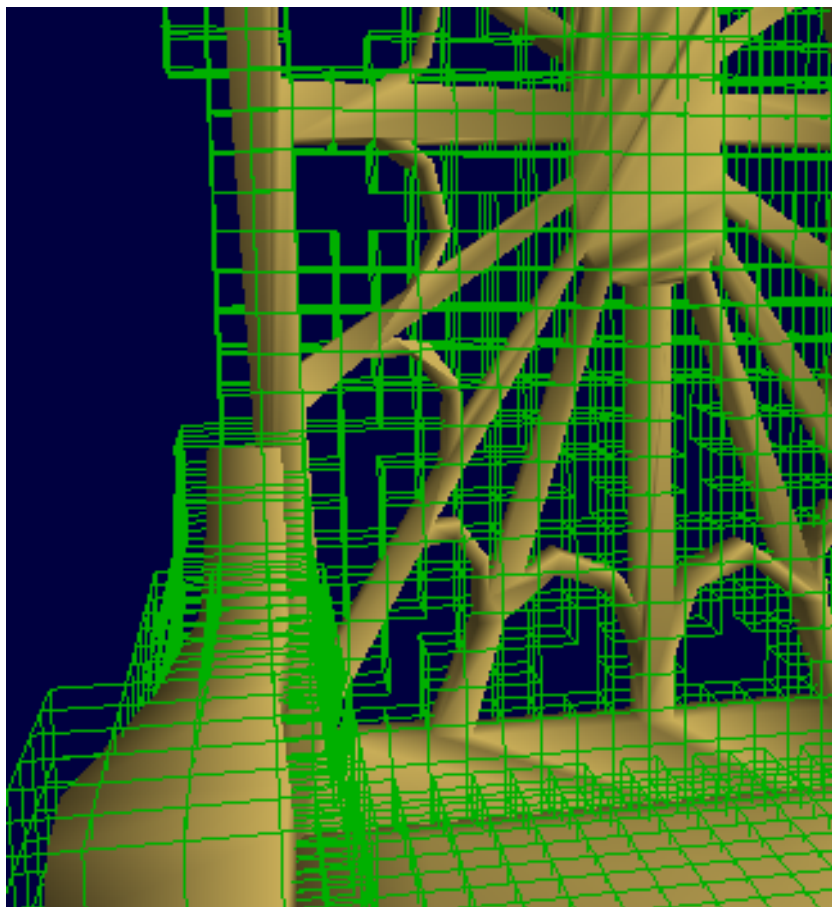
- kd tree** (Yellow box): The top component, which provides input to the Triangle list.
- Triangle list** (Green box): Receives input from the kd tree and performs an **intersection** operation (indicated by a box with an arrow pointing to it from the right).
- indices** (Orange box): Receives input from the Triangle list.
- vertices** (Blue box): The bottom component, which receives input from the indices.

Arrows indicate the flow of data: from the kd tree to the Triangle list, from the Triangle list to the indices, and from the indices to the vertices. A box labeled "intersection" points to the Triangle list, indicating the operation performed on the data received from the kd tree.





# Регулярная сетка

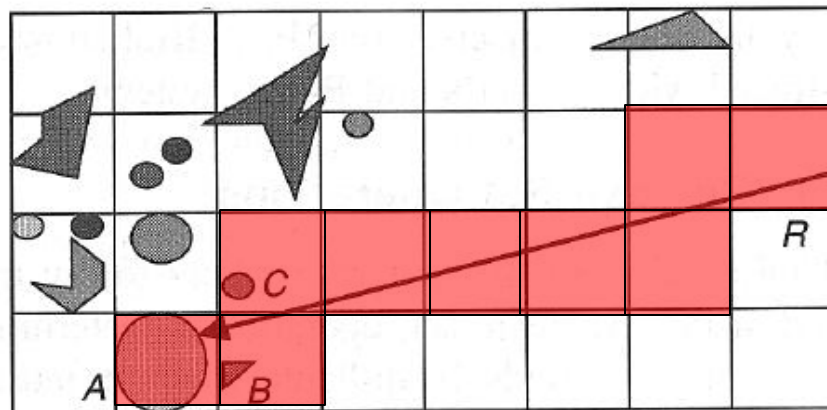
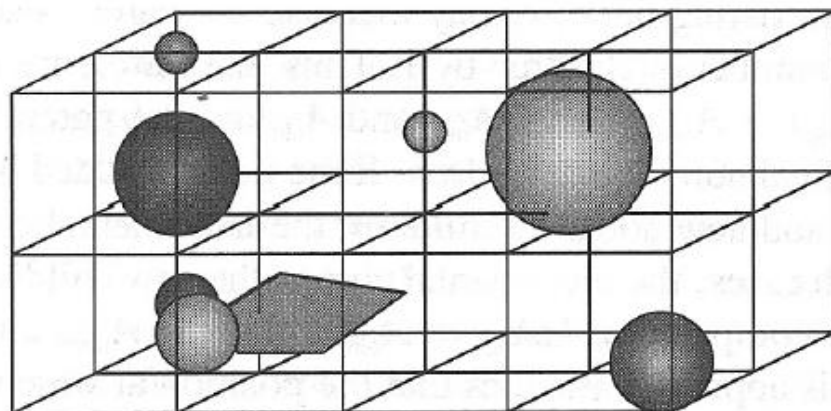




# Регулярная сетка

## ⌘ 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

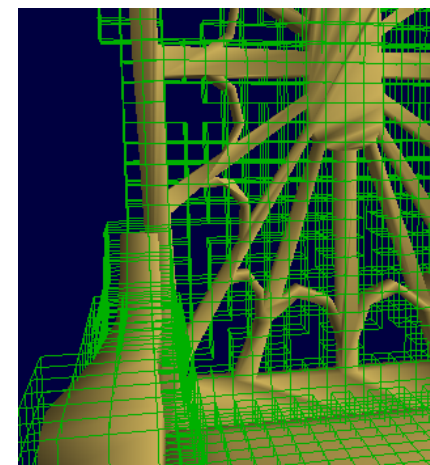
## ⌘ Преимущества

- ☑ Просто и быстро строится
- ☑ Простой алгоритм траверса

## ⌘ Недостатки

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

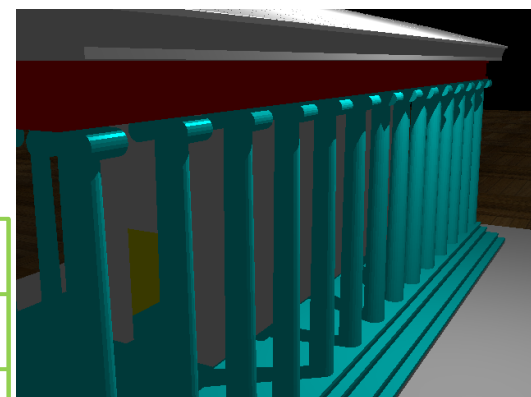
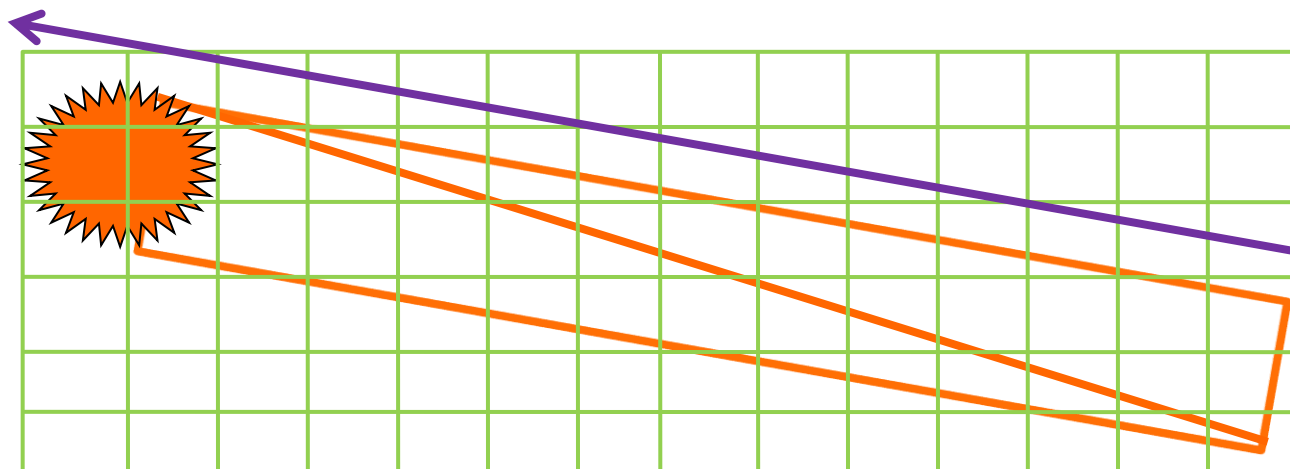
## ⌘ Только для небольших сцен (1-50К)





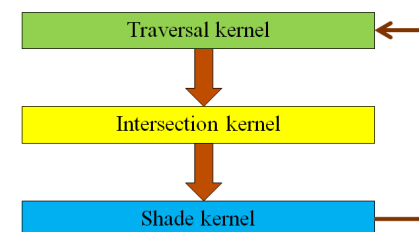
# Регулярная сетка

⌘ Почему сетка плохо разбивает геометрию?



⌘ Перебрали 15 вокселей

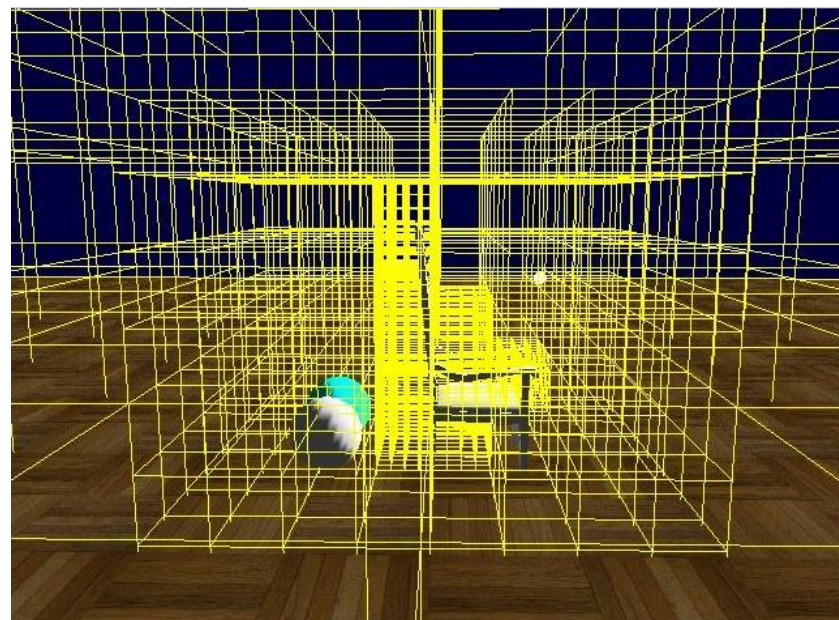
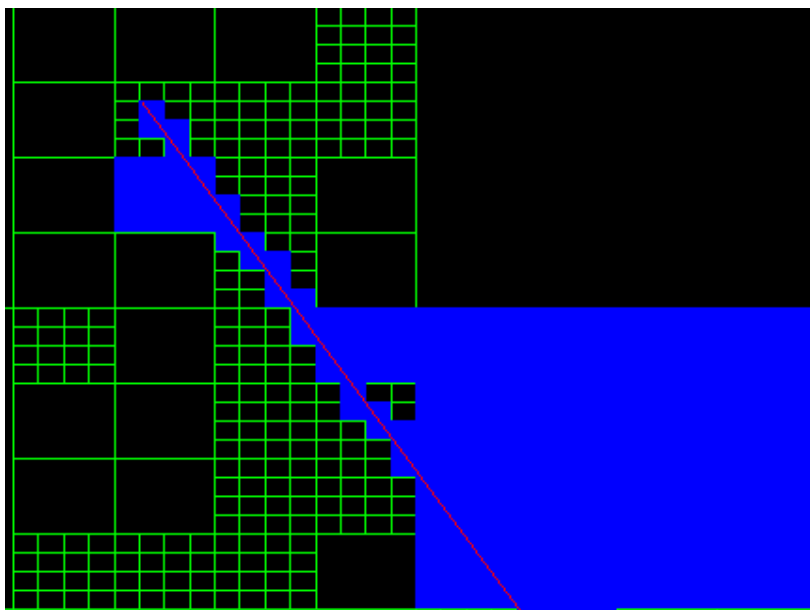
⌘ 7 раз посчитали пересечение с одним и тем же треугольником!



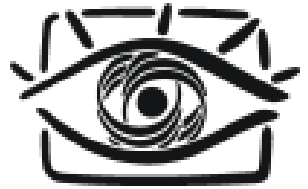


# Иерархическая сетка

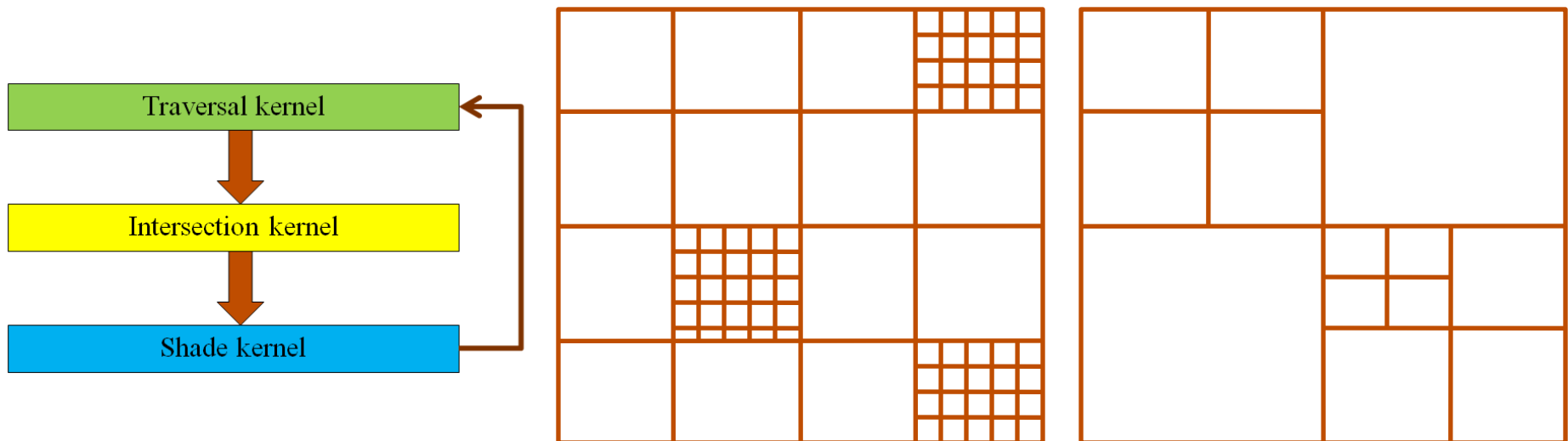
- ⌘ Небольшое число вокселей
- ⌘ Рекурсивно разбиваем воксели в местах с плотной геометрией



# Что дает иерархическая сетка?



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

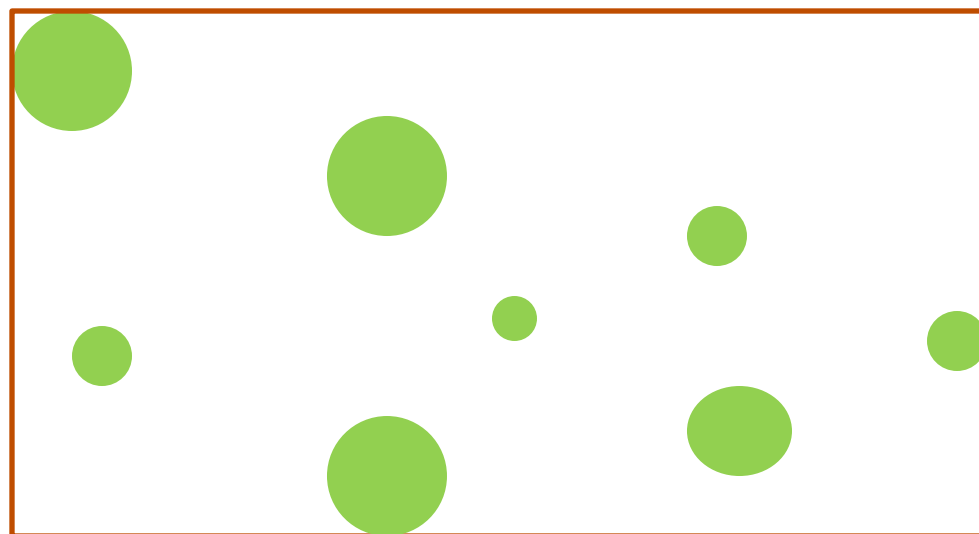






# BVH деревья

## ⌘ Bounding Volume Hierarchy

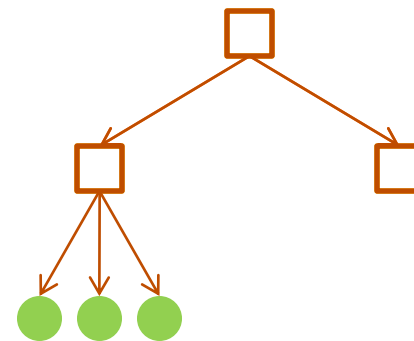
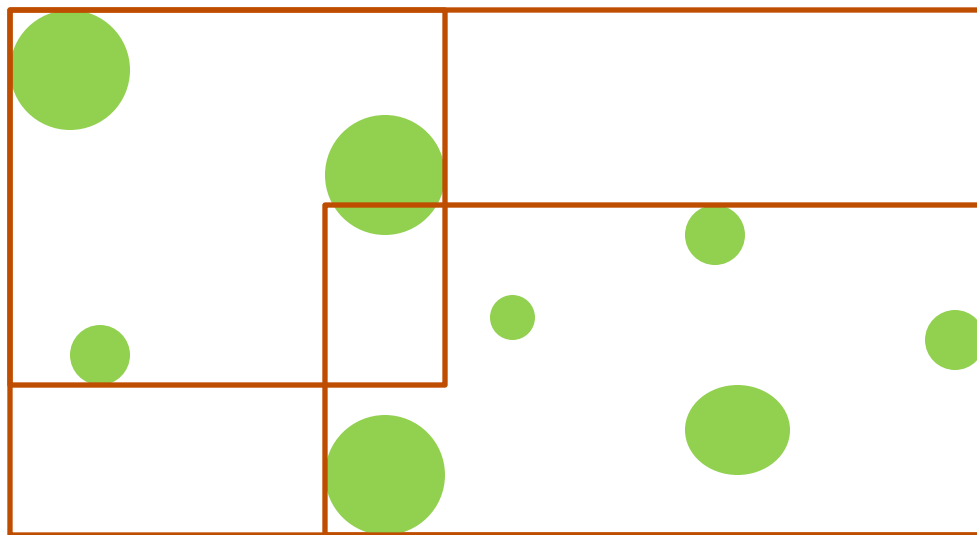






# BVH деревья

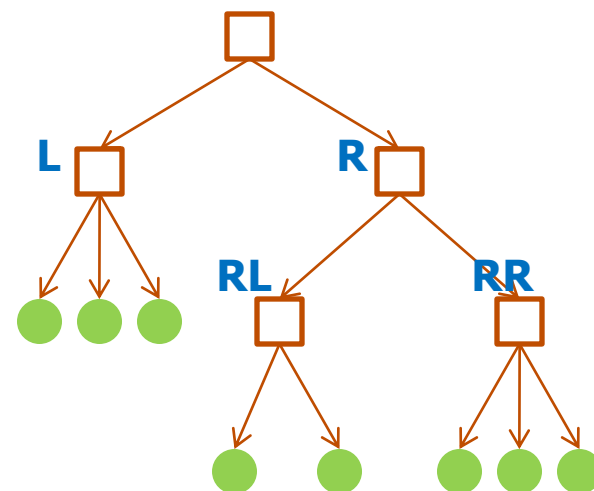
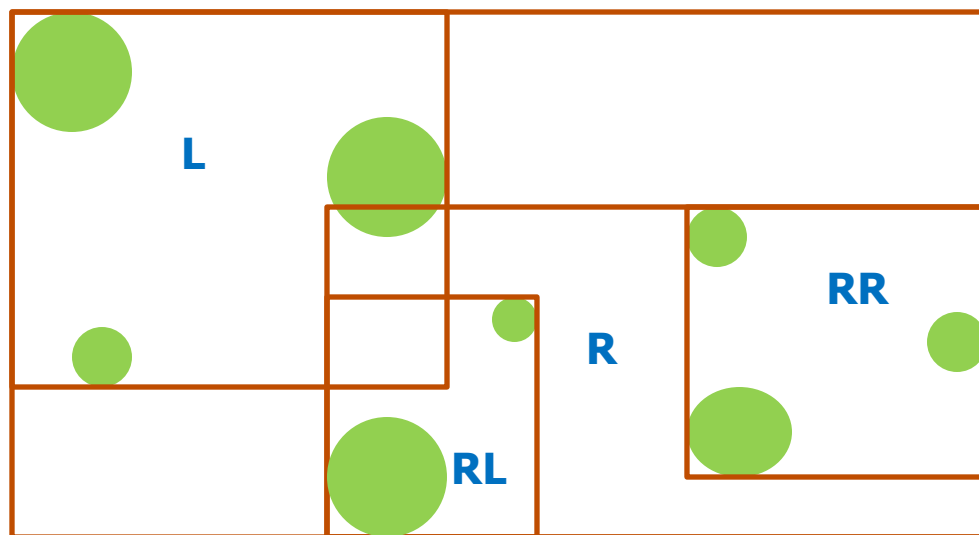
## ⌘ Bounding Volume Hierarchy





# ВУН деревья

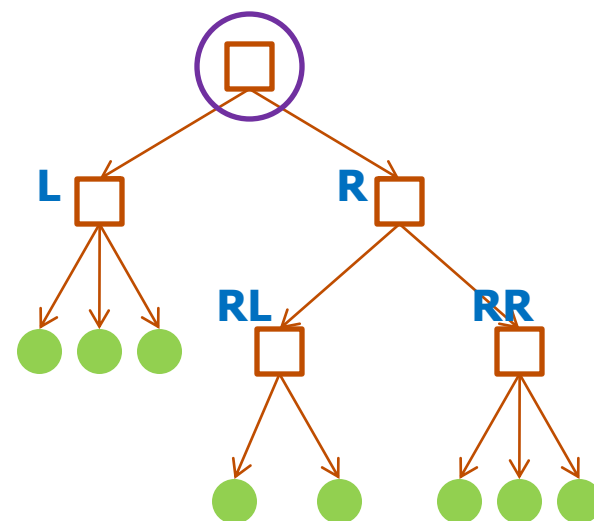
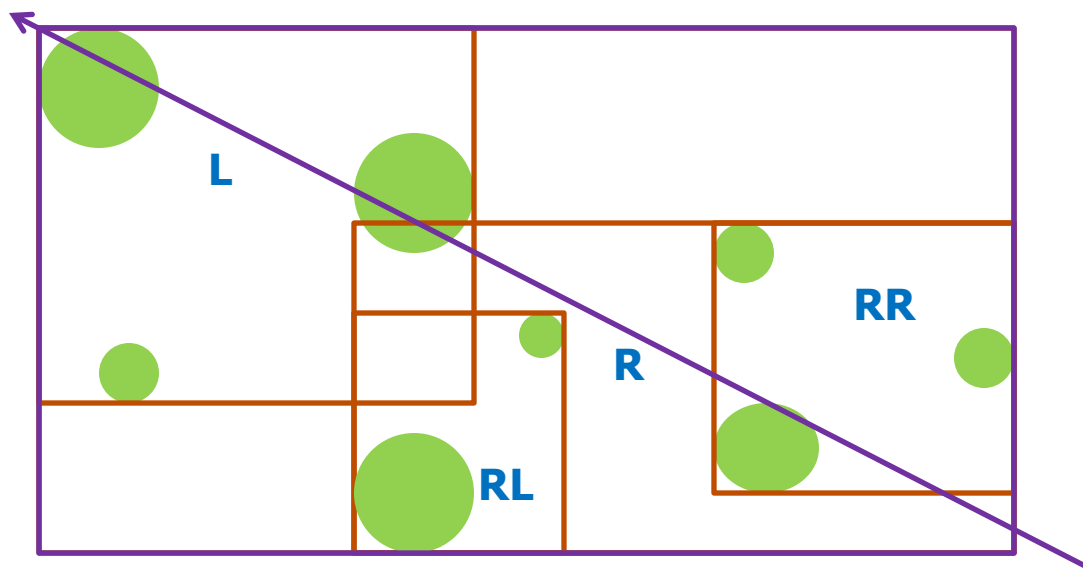
## ⌘ Bounding Volume Hierarchy





# BVH деревья

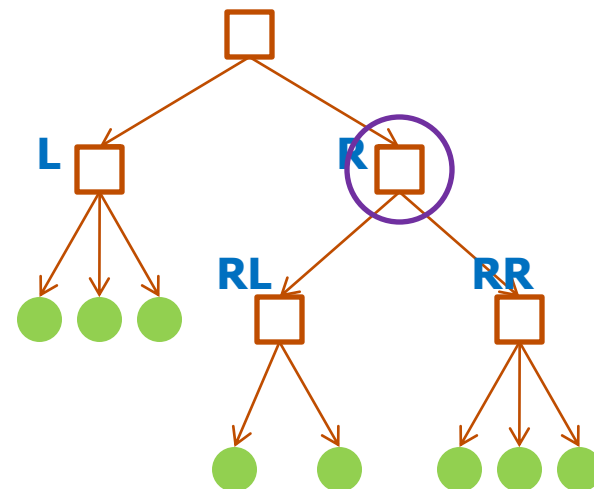
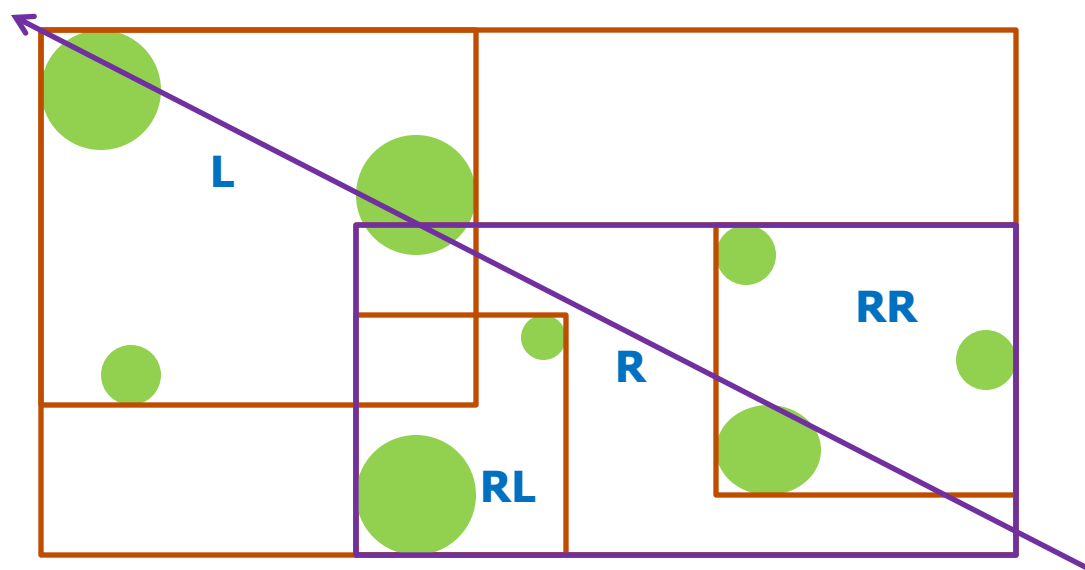
## ✂ Траверс на CPU





# BVH деревья

## ⌘ Траверс на CPU

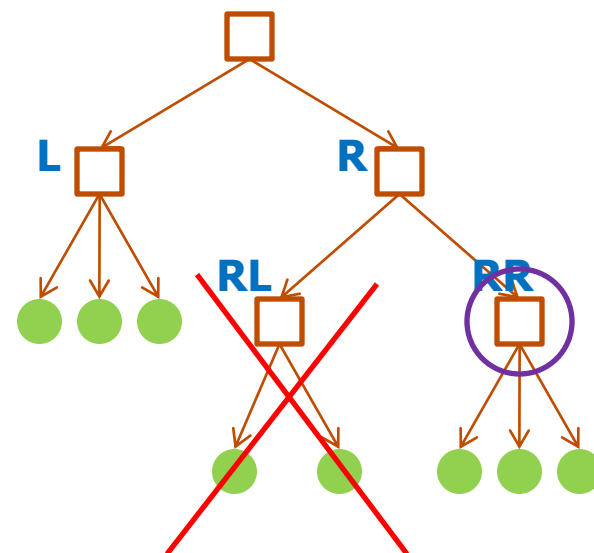
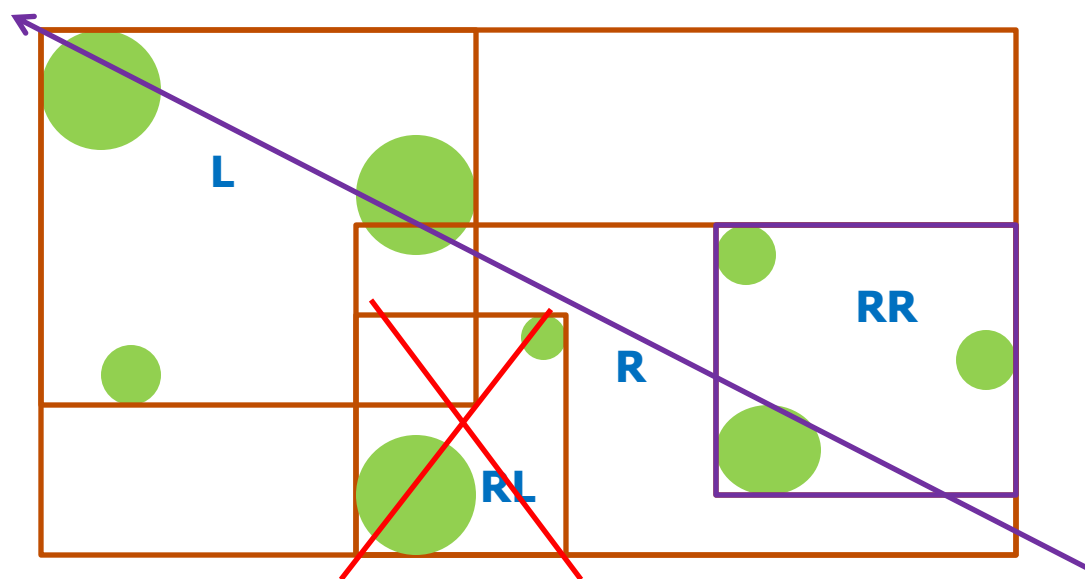


Стек: L



# BVH деревья

## ⌘ Траверс на CPU

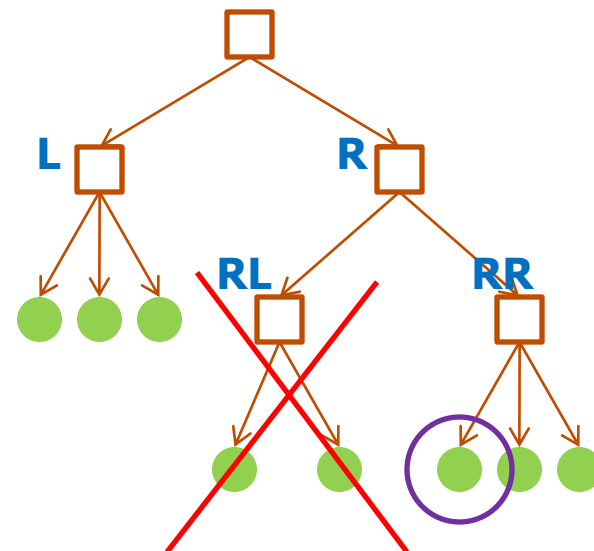
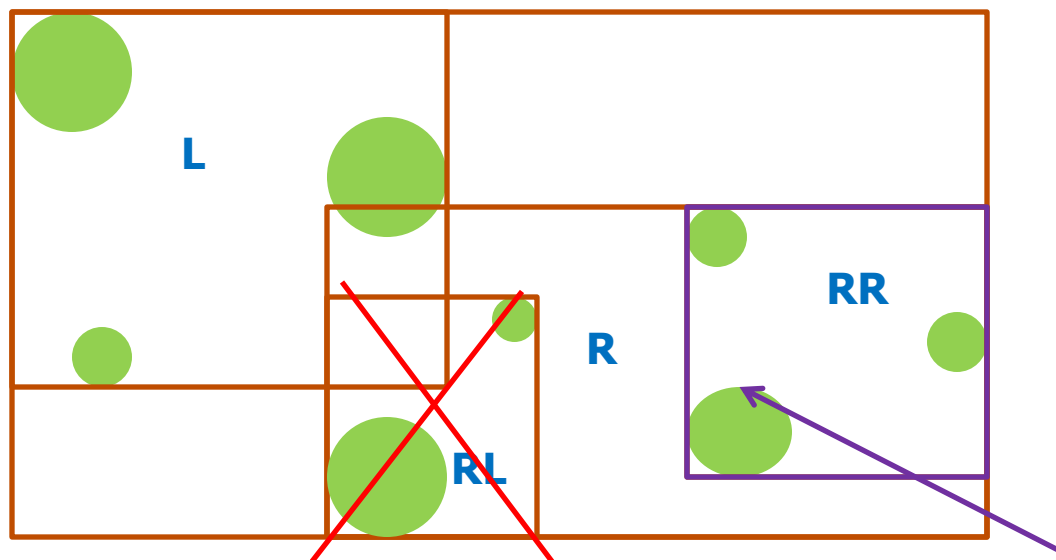


Стек: L



# BVH деревья

## ✂ Траверс на CPU

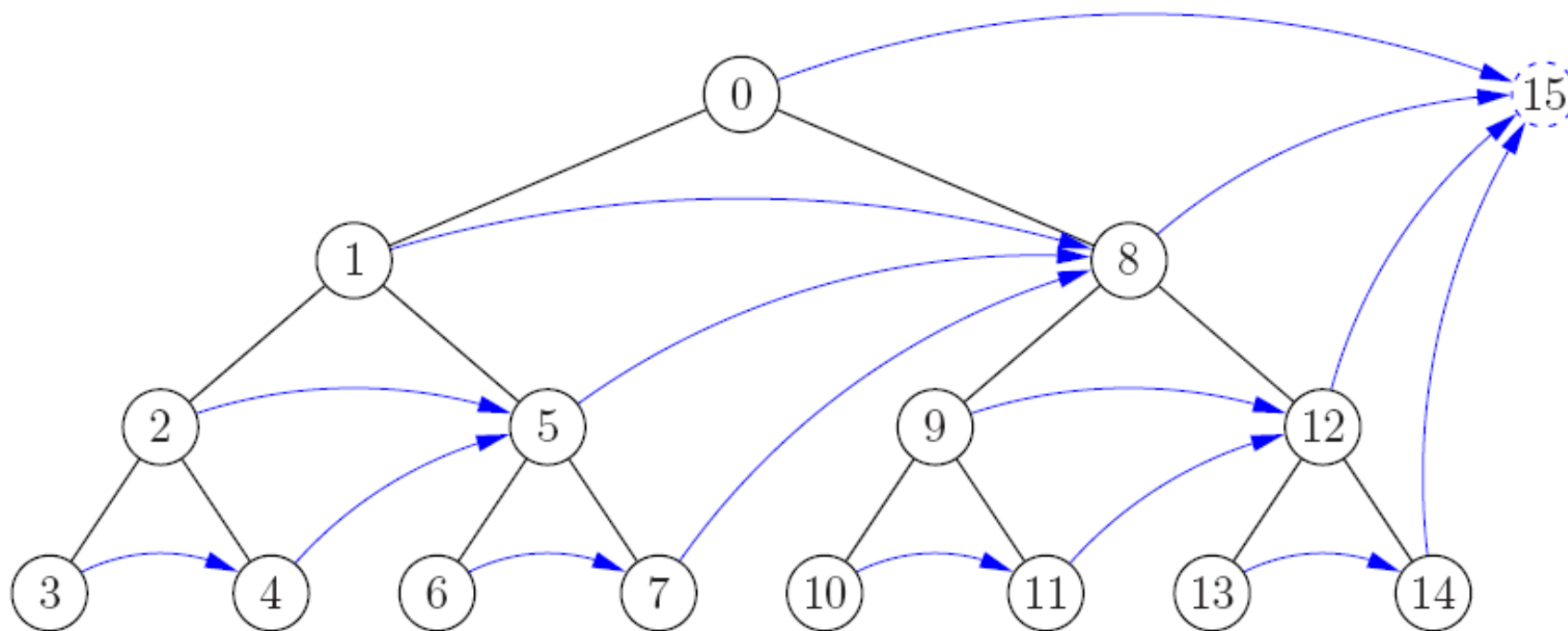


Стек: L



# ВУН деревья

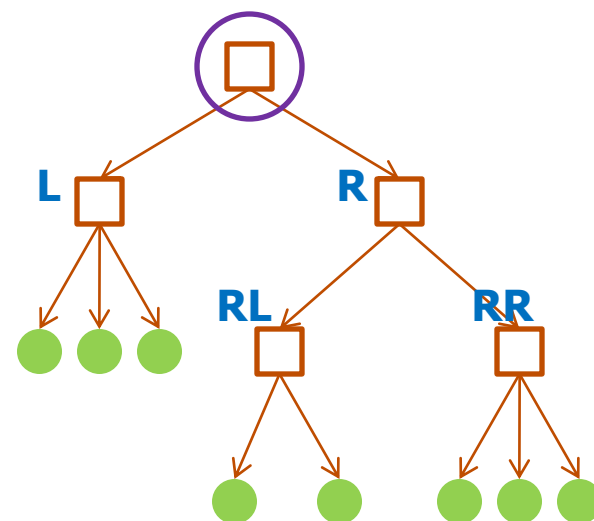
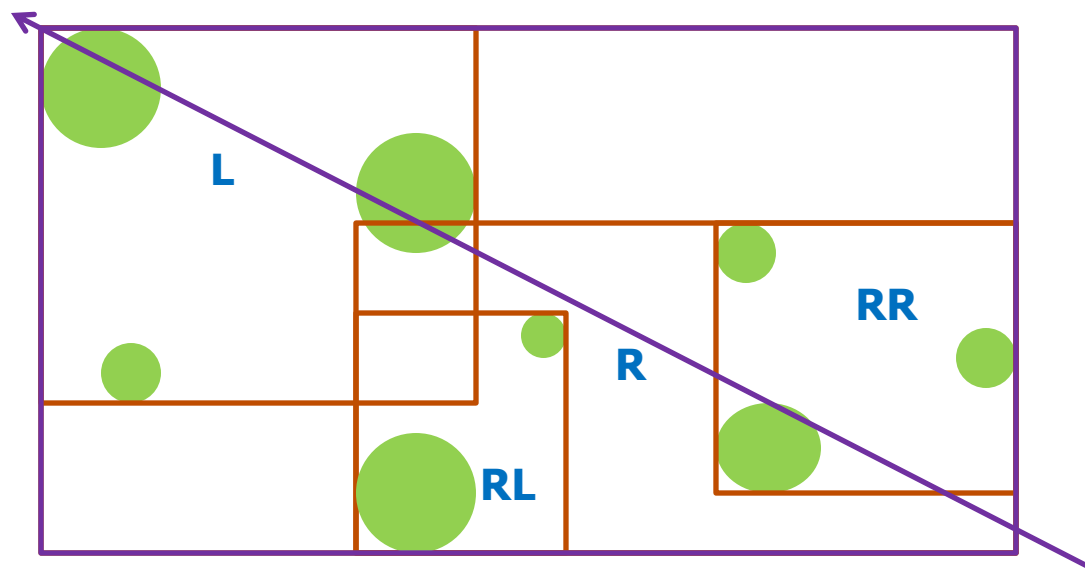
⌘ Траверс на GPU





# BVH деревья

## ⌘ Траверс на GPU

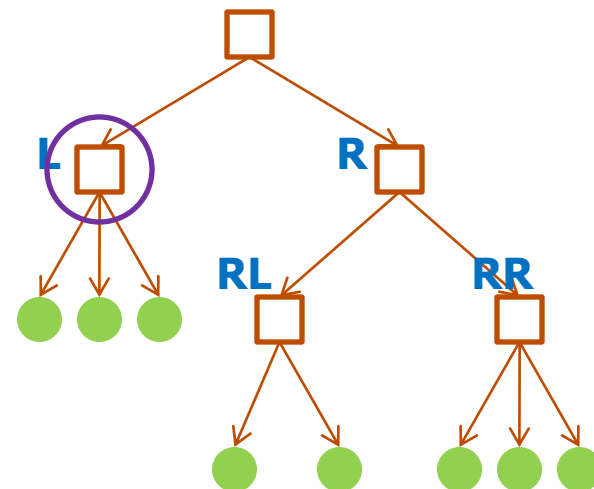
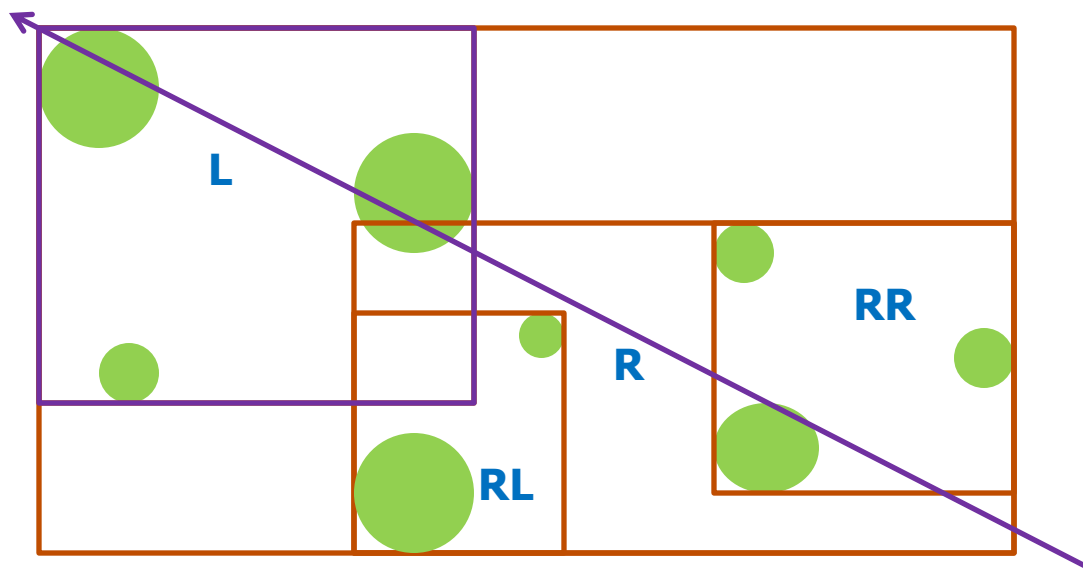






# BVH деревья

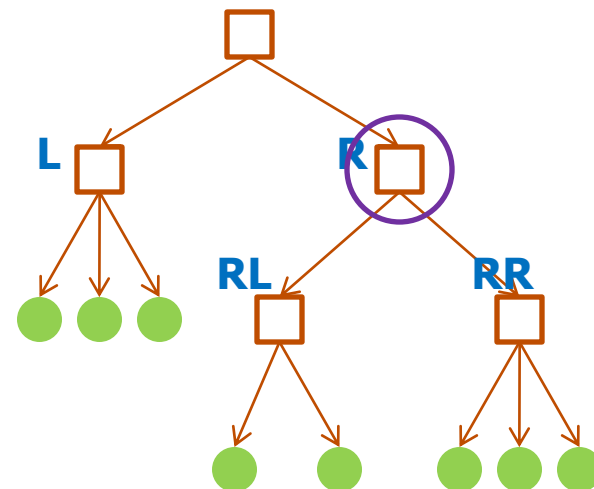
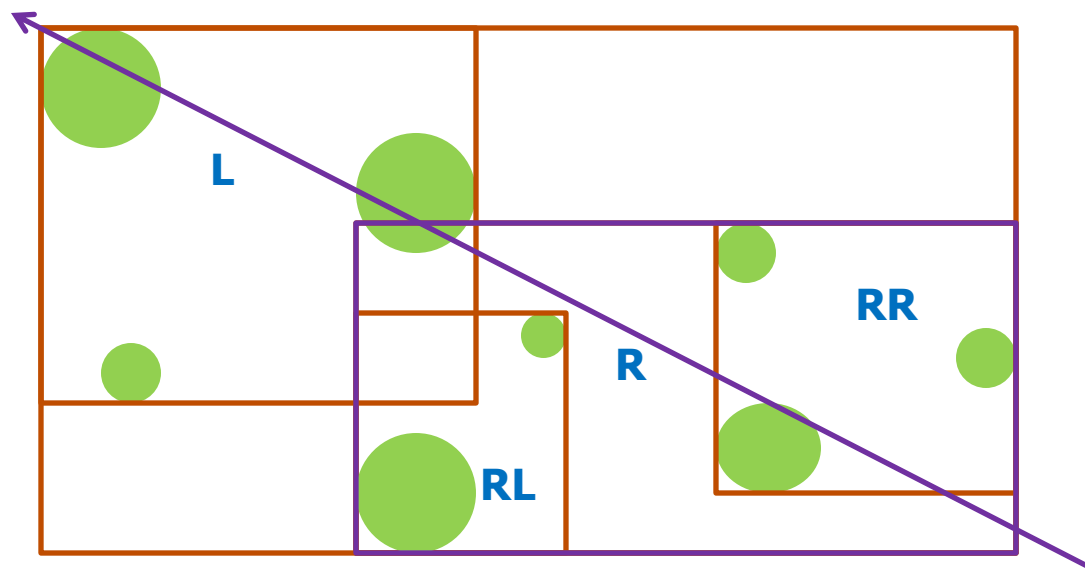
## ⌘ Траверс на GPU





# BVH деревья

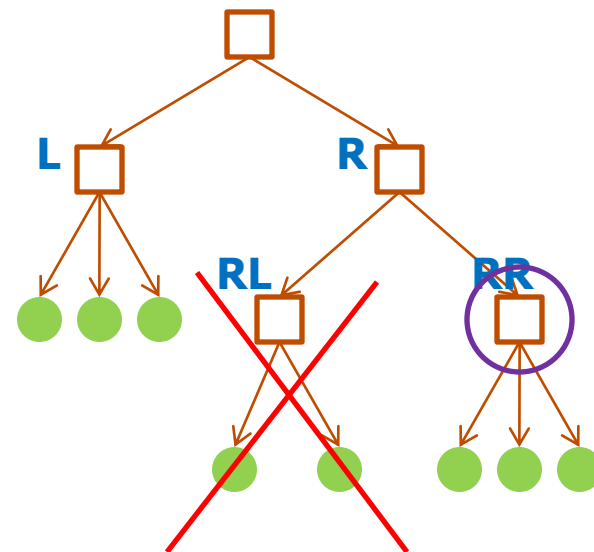
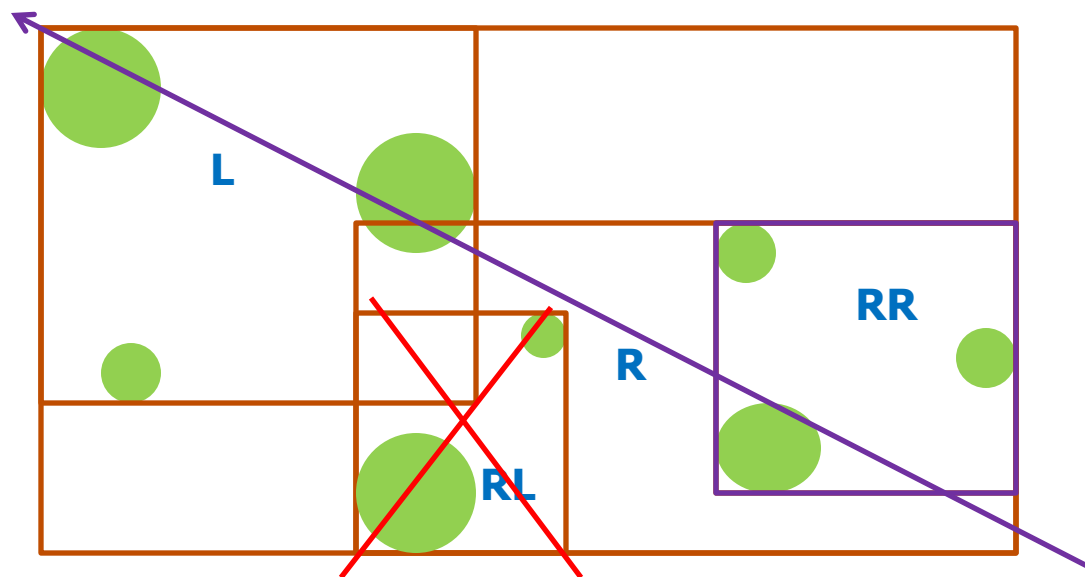
## ⌘ Траверс на GPU





# BVH деревья

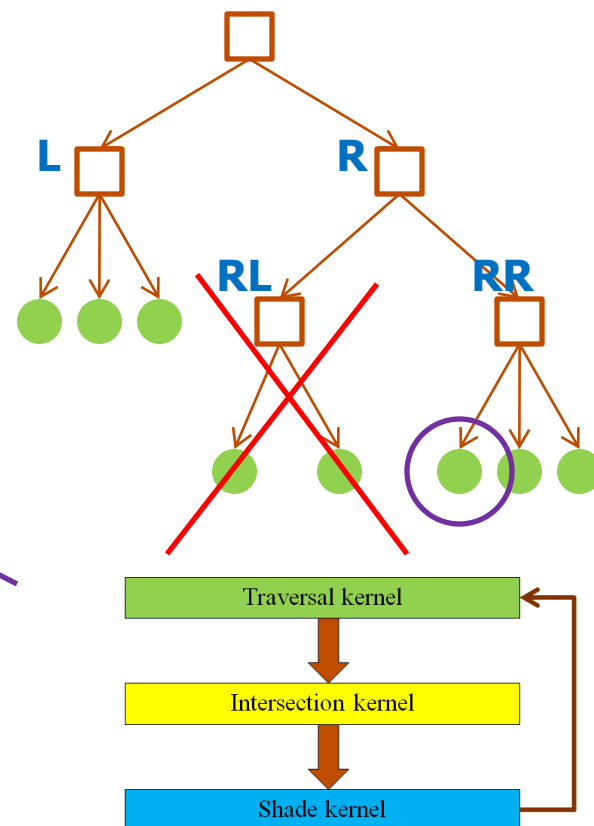
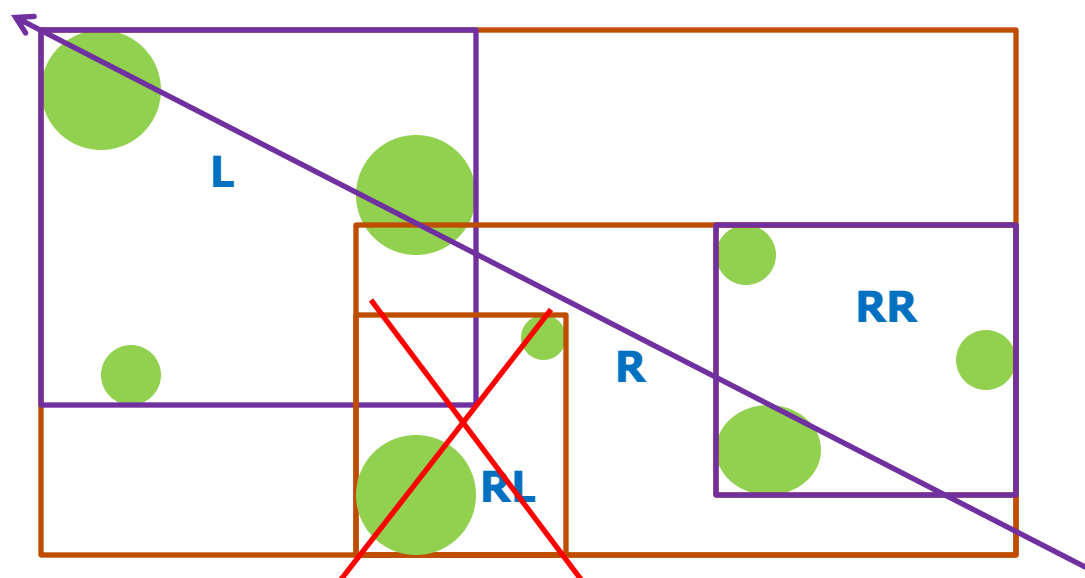
## ⌘ Траверс на GPU





# BVH деревья

## ⌘ Траверс на GPU





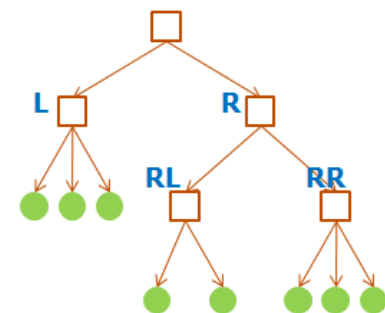
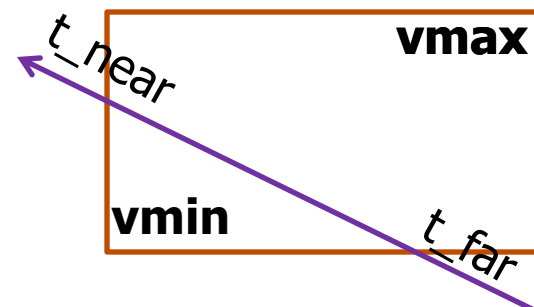
# BVH деревья

## ⌘ 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?

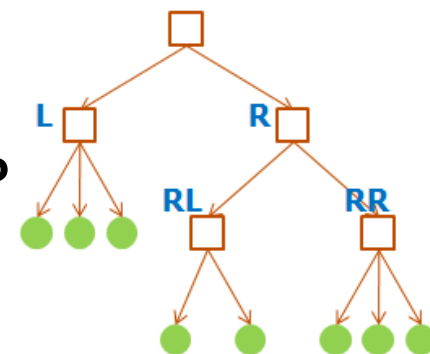
☑ 24 mad-а покрывают латентность текстурной памяти

## 1. Стек на локальной памяти

☑ Локальная память это не так медленно, как может показаться

## 2. Бесстековый алгоритм

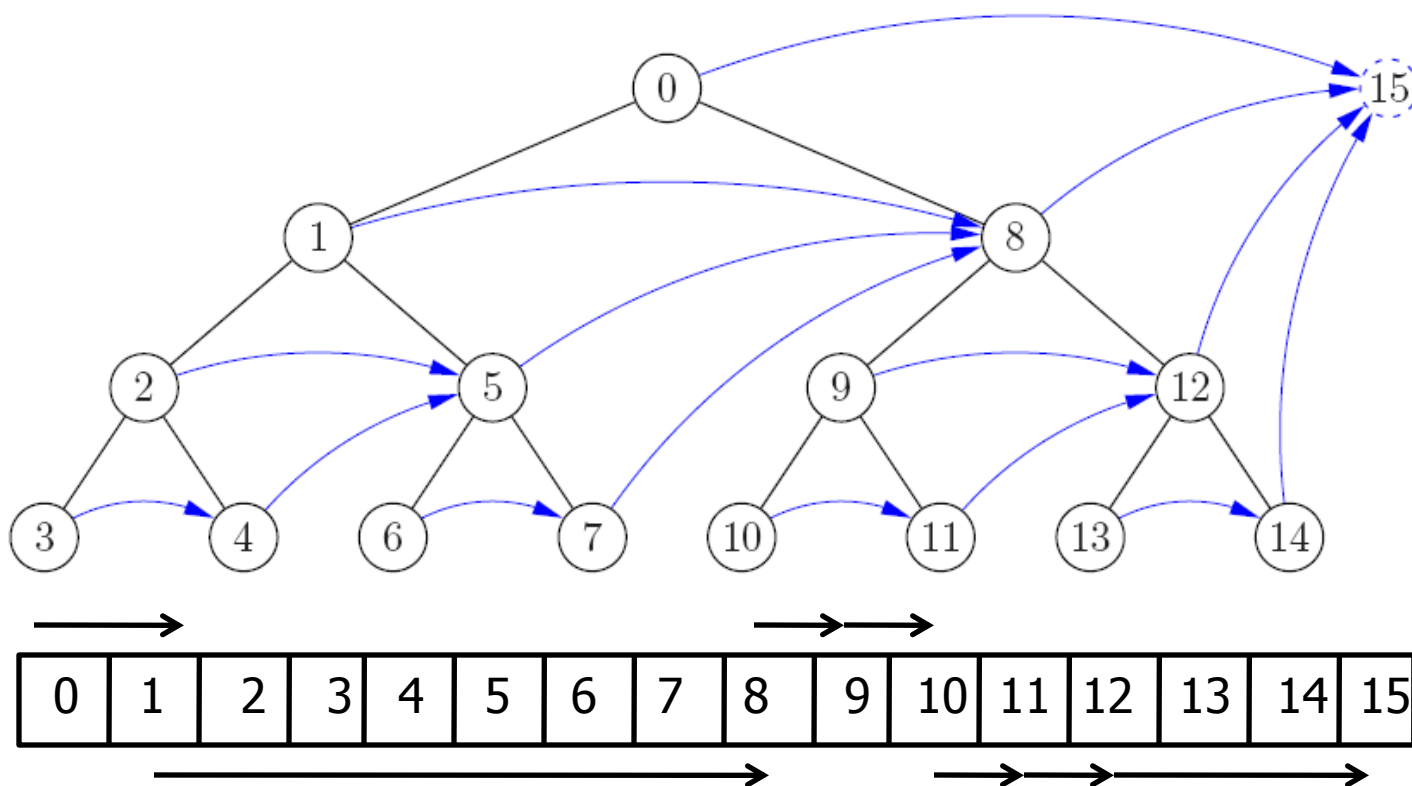
☑ Перебираем массив всегда строго слева направо





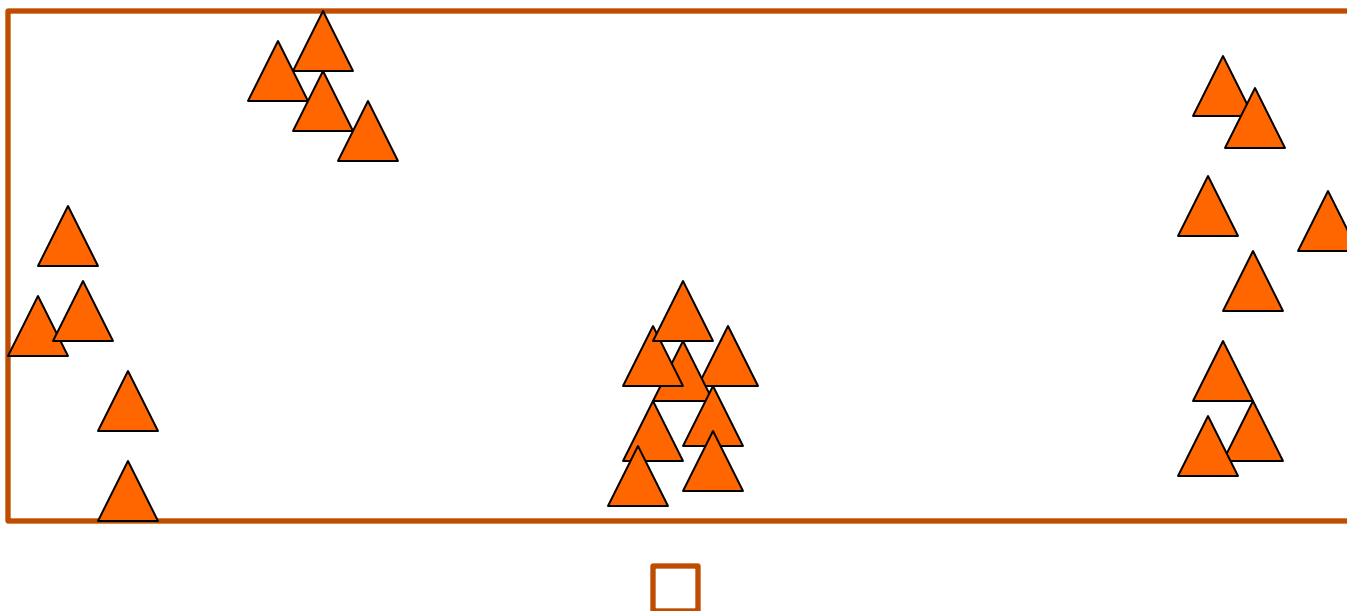
# ВУН деревья

## ⌘ Безстековый траверс на GPU





# kd-деревья

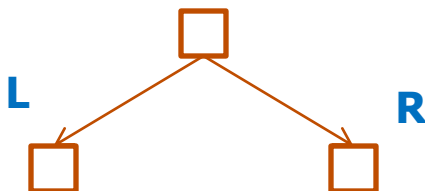
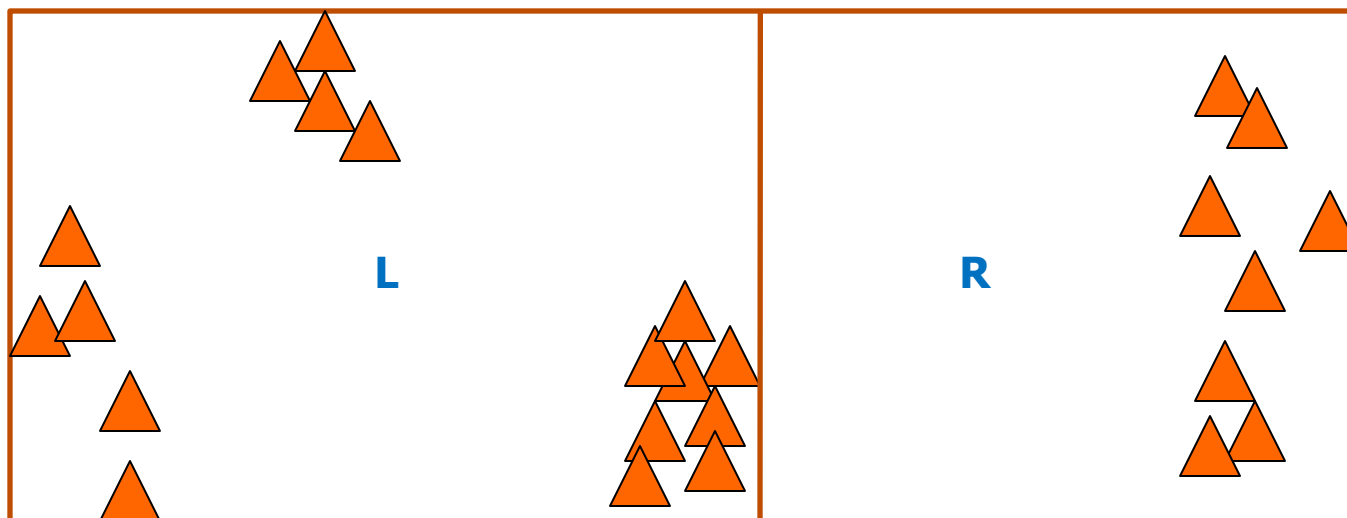


```
struct KdTreeNode
{
    float split;
    uint leftOffset: 29;
    uint splitAxis: 2;
    uint leaf: 1;
};
```





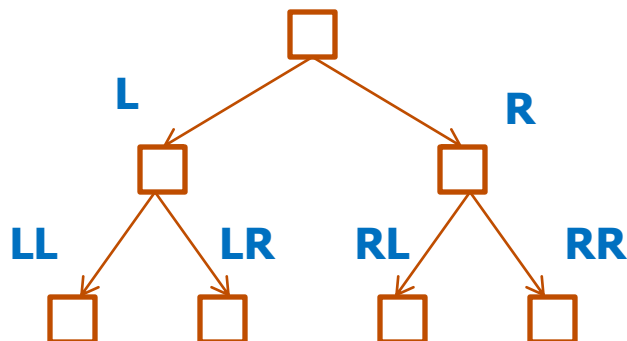
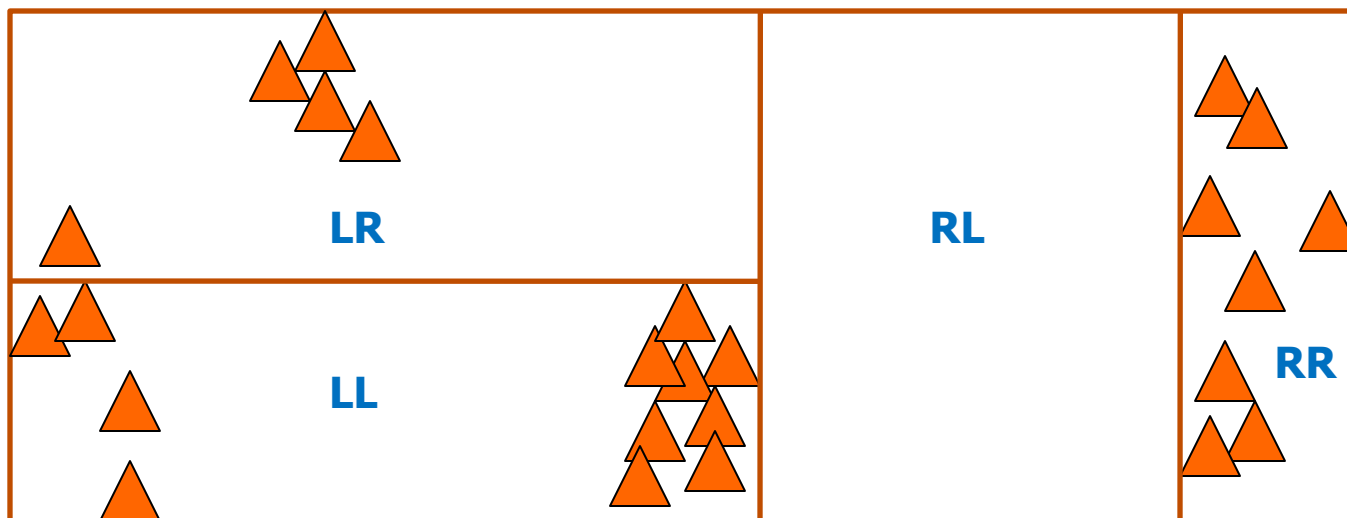
# kd-деревья



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



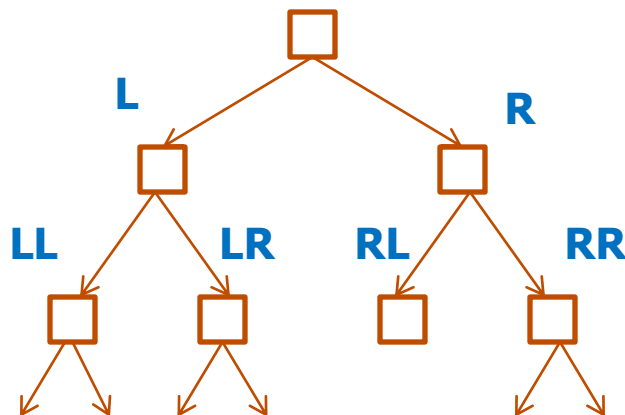
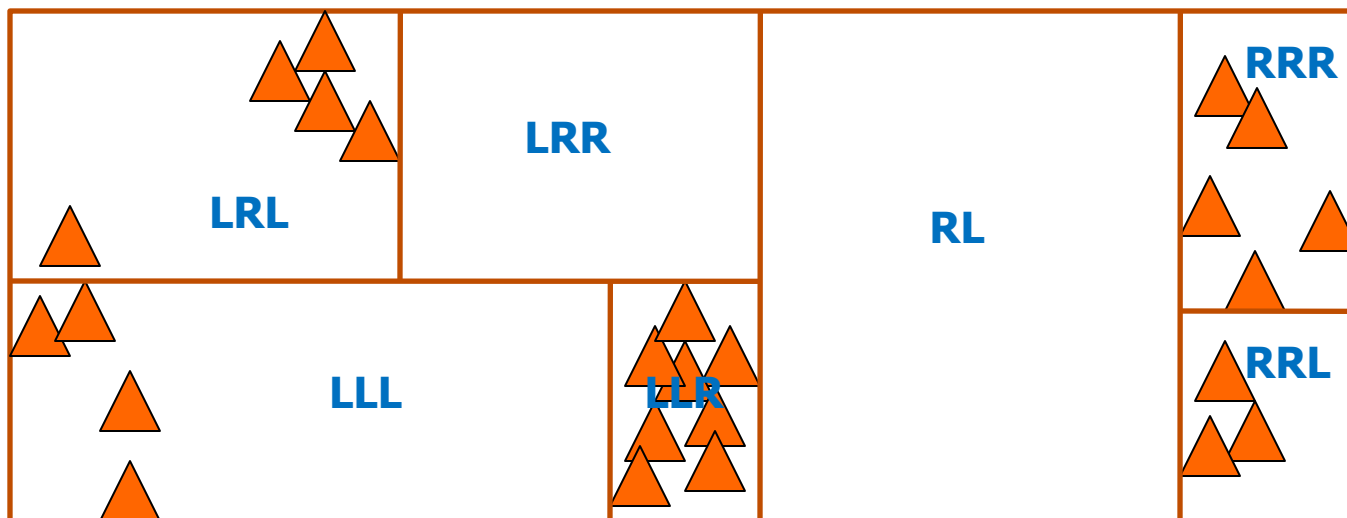
# kd-деревья



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



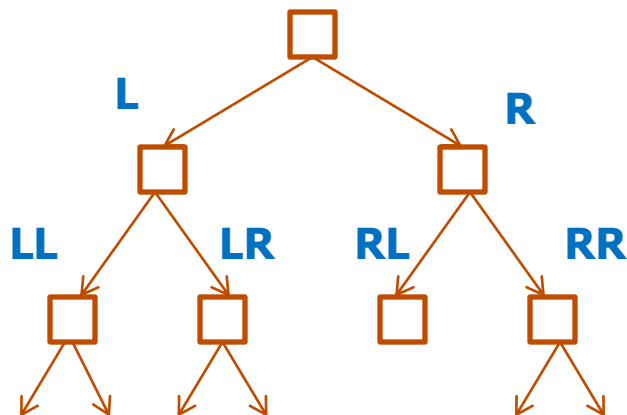
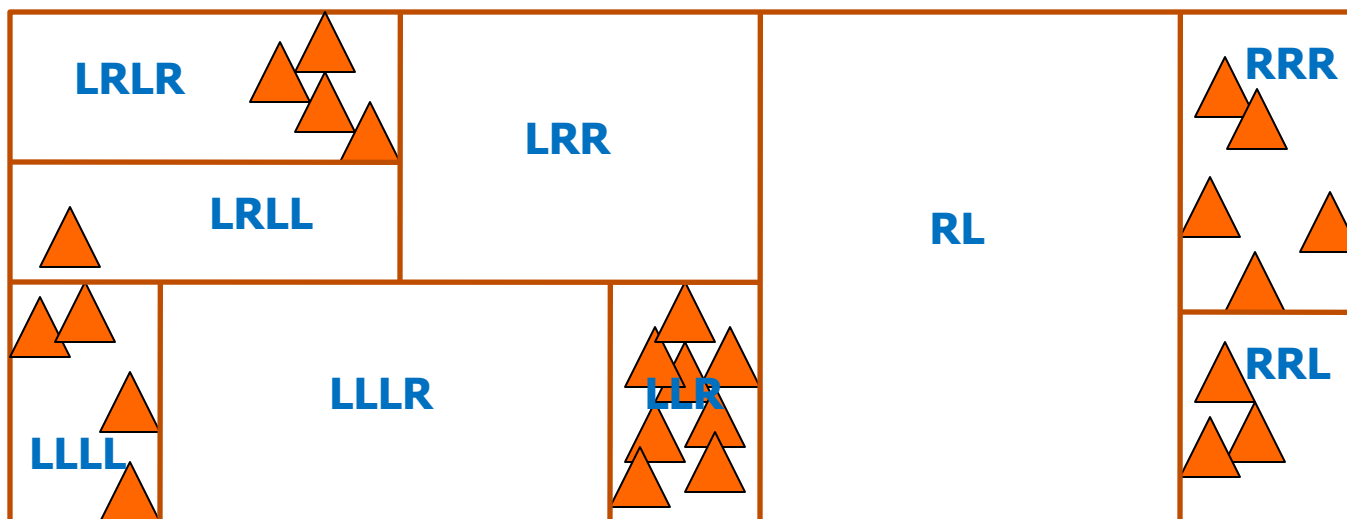
# kd-деревья



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



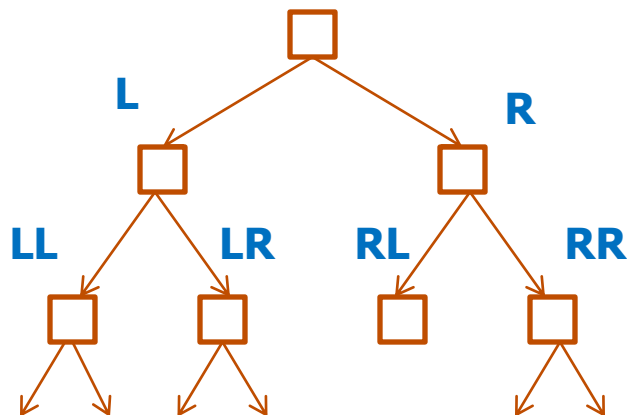
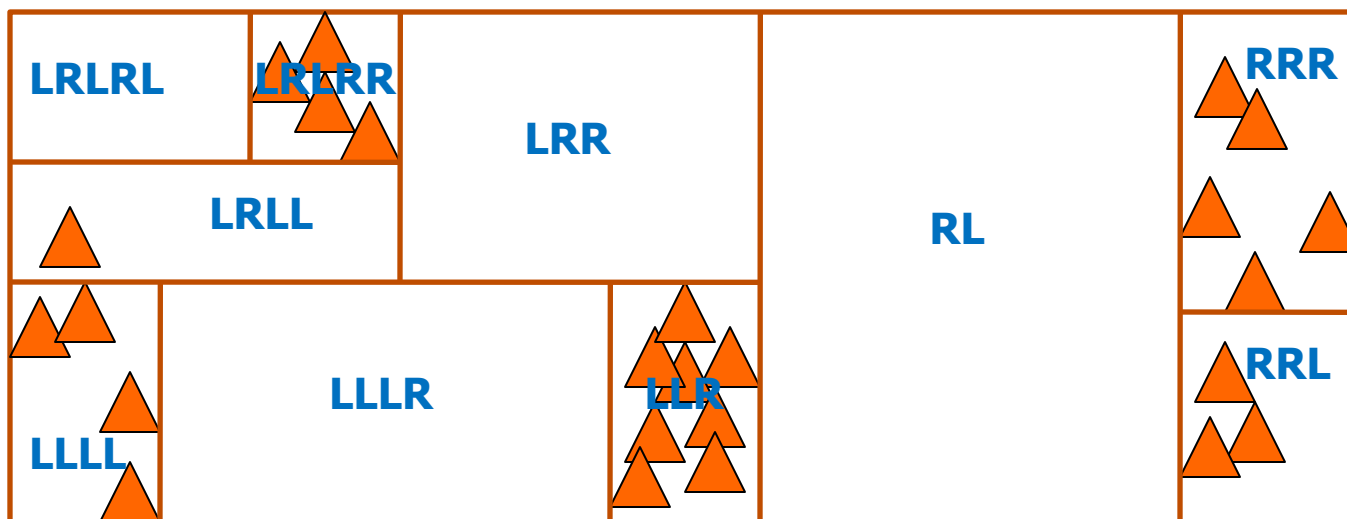
# kd-деревья



```
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

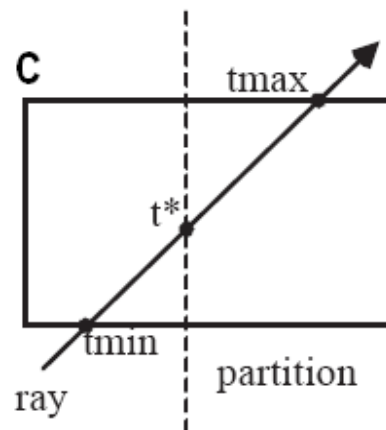
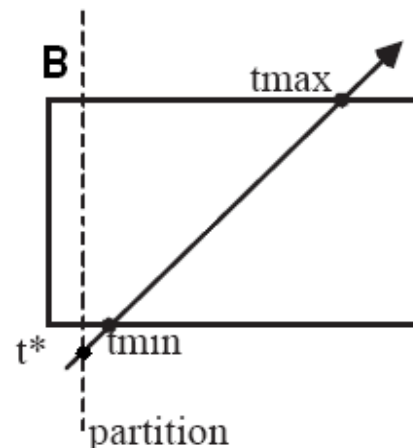
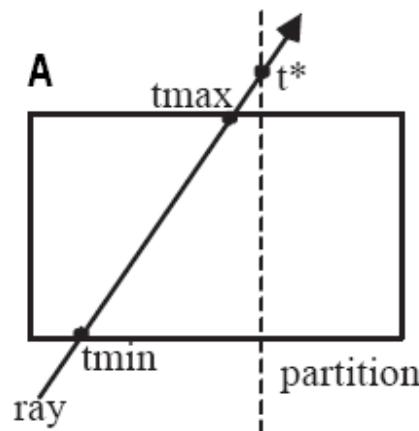
☑  $t$ ,  $t_{\min}$ ,  $t_{\max}$  – 3

☑ node – 2

☑  $t_{id}$ ,  $stack\_top$  – 2

☑ На практике удалось уложиться в 16!

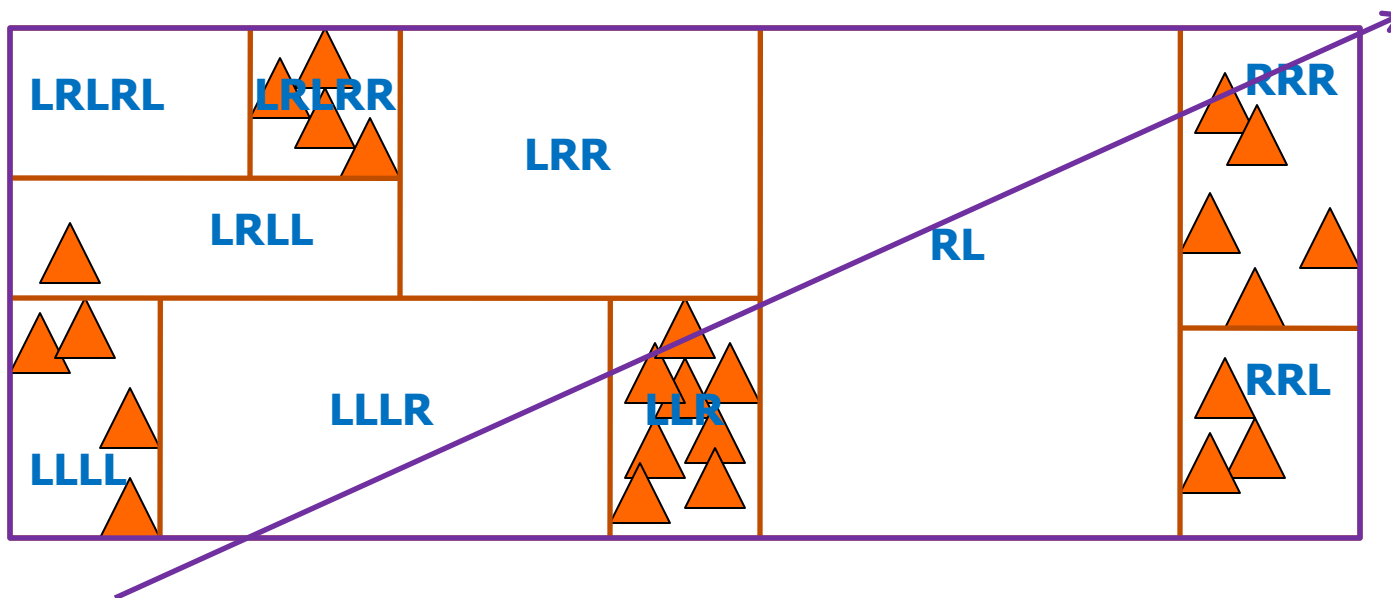
☑ Стек: локальная память





# kd-деревья

## ⌘ Алгоритм траверса



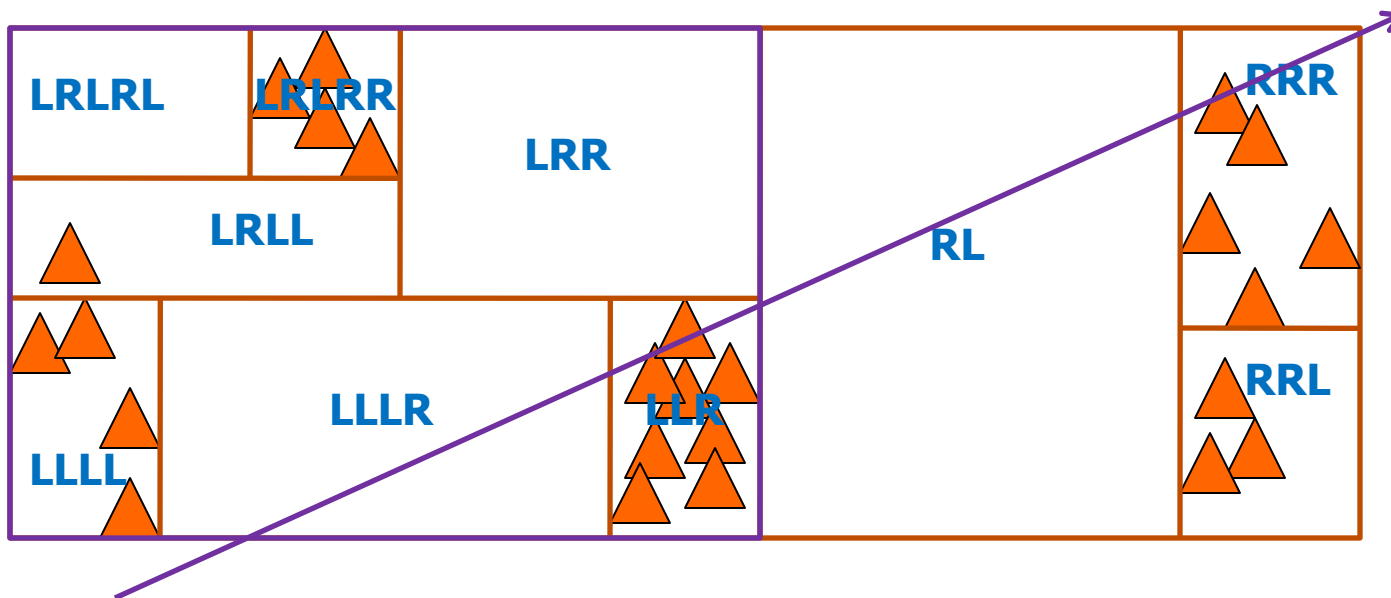
Стек:

Текущий узел:



# kd-деревья

## ⌘ Алгоритм траверса



Стек: **R**

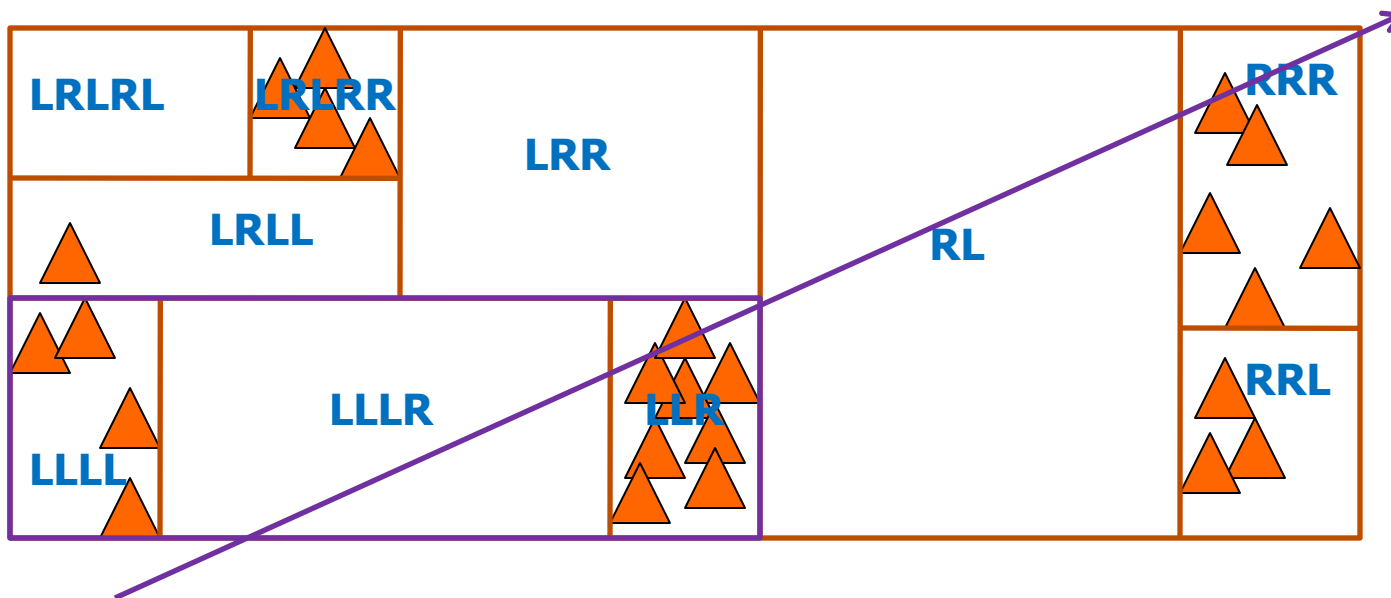
Текущий узел: **L**





# kd-деревья

## ⌘ Алгоритм траверса



Стек: **R**

Текущий узел: **LL**

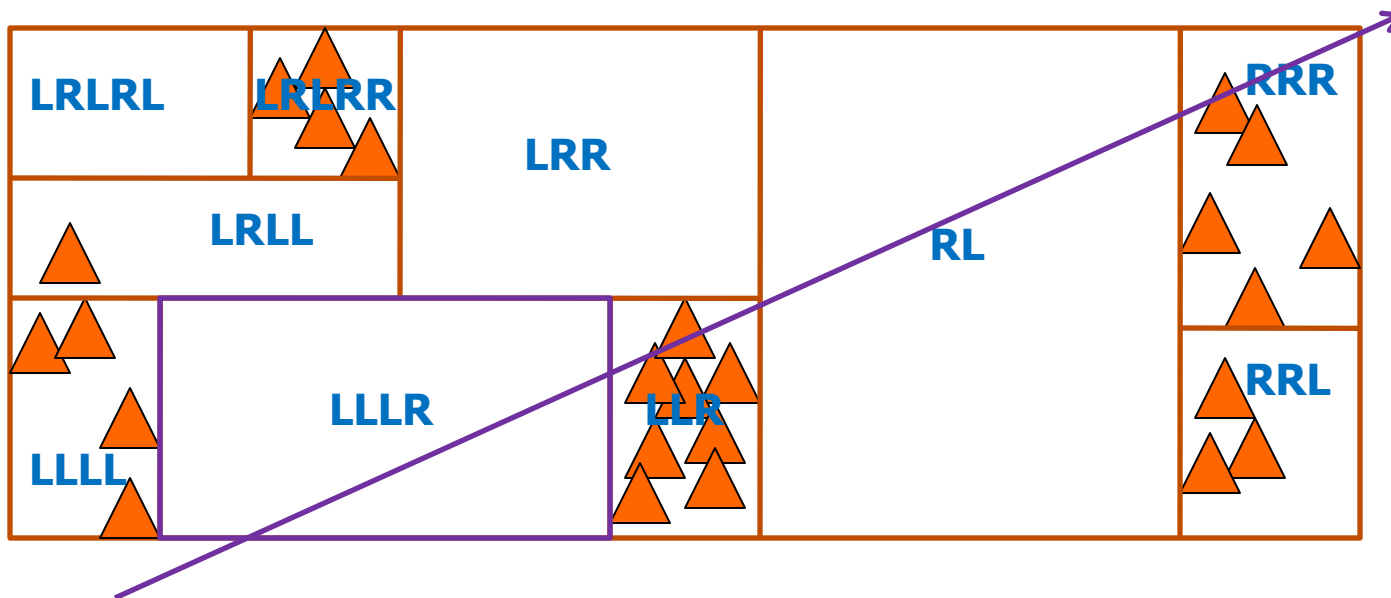


Текущий узел: **LLL**



# kd-деревья

## ⌘ Алгоритм траверса

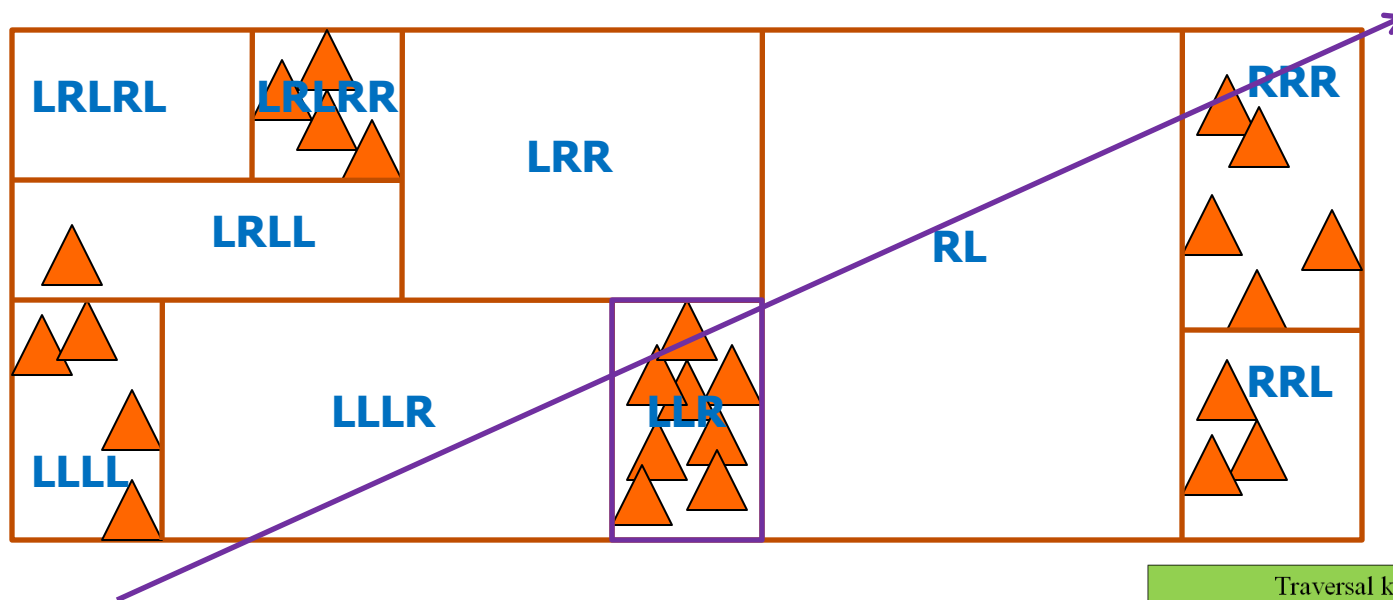


Стек: **LLR, R**  
→ Текущий узел: **LLL**



# kd-деревья

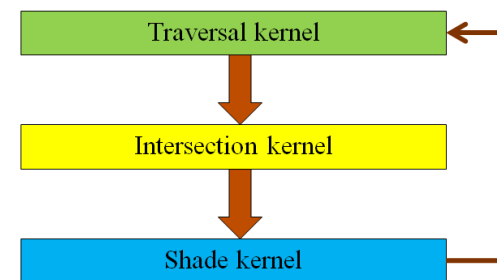
## ⌘ Алгоритм траверса



Стек: **R**

→ Текущий узел: **LLR**

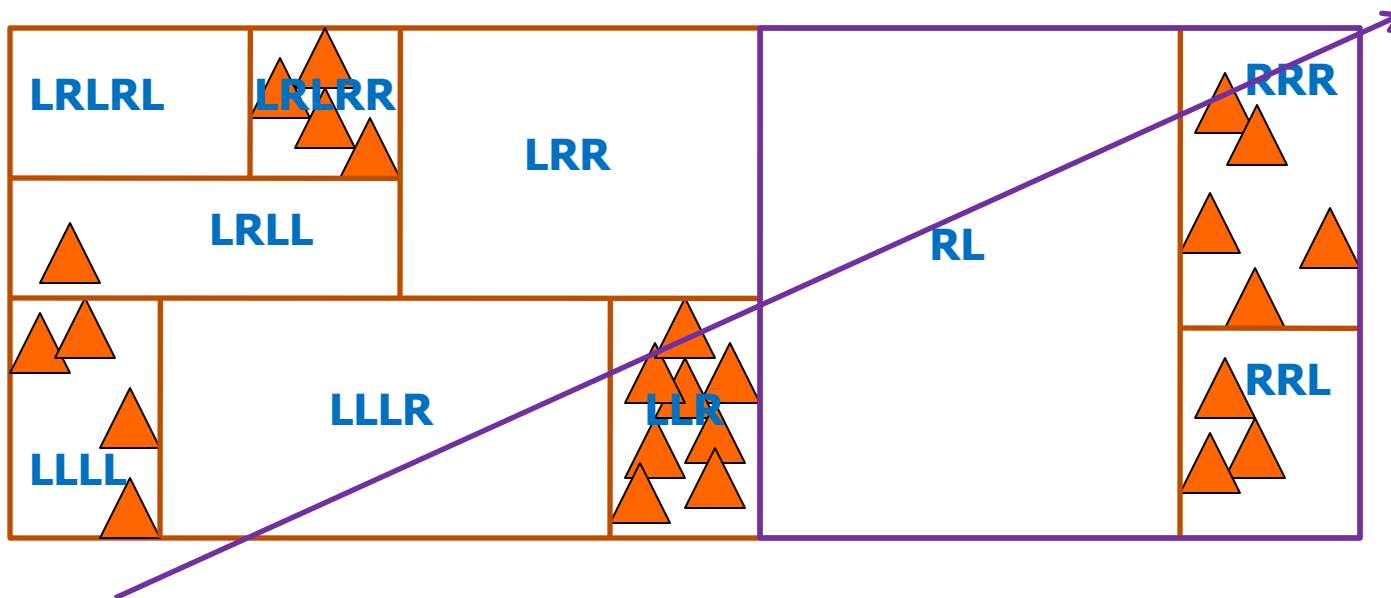
Можно было бы остановиться!





# kd-деревья

## ⌘ Алгоритм траверса



Стек:

Текущий узел: **R**

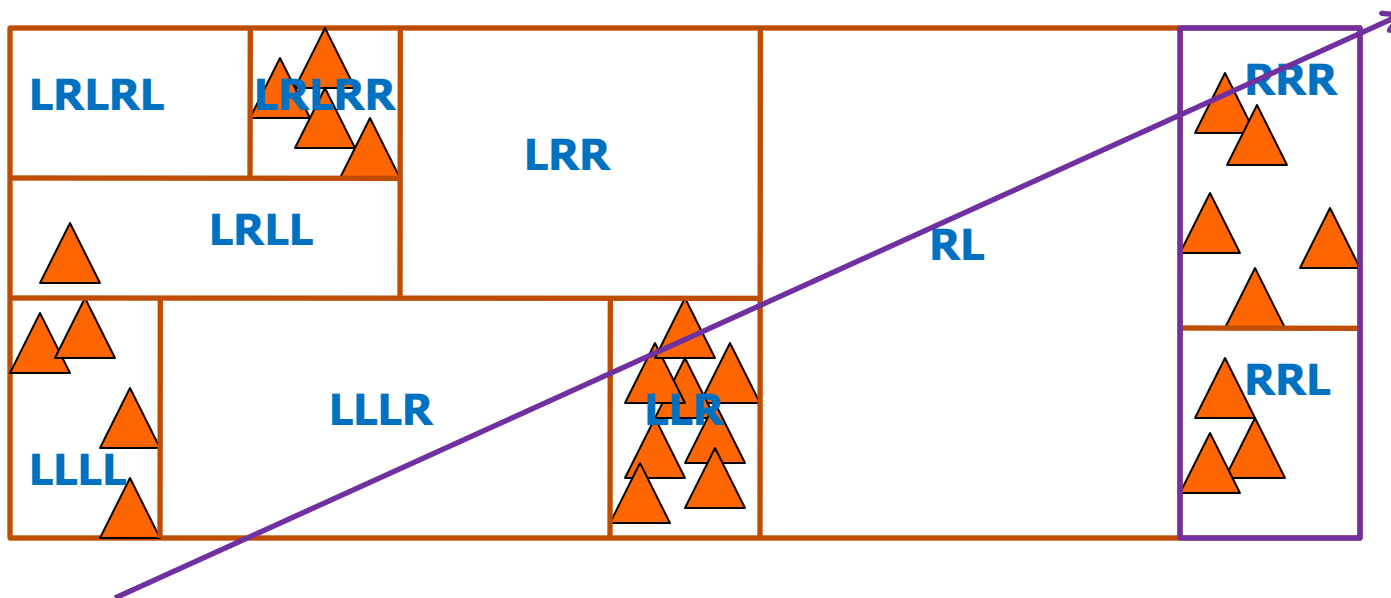


Стек: **RR**  
Текущий узел: **RL**



# kd-деревья

## ⌘ Алгоритм траверса



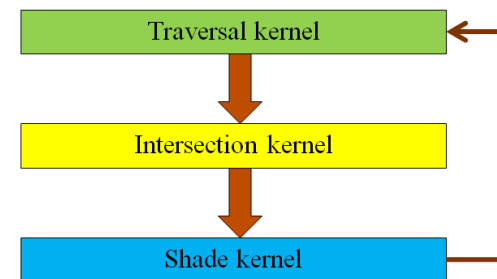
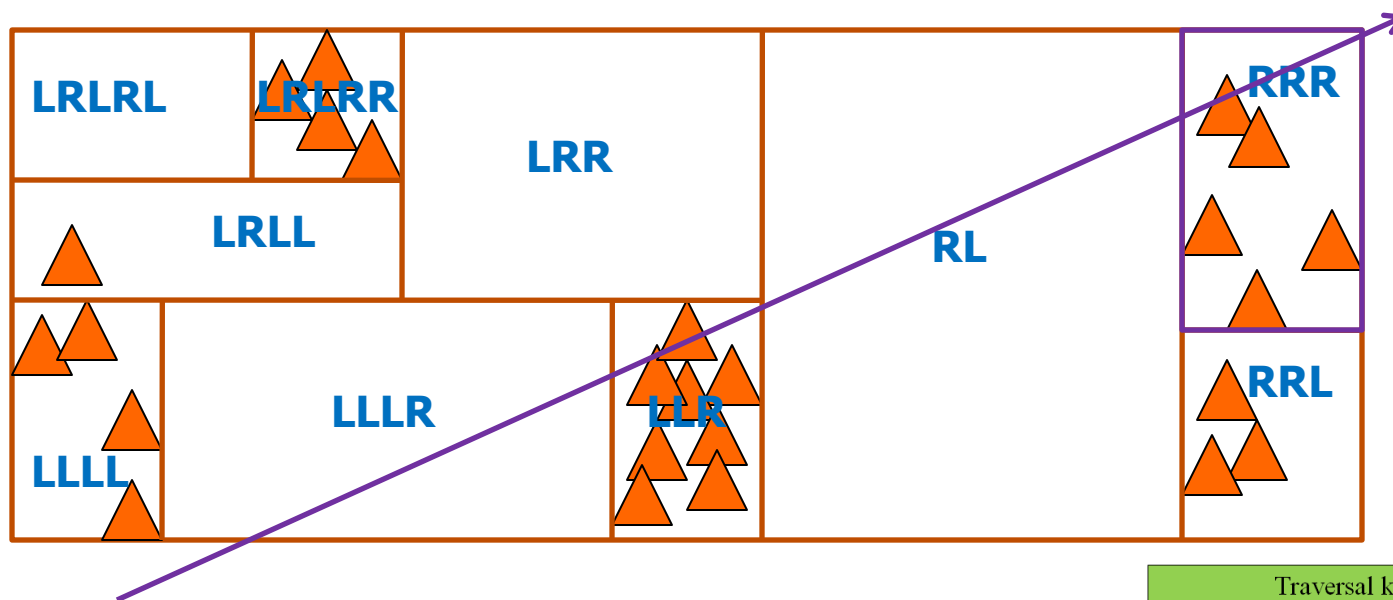
Стек:

Текущий узел: **RR**



# kd-деревья

## ⌘ Алгоритм траверса



Стек:

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

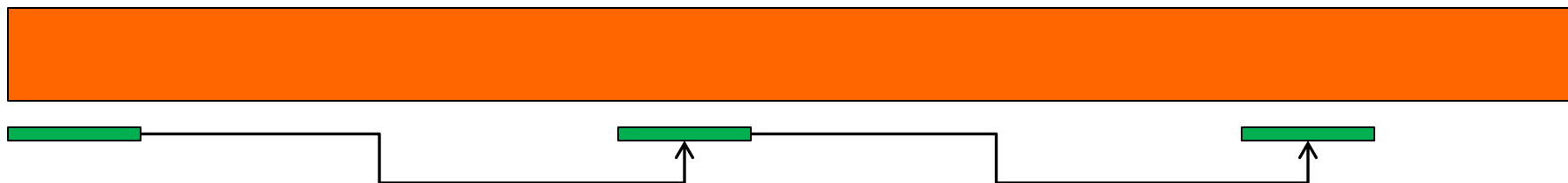




# Производительность

## ⌘ Persistent threads

```
__global__ void my_kernel()  
{  
    for (int warp = 0; warp < 16; warp++)  
    {  
        int tid = blockDim.x*blockIdx.x + threadIdx.x + warp*step;  
        ... // тут код вашего кёрнела  
    }  
}
```





# Производительность

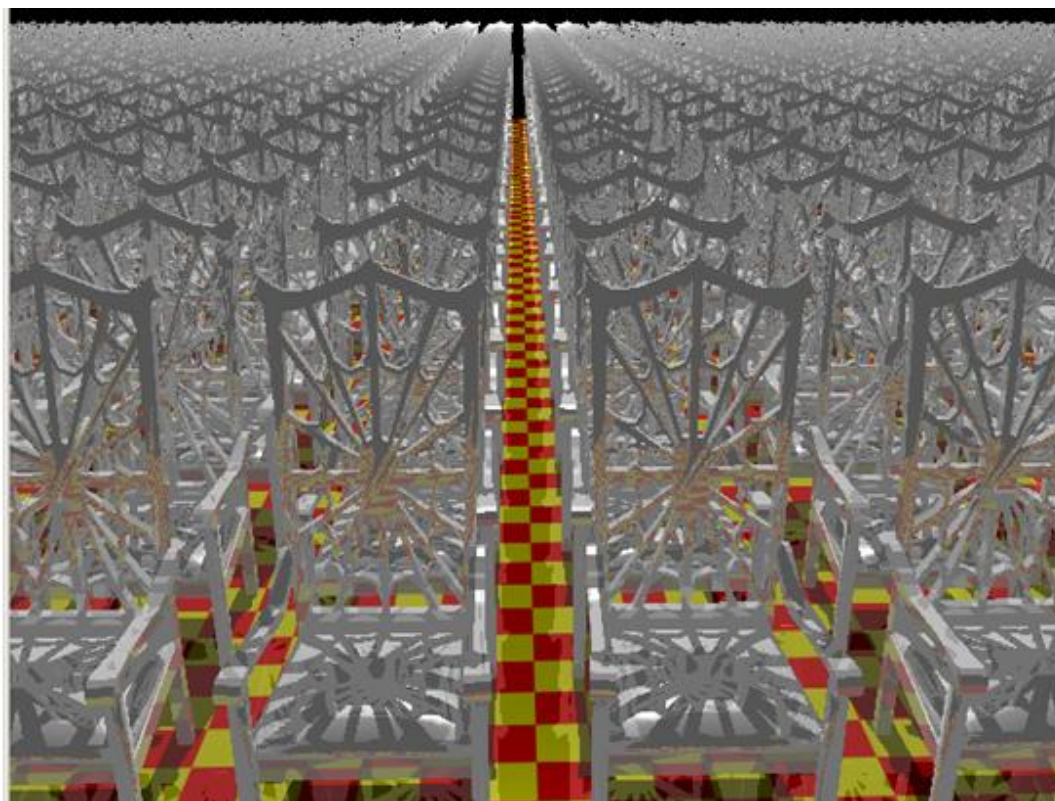
- ⌘ Conference Room (281K треугольников)
- ⌘ ~40М лучей в секунду на GTX260





# Производительность

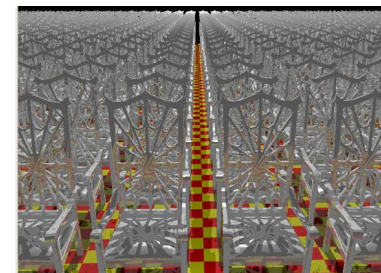
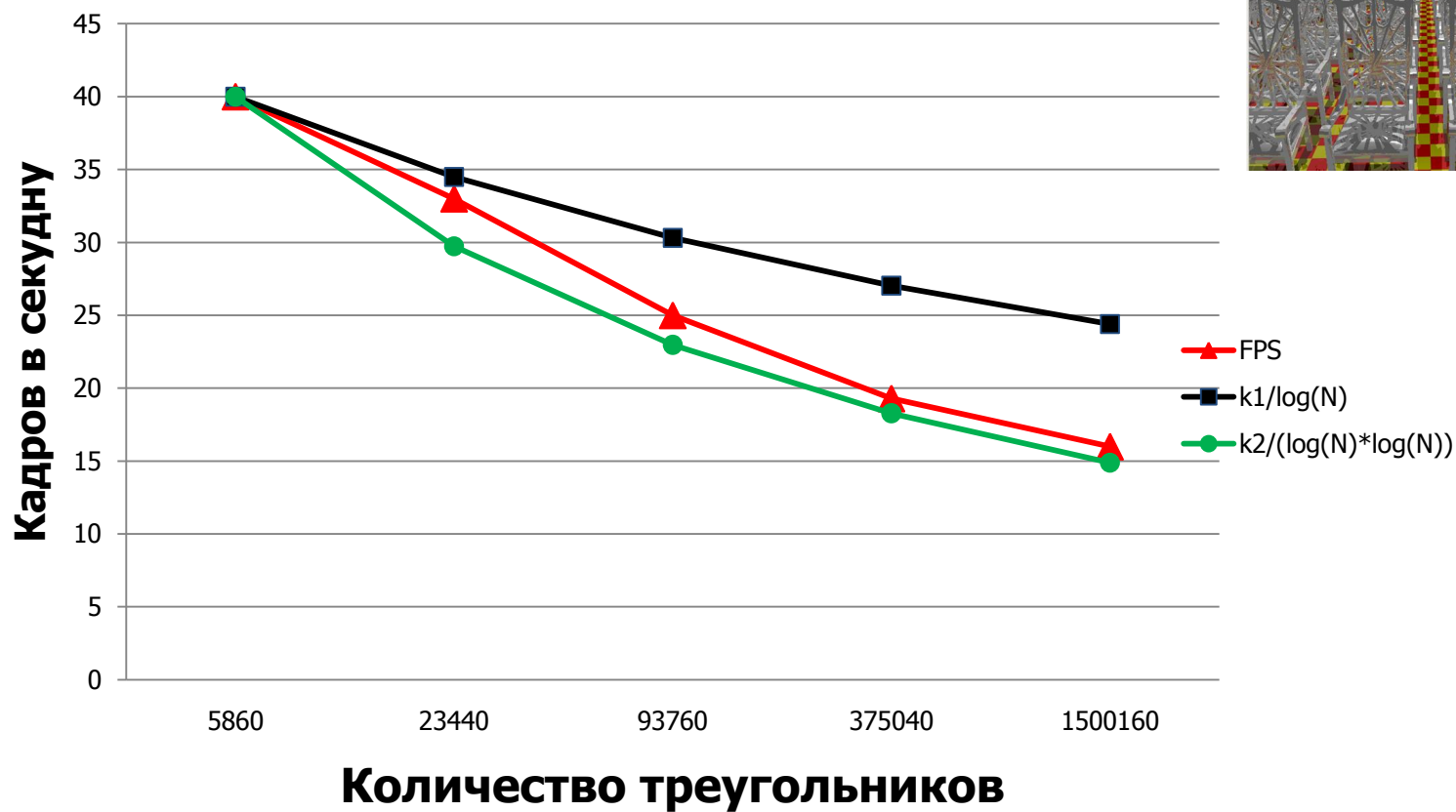
⌘ Стулья (1.5М треугольников)





# Производительность

1024x1024, GTX260





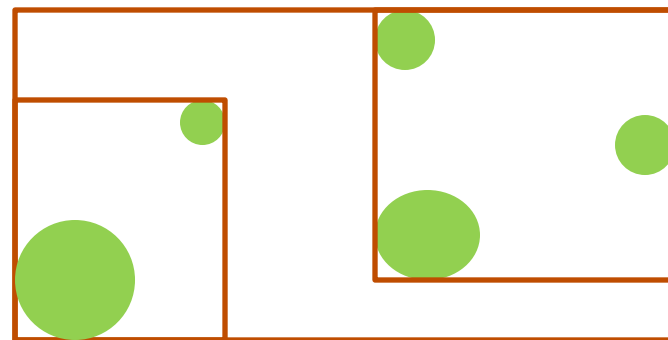
# kd-tree vs BVH на CUDA

⌘ BVH со стеком на локальной памяти

- ⏏ Покрывается латентность текстурной памяти
- ⏏ Меньше глубина
- ⏏ Лишние плоскости

⌘ kd-tree

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





# Резюме

## ⌘ OpenGL interop

- ☑️ текстуры, PBO, VBO, FBO

## ⌘ Occupancy

- ☑️ Оптимизации по локальным переменным

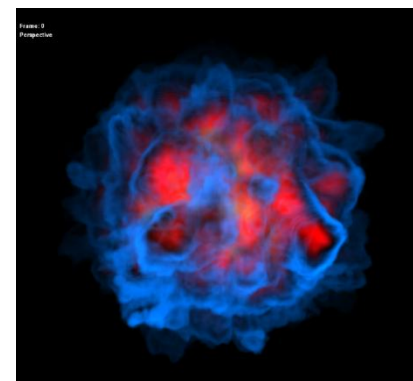
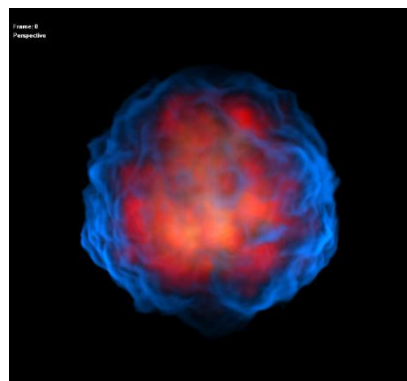
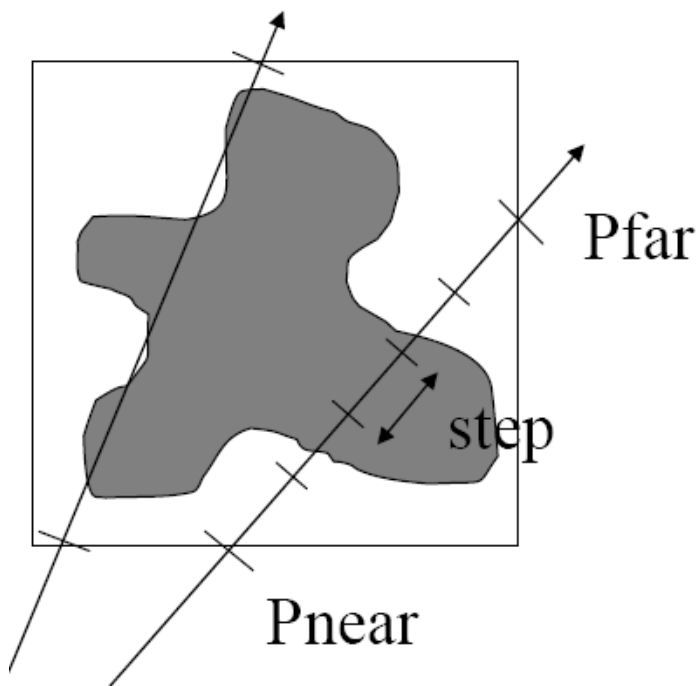
- ☑️ Uber kernel

- ☑️ Separate kernel

## ⌘ Persistent threads

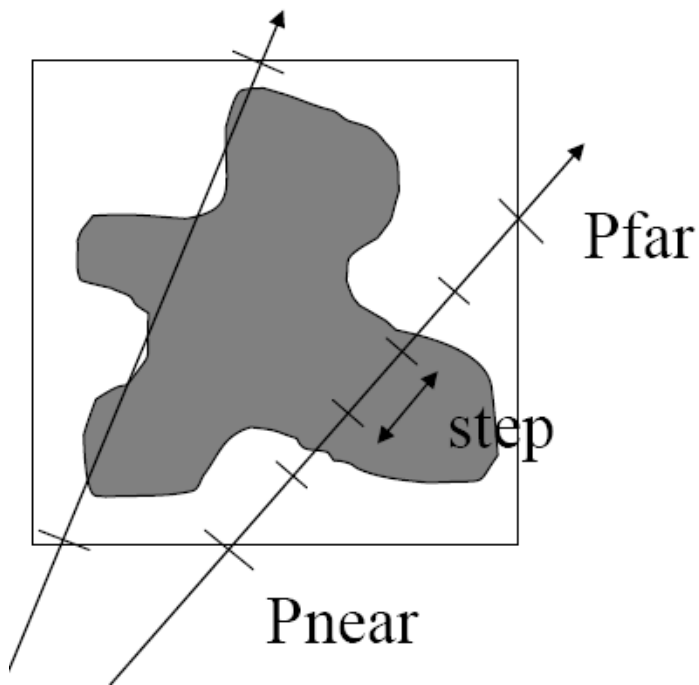


# Ray marching





# Ray marching



```
float4 RayMarchPS(Ray eyeray : TEXCOORD0,
                  uniform int steps) : COLOR
{
    eyeray.d = normalize(eyeray.d);

    // calculate ray intersection with bounding box
    float tnear, tfar;
    bool hit = IntersectBox(eyeray, boxMin, boxMax, tnear, tfar);
    if (!hit) discard;
    if (tnear < 0.0) tnear = 0.0;

    // calculate intersection points
    float3 Pnear = eyeray.o + eyeray.d*tnear;
    float3 Pfar = eyeray.o + eyeray.d*tfar;

    // march along ray, accumulating color
    half4 c = 0;
    half3 step = (Pnear - Pfar) / (steps-1);
    half3 P = Pfar;
    for(int i=0; i<steps; i++) {
        half4 s = VOLUMEFUNC(P);
        c = s.a*s + (1.0-s.a)*c;
        P += step;
    }
    c /= steps;
    return c;
}
```



