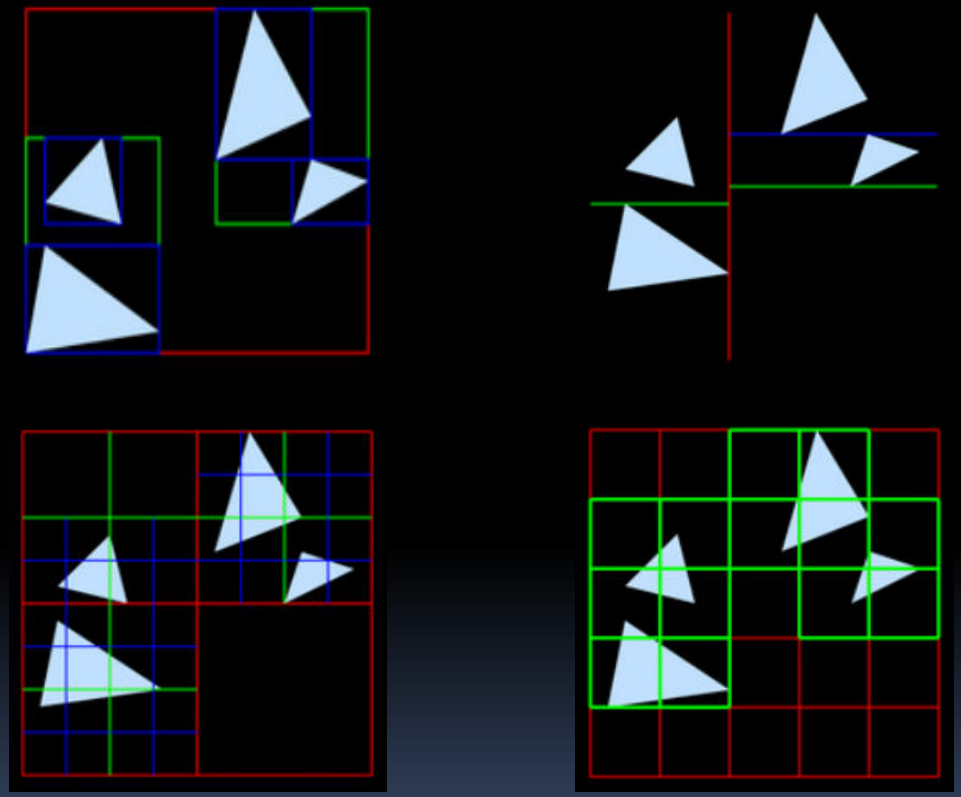
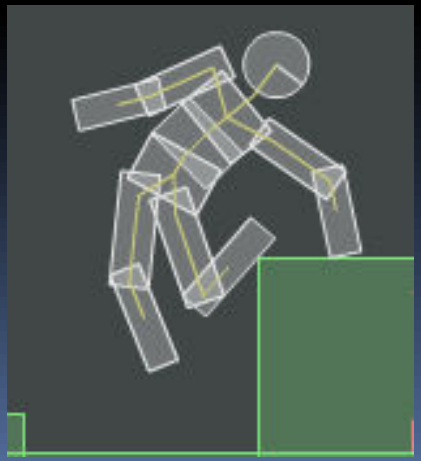
Структуры пространственного разбиения



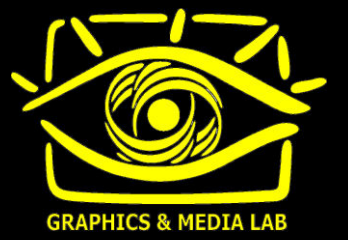
Фролов В.А.
vfrolov@graphics.cs.msu.ru

Области применения

GRAPHICS & MEDIA LAB

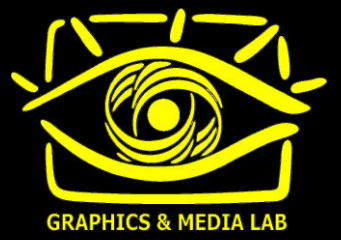


Виды структур пространственного разбиения



- BSP (Binary Space Partion)
  - ✓ классичесий вариант
  - ✓ kd-tree
- BVH (Bounding Volume Hierarchy)
  - ✓ sphere-tree
  - ✓ AABB-tree (Axis Aligned Binding Box)
- BIH (Bounding Interval Hierarchy)
- Regular grid
- Oc-tree
- Z-ordering (UB-tree)

BSP (классический вариант)



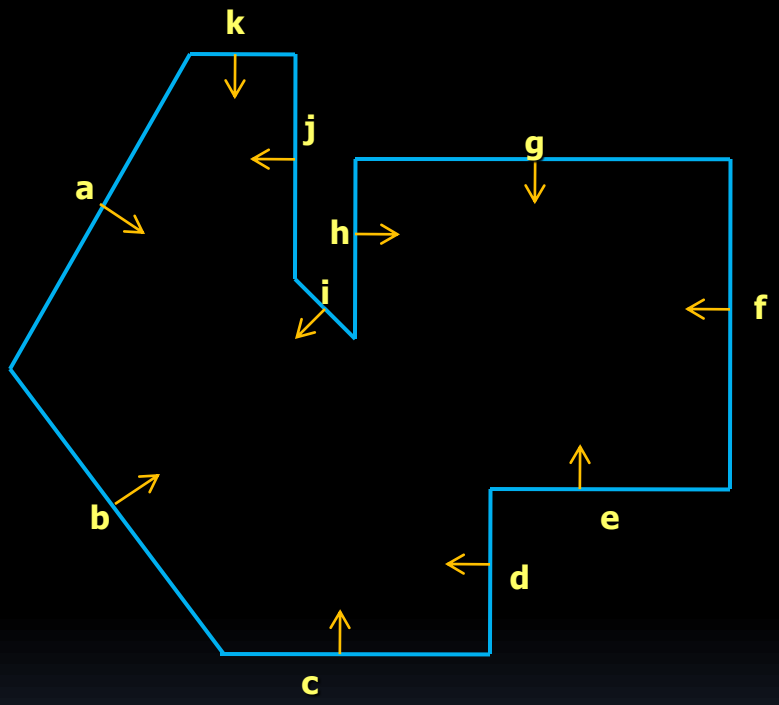
Основные моменты:

- Бинарное дерево
- Плоскости разбиения проходят по полигонам
- Обычно вместо самой плоскости в узле хранят указатель на разбивающий полигон
- Зная позицию камеры, можно определить ближнее и дальнее поддеревья

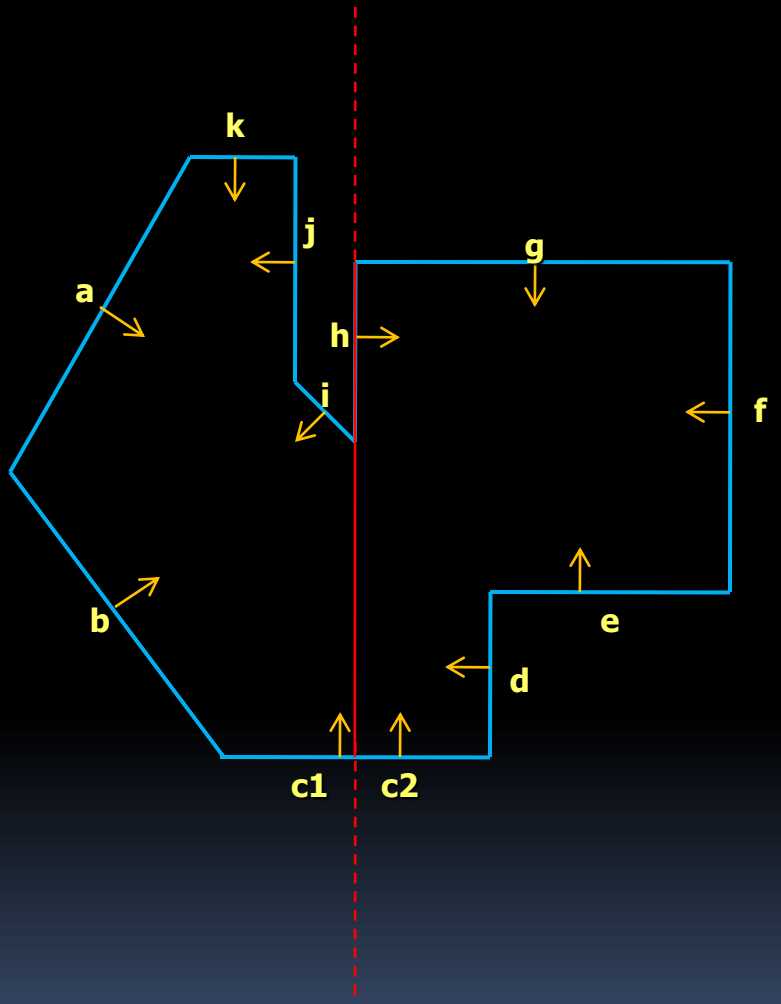
```
struct BSPNode
{
  int poly_index;
  int plane_index;
  int front_index;
  int back_index;
};
```

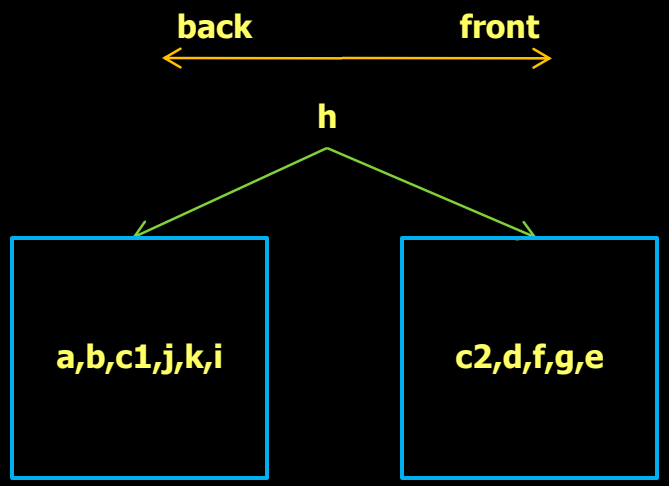
```
struct BSPNode
{
  int poly_index;
  //int plane_index; // = poly_index
  int front_index;
  //int back_index; // = front_index + 1
};
```

BSP (классический вариант)

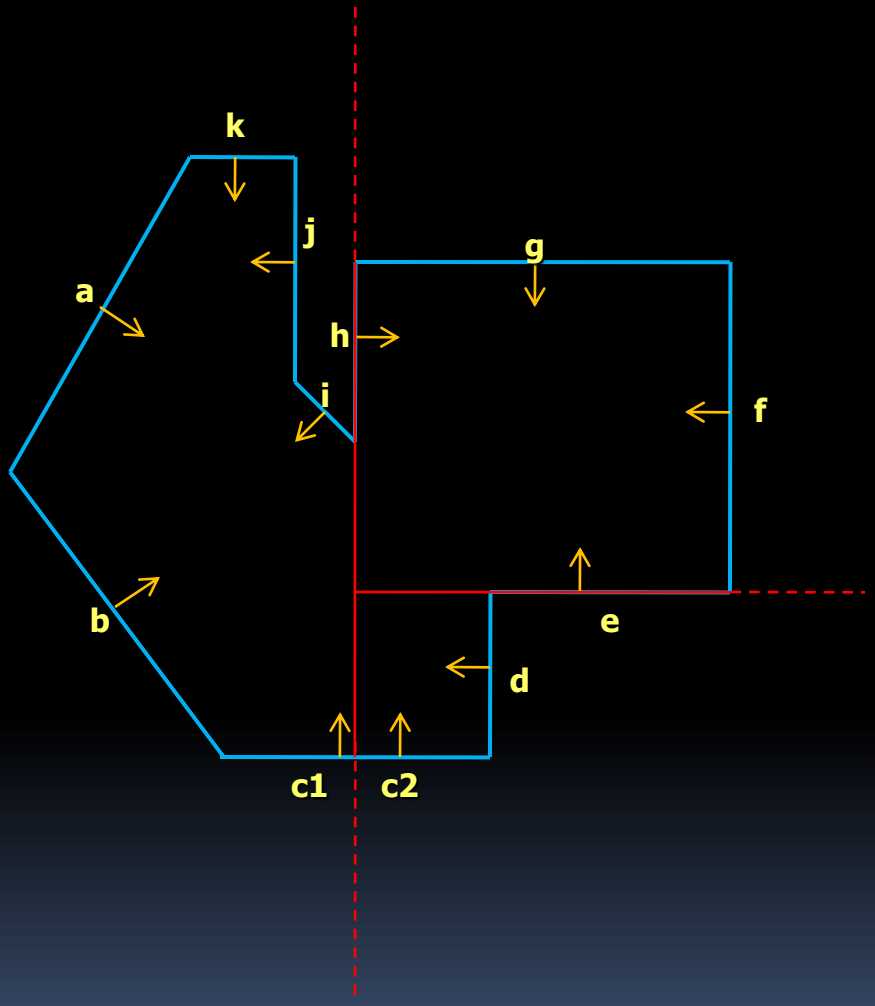


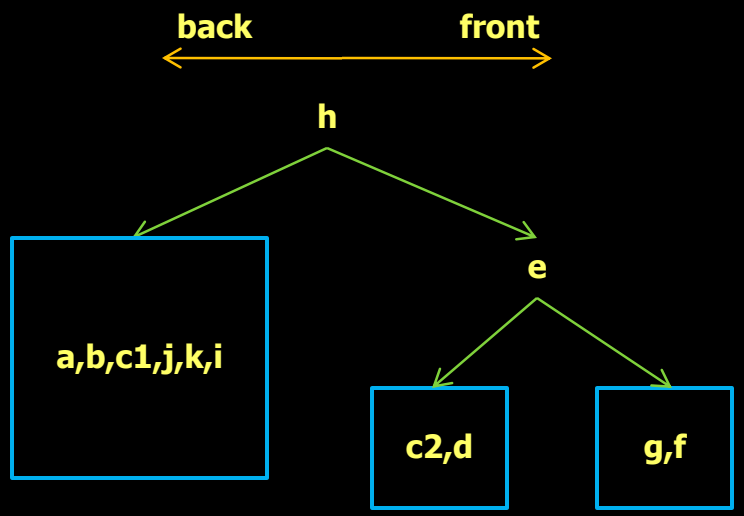
BSP (классический вариант)



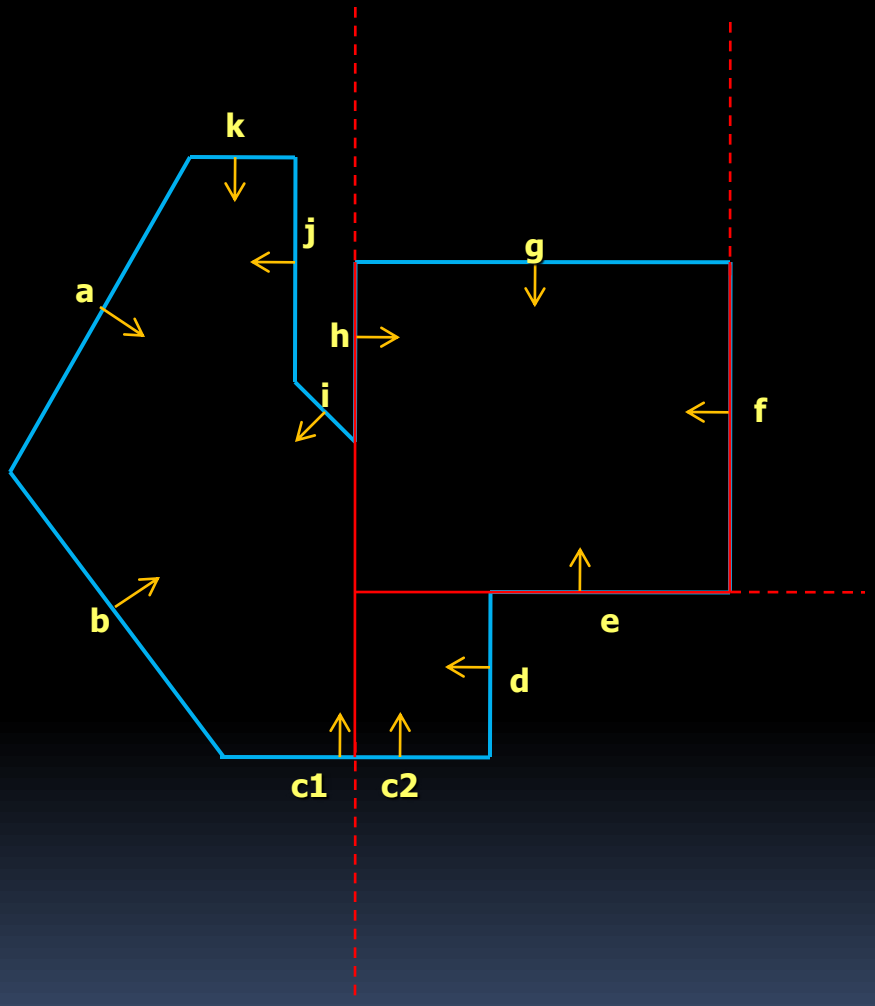


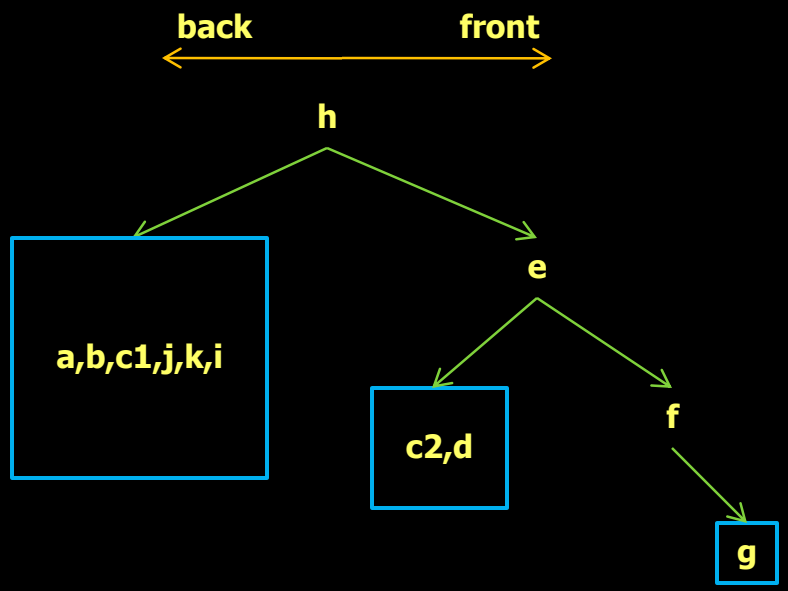
BSP (классический вариант)



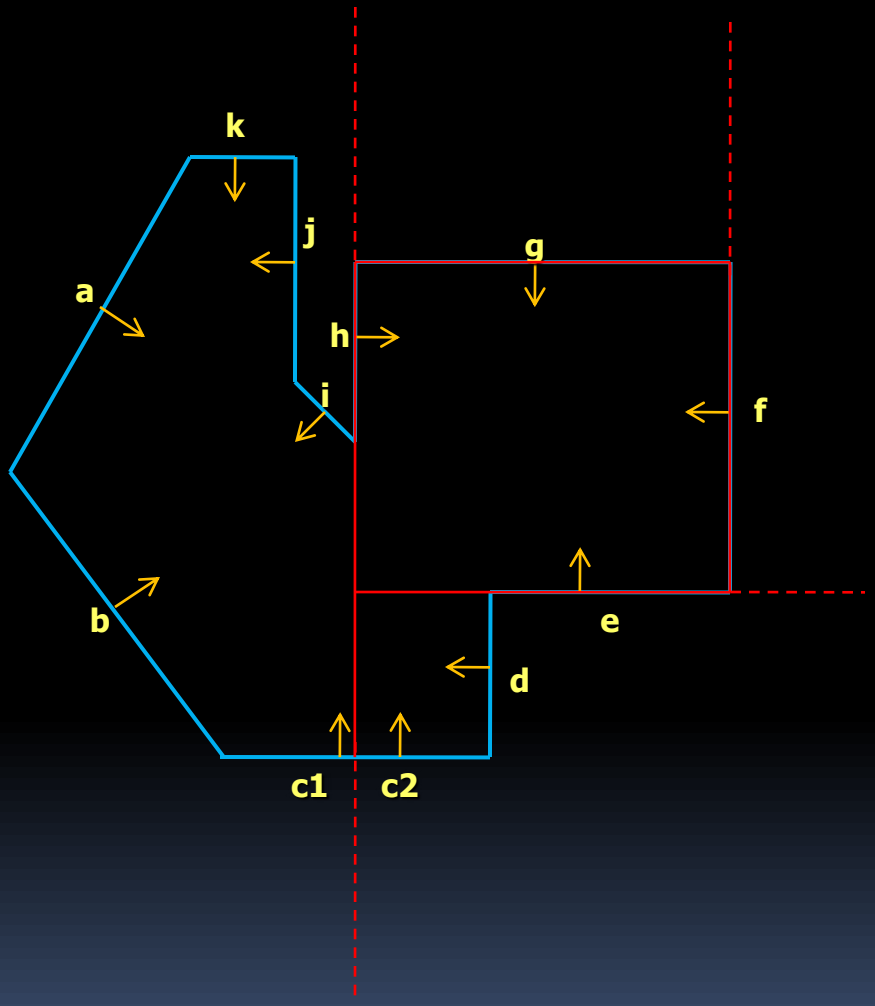


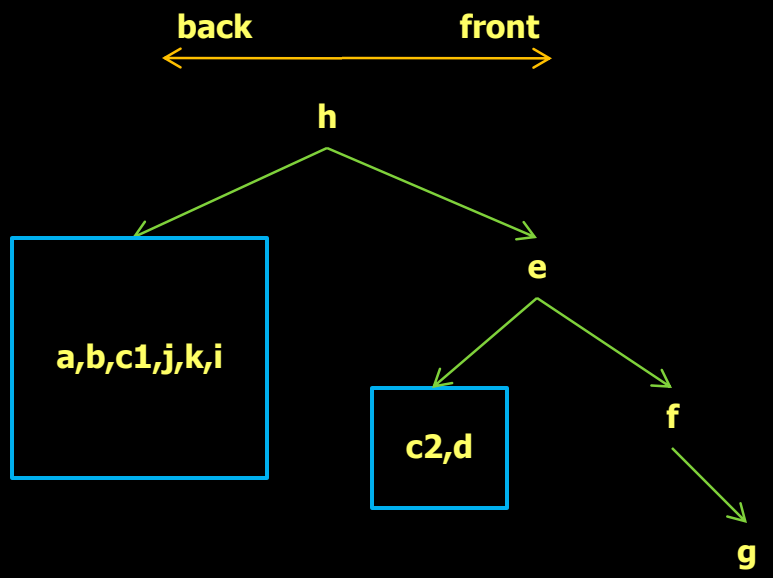
BSP (классический вариант)



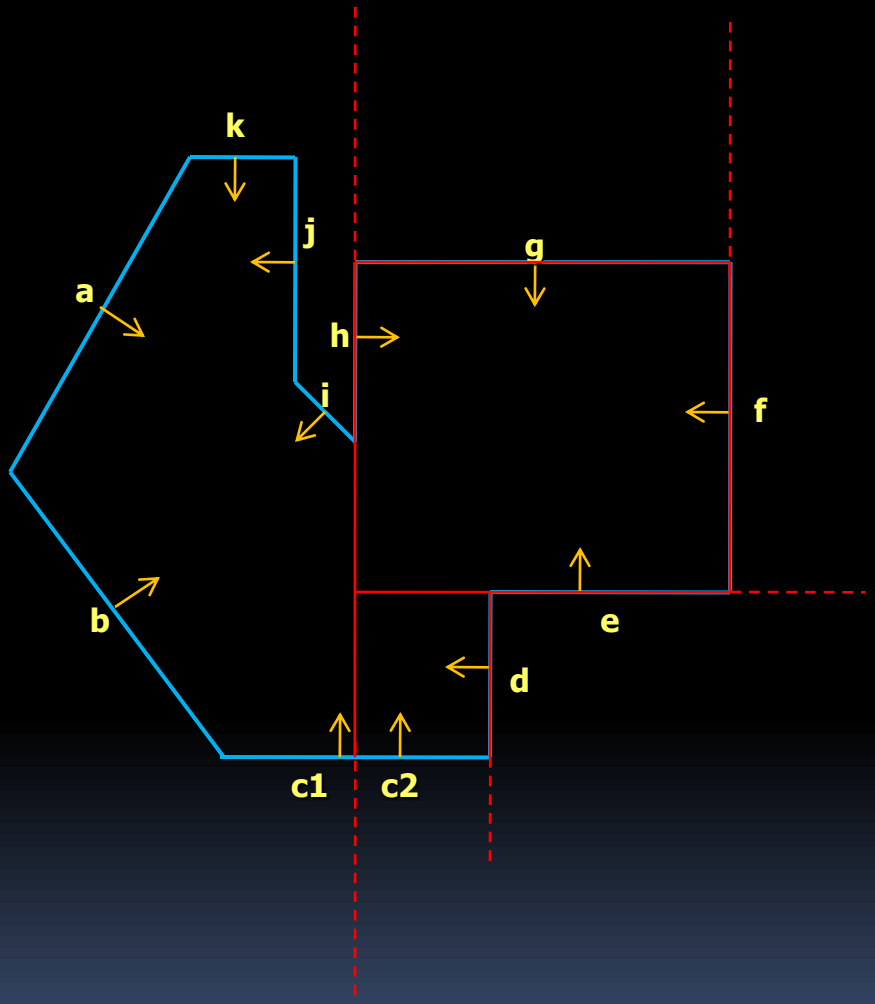


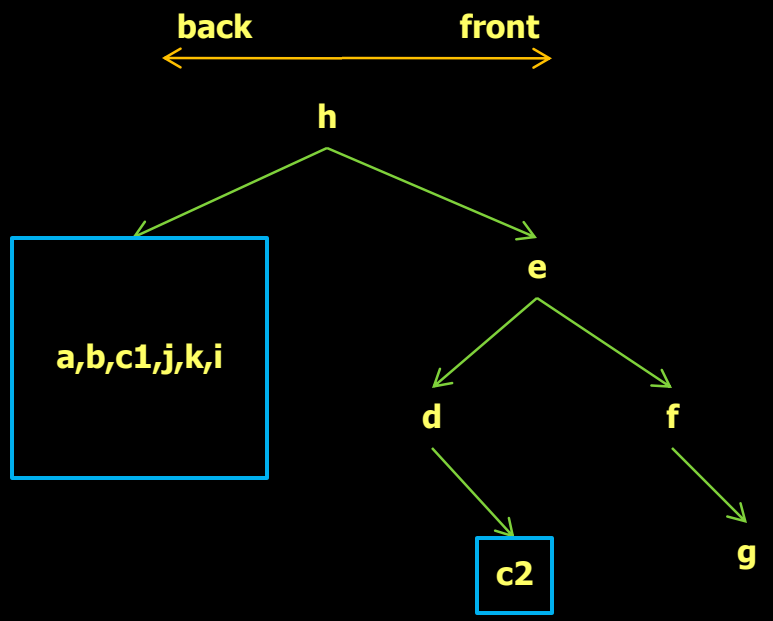
BSP (классический вариант)



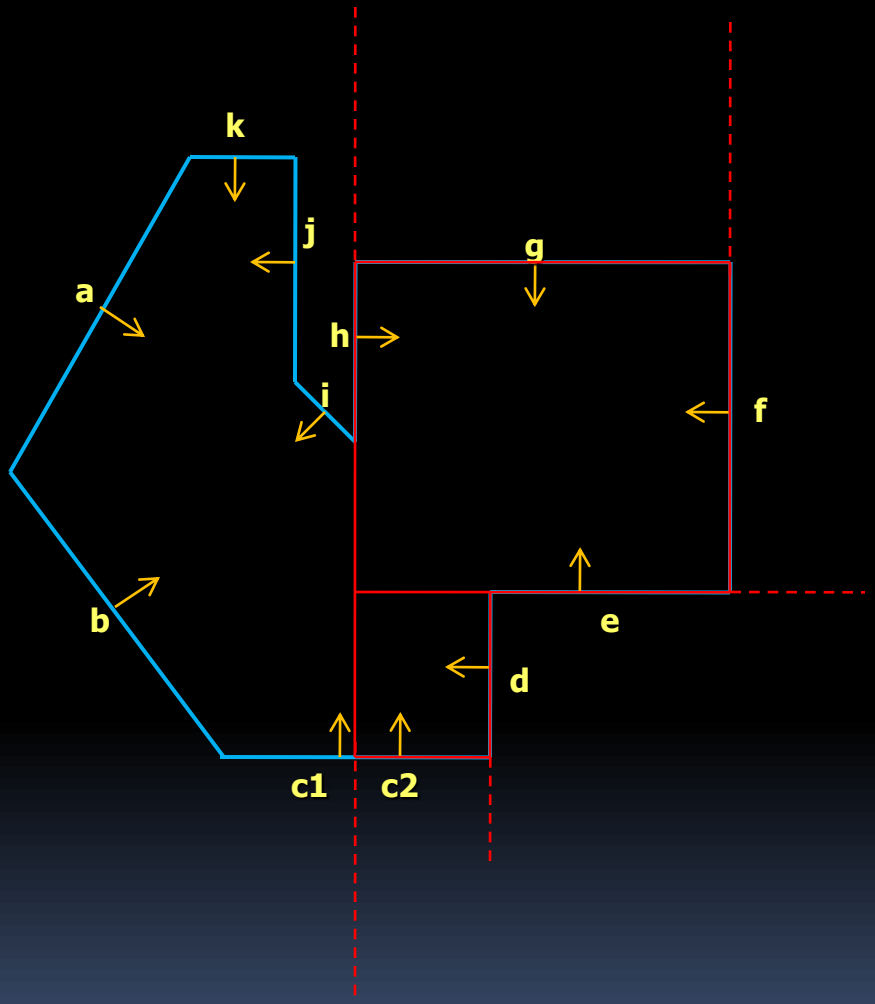


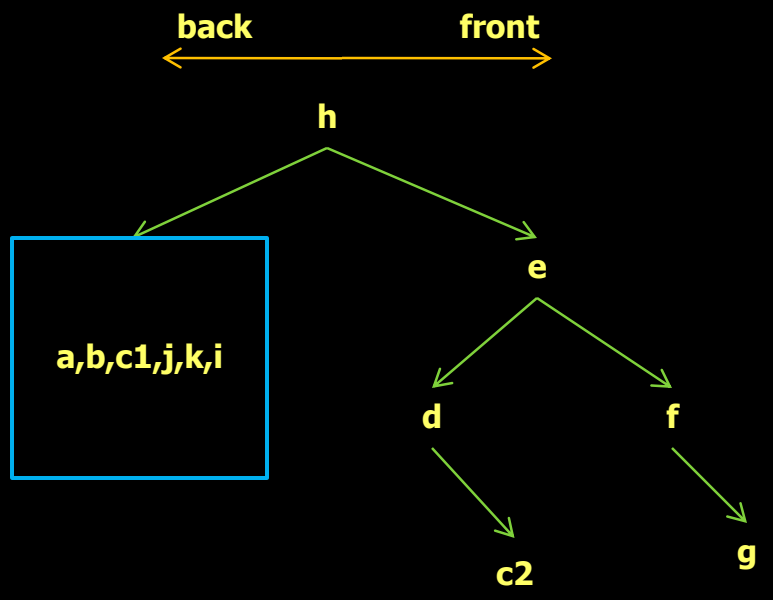
BSP (классический вариант)



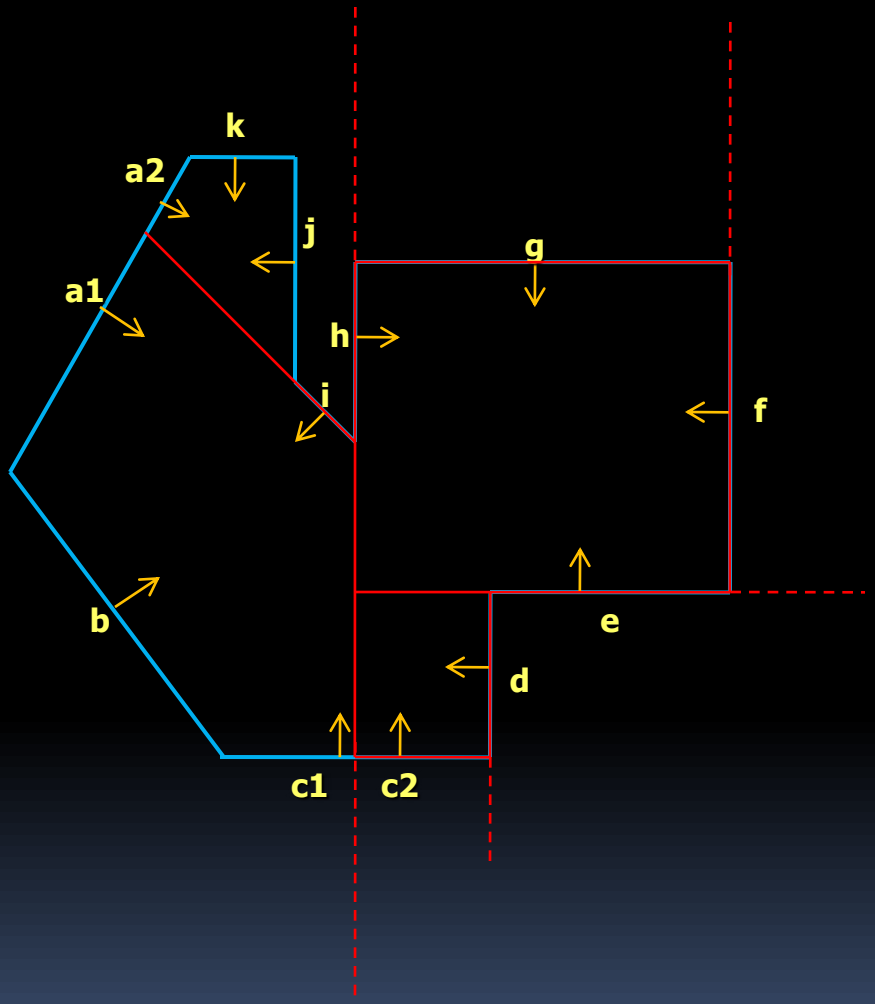


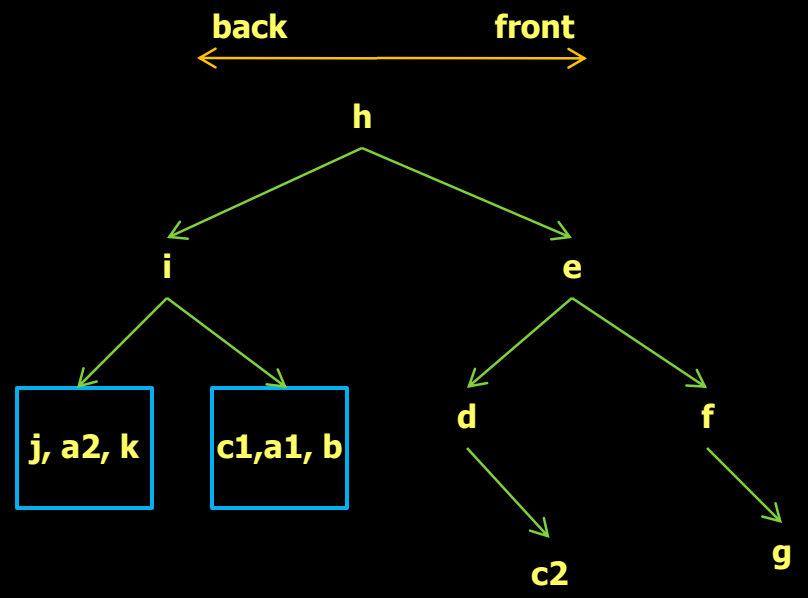
BSP (классический вариант)



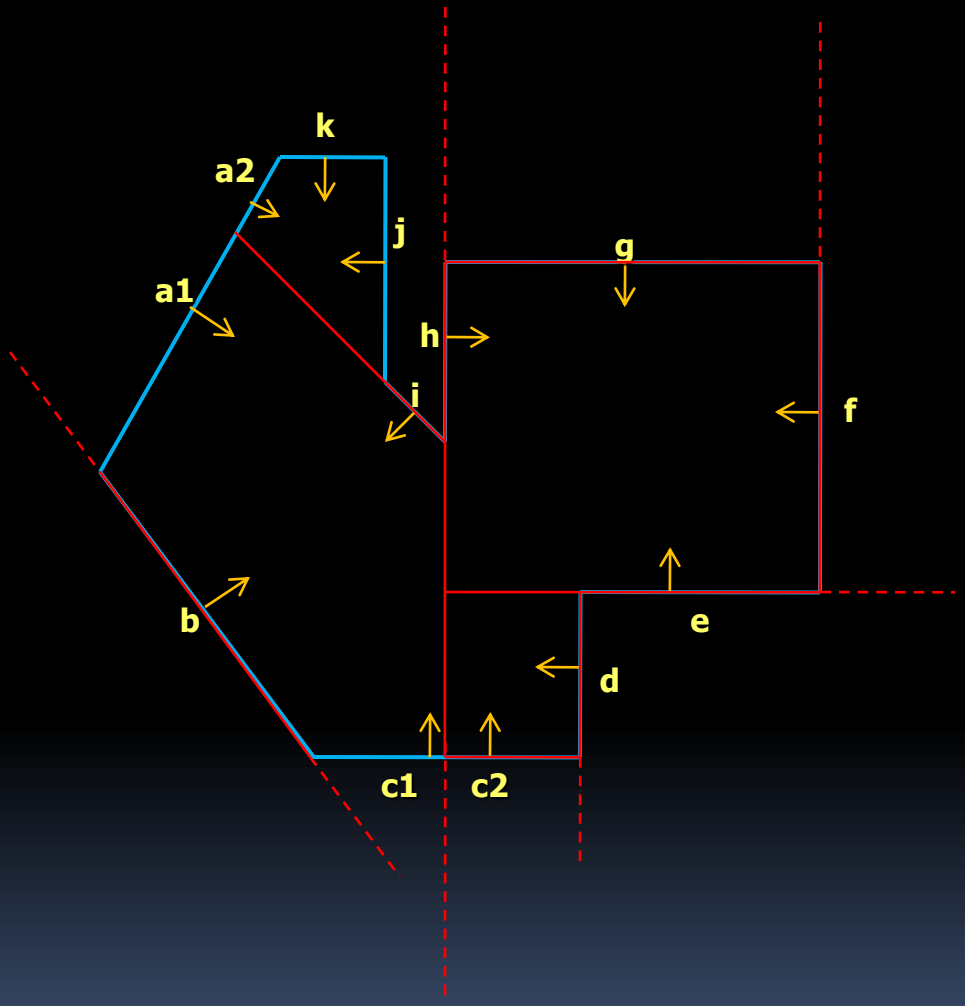


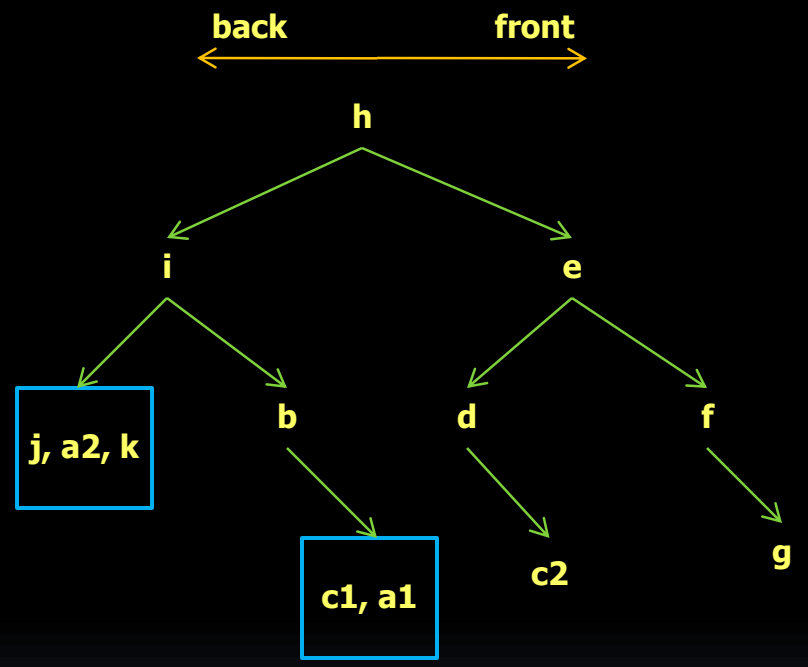
BSP (классический вариант)



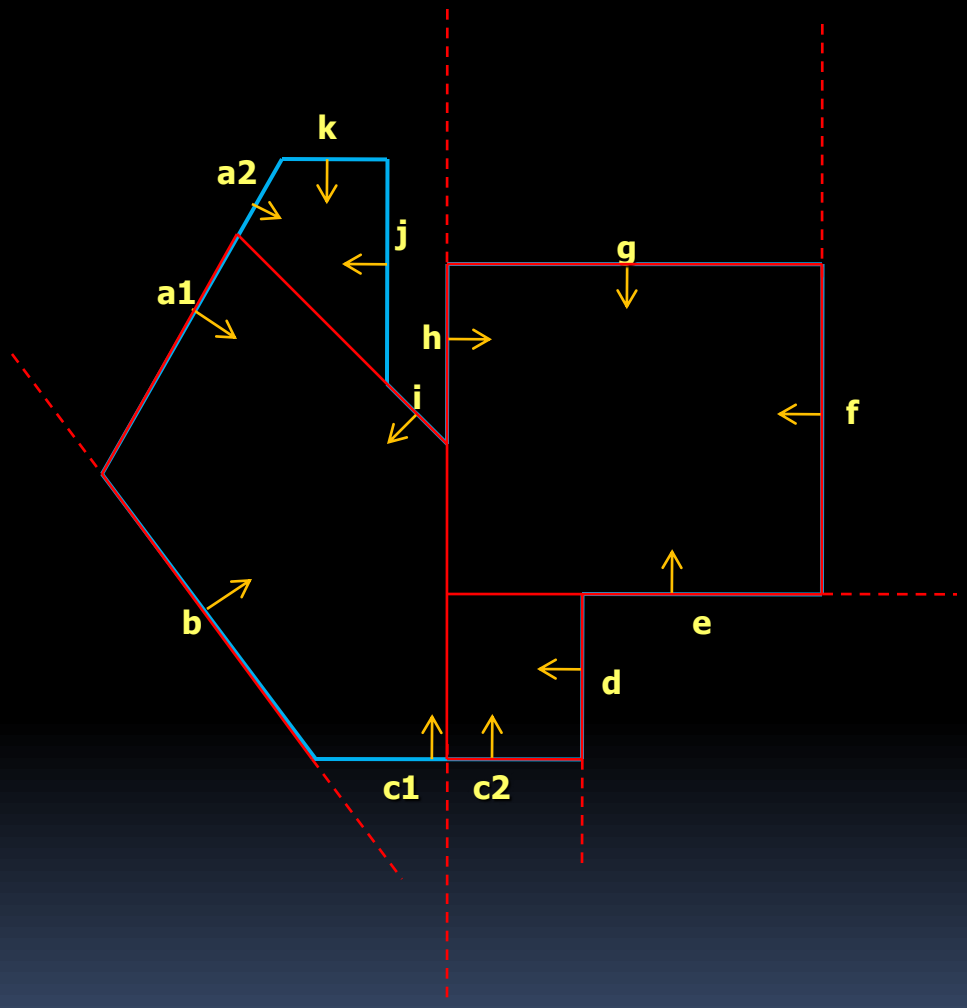


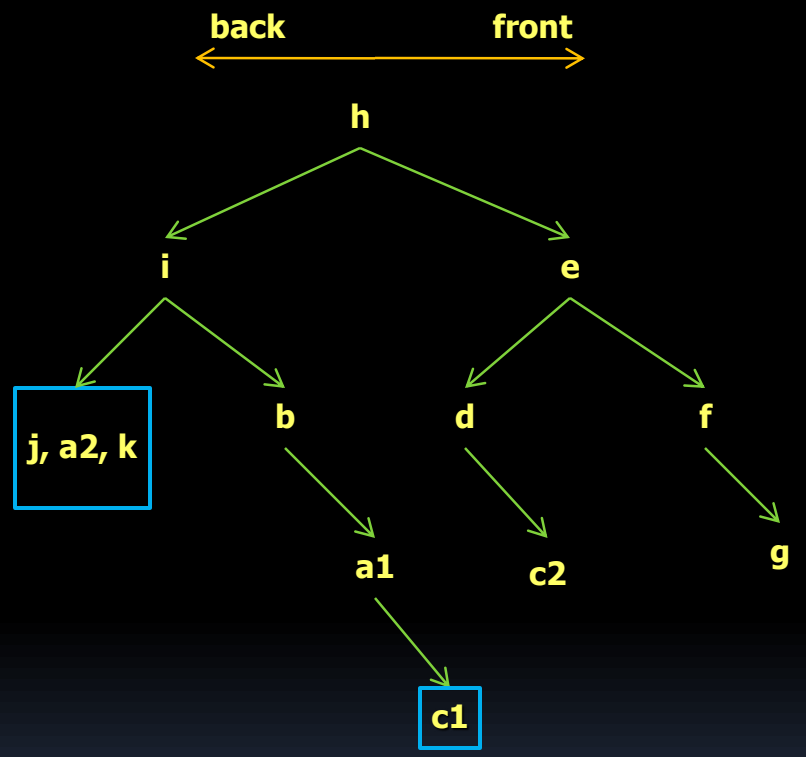
BSP (классический вариант)



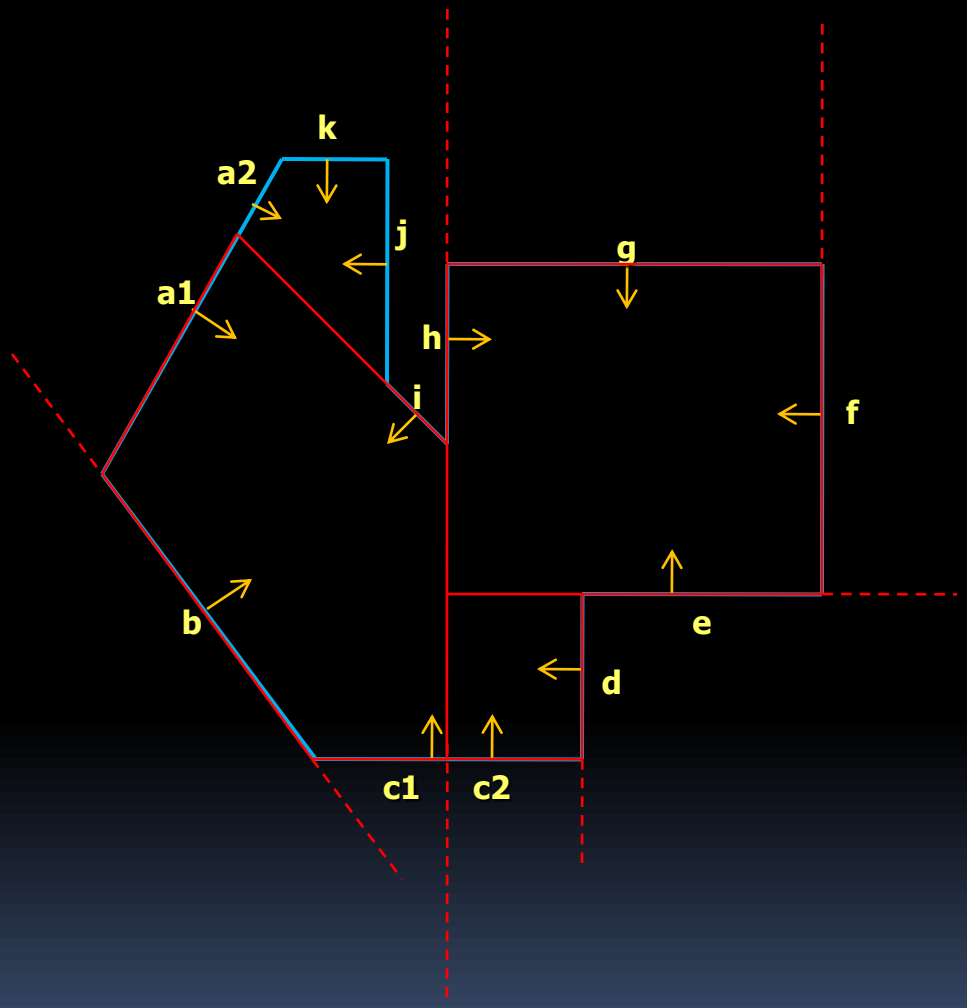


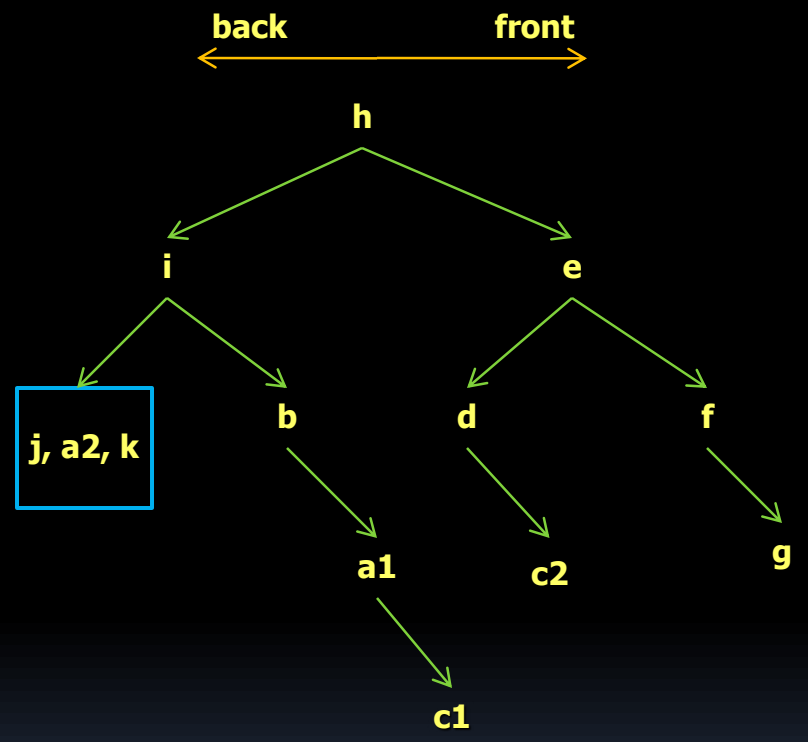
BSP (классический вариант)





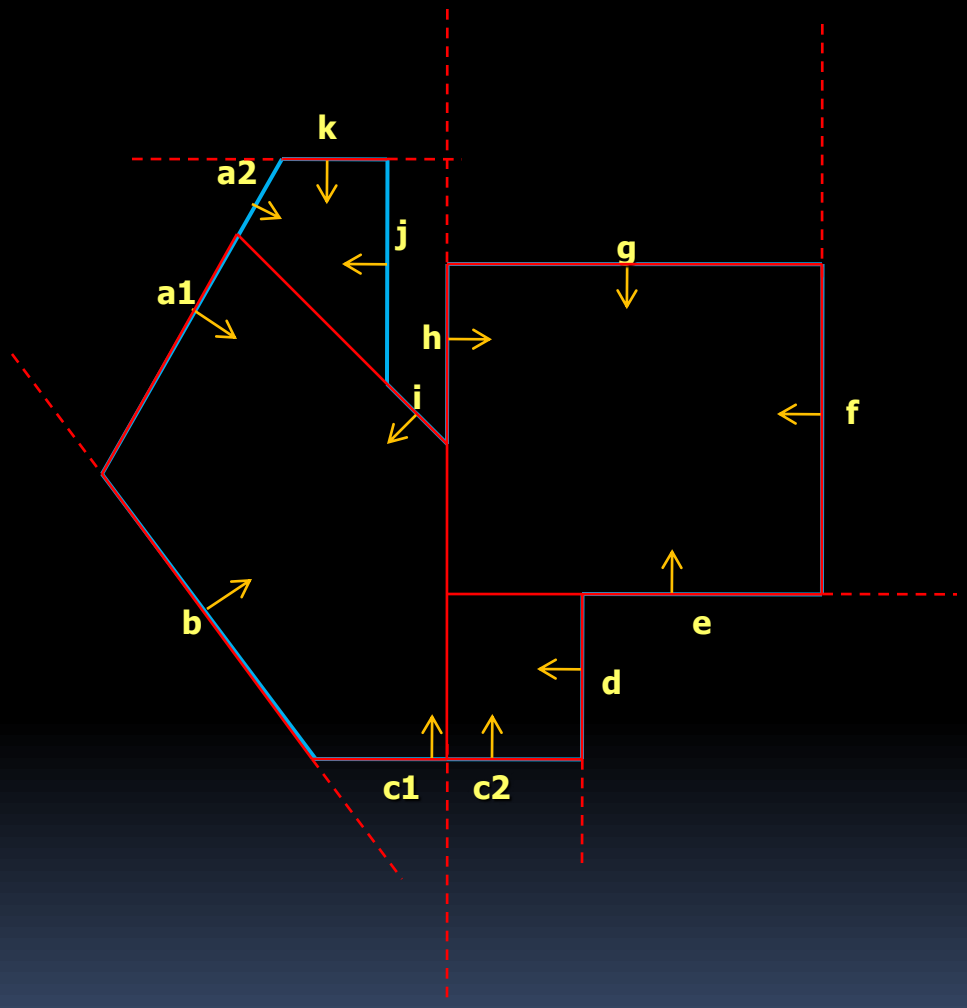
BSP (классический вариант)

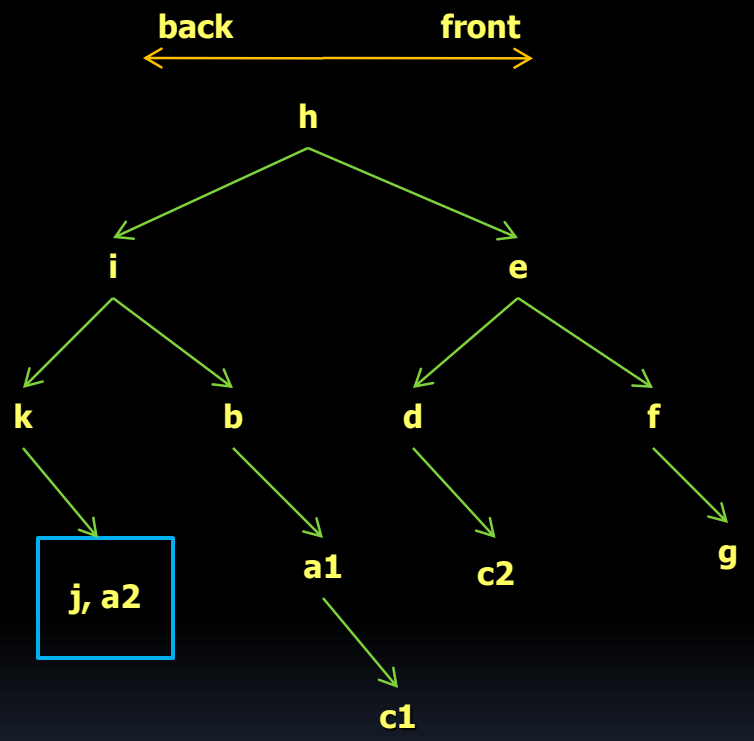




BSP (классический вариант)

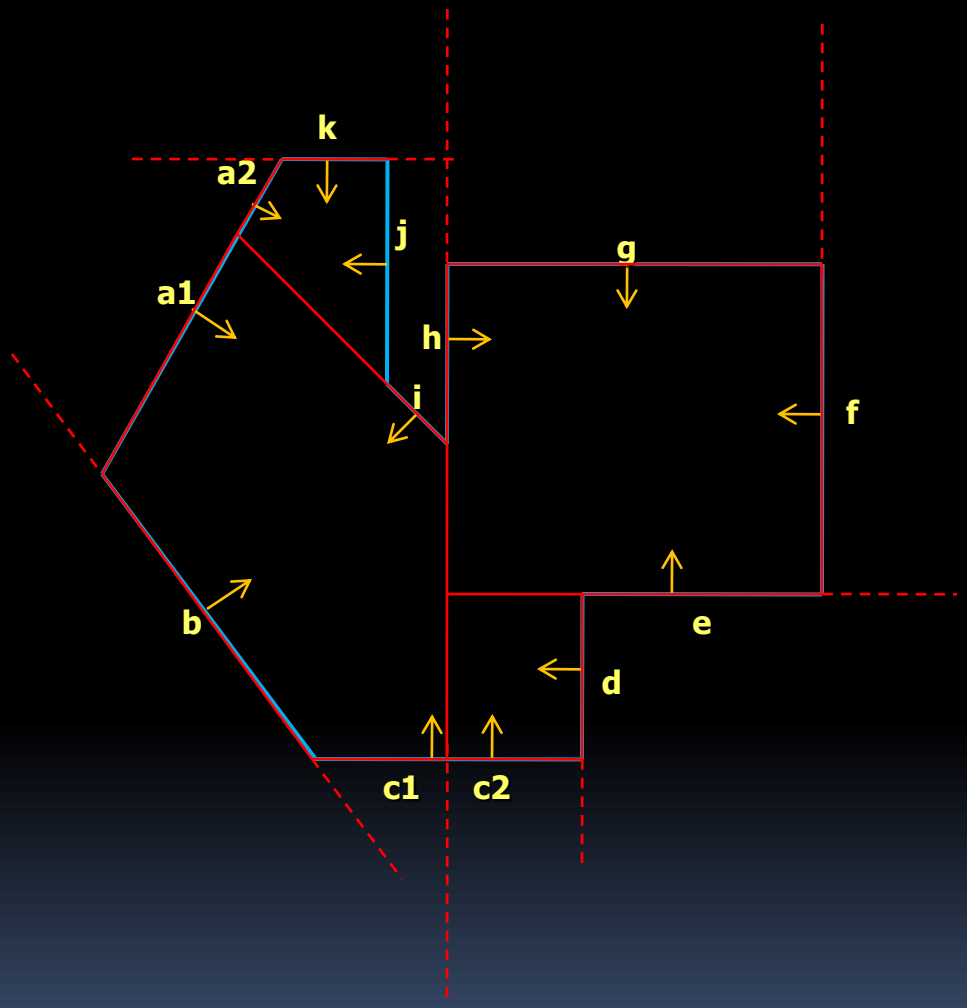
GRAPHICS & MEDIA LAB

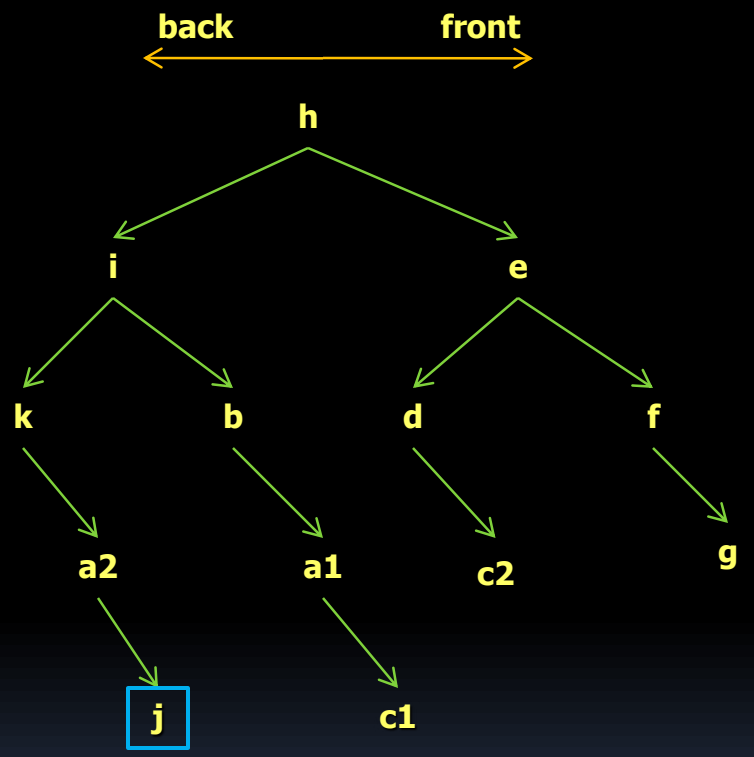




BSP (классический вариант)

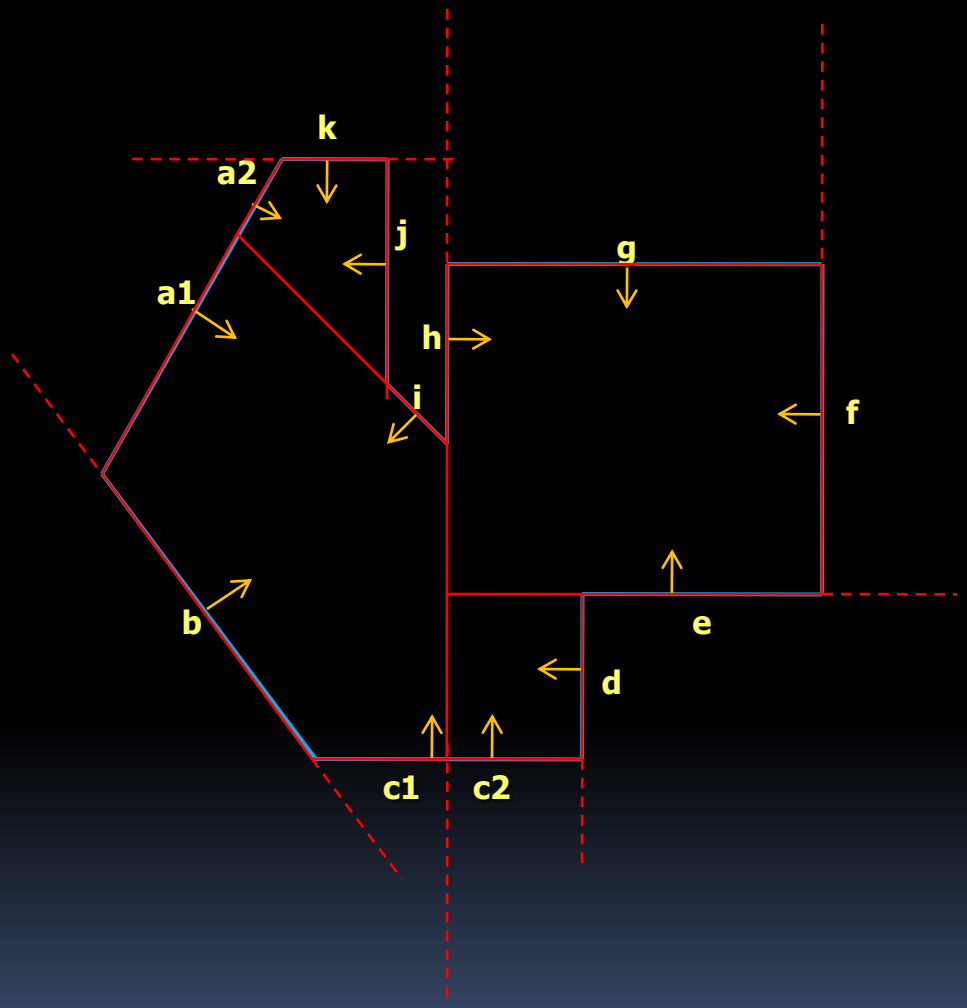
GRAPHICS & MEDIA LAB

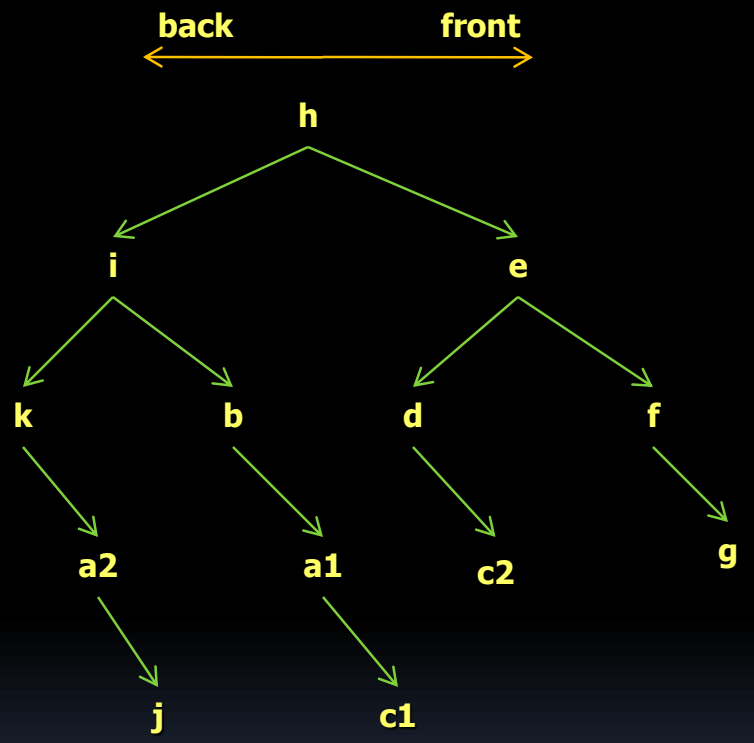




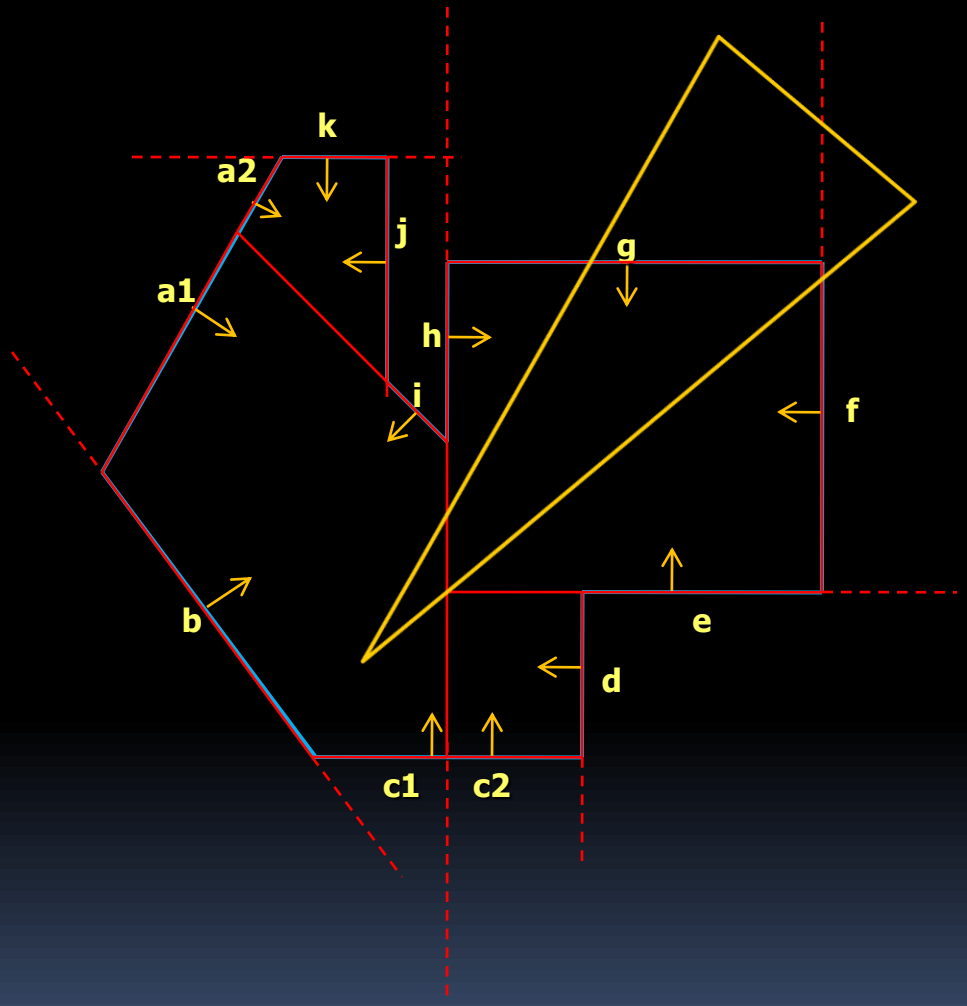
BSP (классический вариант)

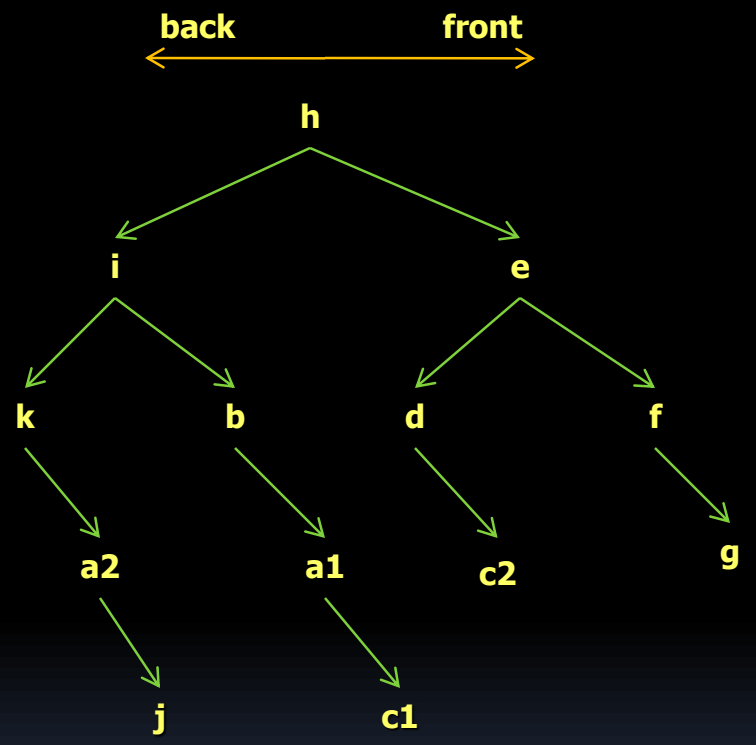
GRAPHICS & MEDIA LAB





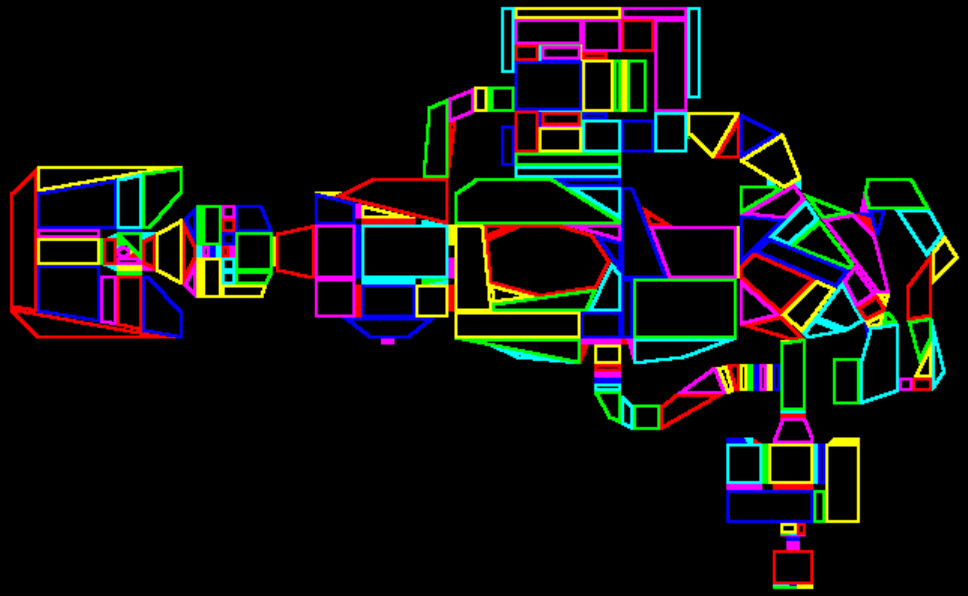
BSP (классический вариант)





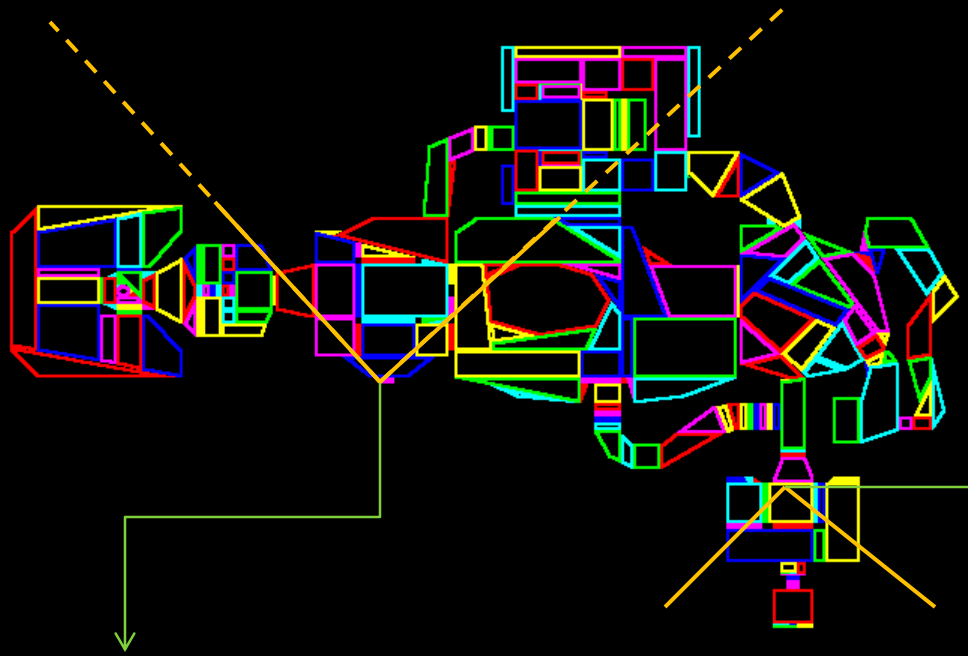
- Теперь траверсим дерево с учетом позиции камеры
- А как же отсечение по пирамиде видимости?

BSP (классический вариант)

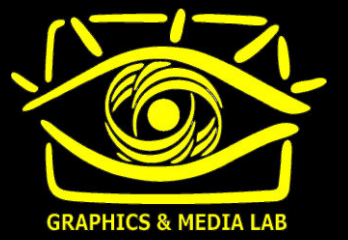


BSP (классический вариант)

GRAPHICS & MEDIA LAB

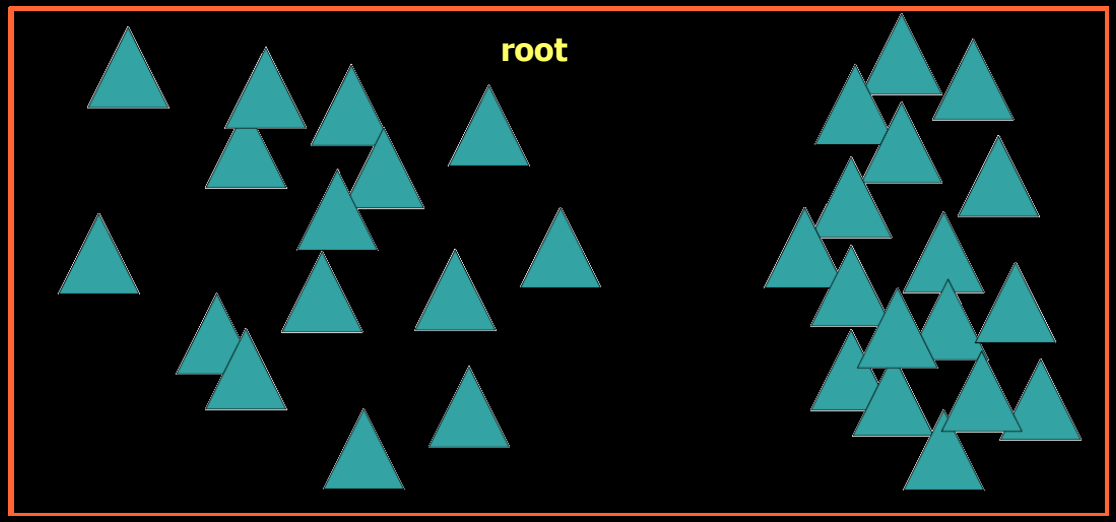


BSP (классический вариант)



- Преимущества
  - Растеризация: порядок обхода от дальнего к ближнему (прозрачность, рендеринг без Z-буффера)
  - Растеризация: удаление невидимых поверхностей
  - Растеризация: возможность хранить треугольники прямо в узлах BSP.
- Недостатки
  - сложная форма узлов – использование дополнительных ограничивающих объемов

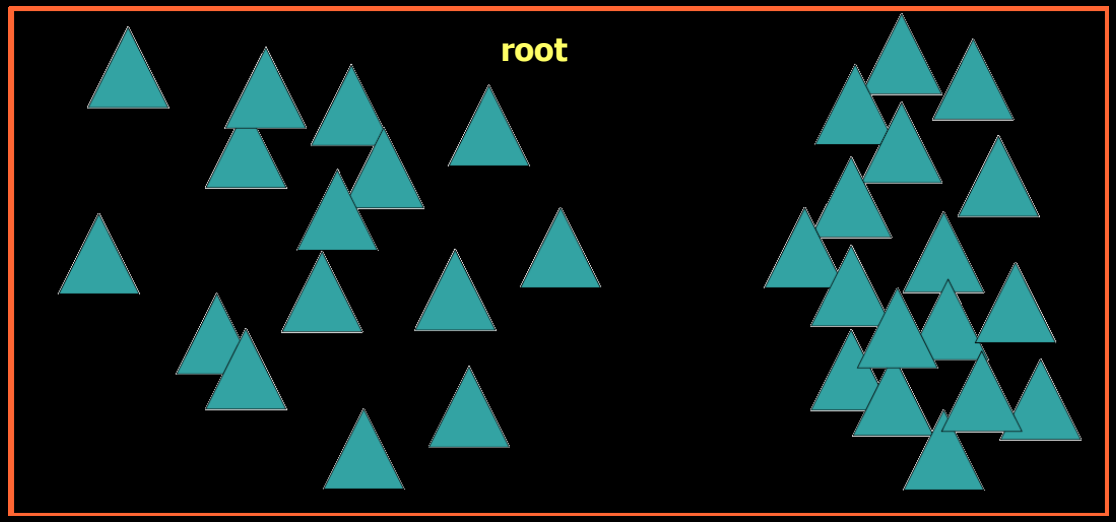
kd-tree



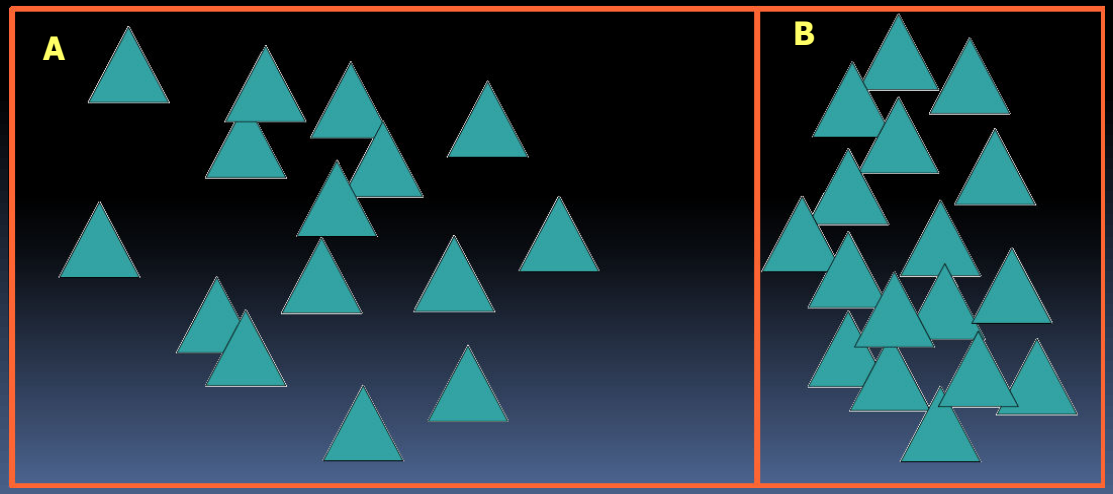
root

kd-tree

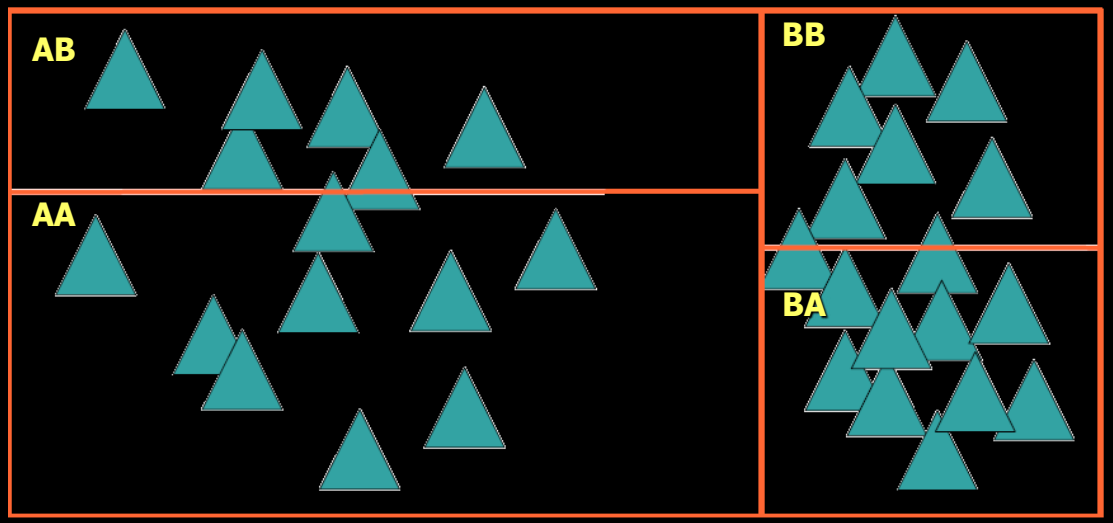
GRAPHICS & MEDIA LAB

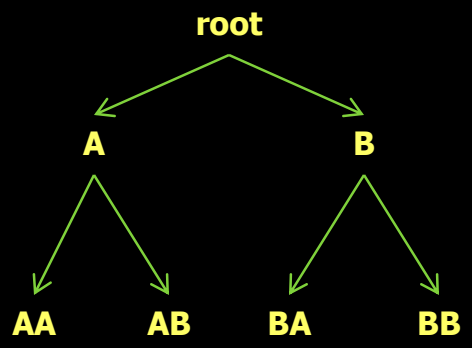


root

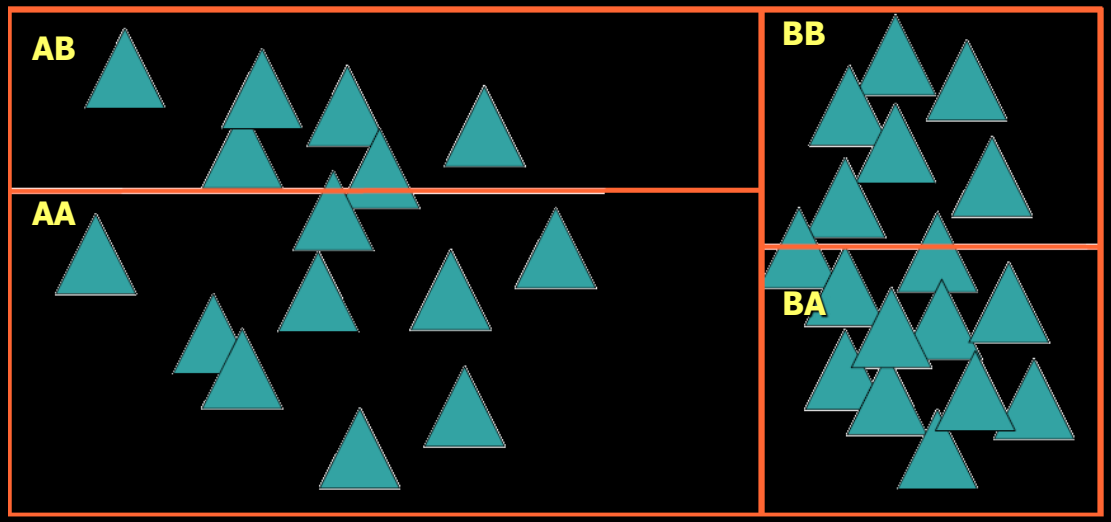


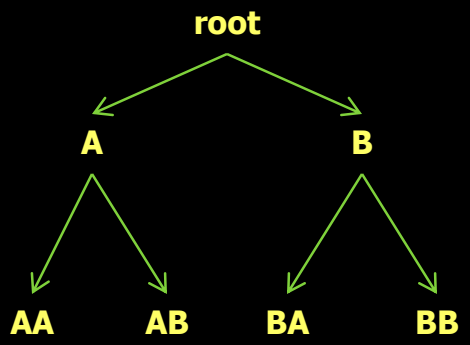
kd-tree

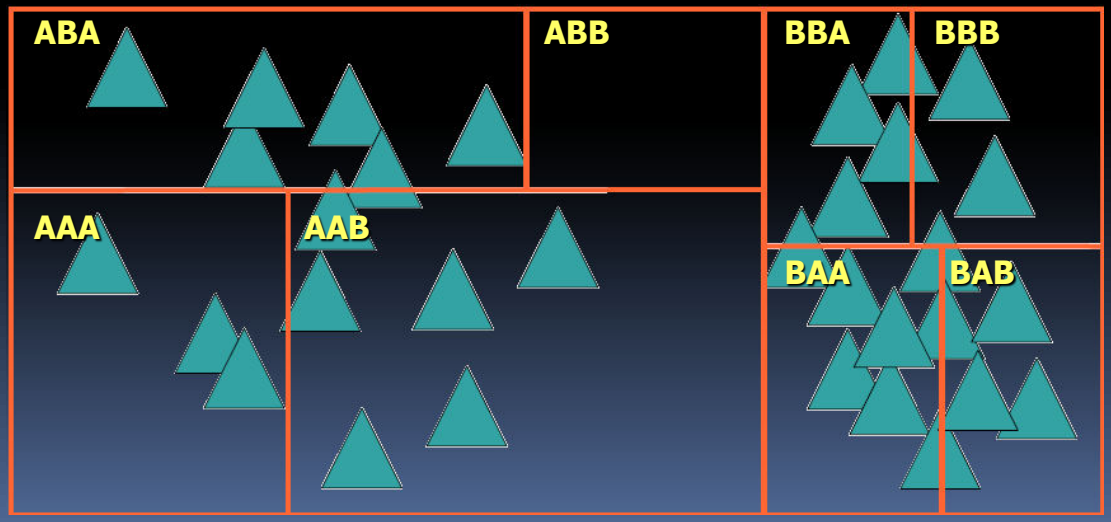


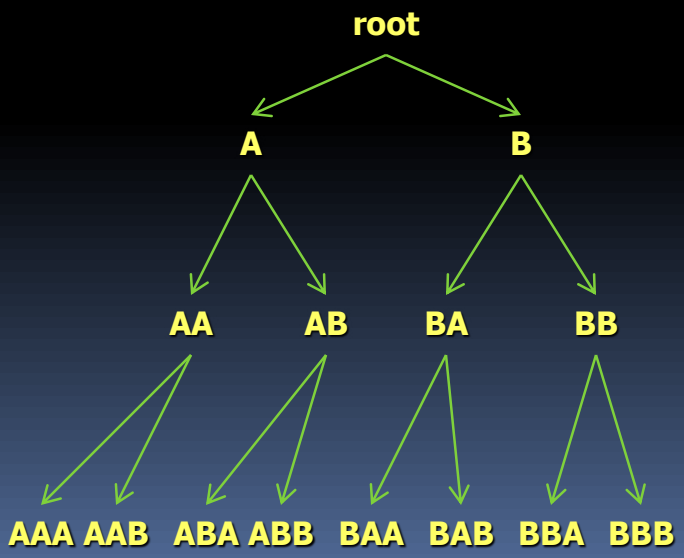


kd-tree


AB
BB
AA
BA


root
A
B
AA
AB
BA
BB


ABA
ABB
BBA
BBB
AAA
AAB
BAA
BAB

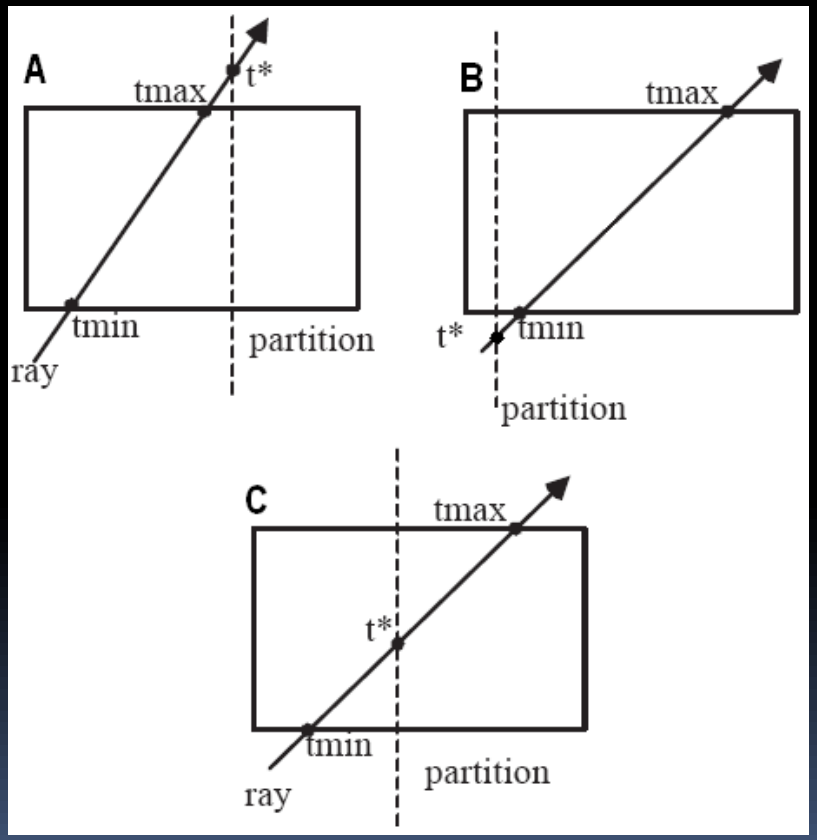

root
A
B
AA
AB
BA
BB
AAA
AAB
ABA
ABB
BAA
BAB
BBA
BBB

kd-tree traversal

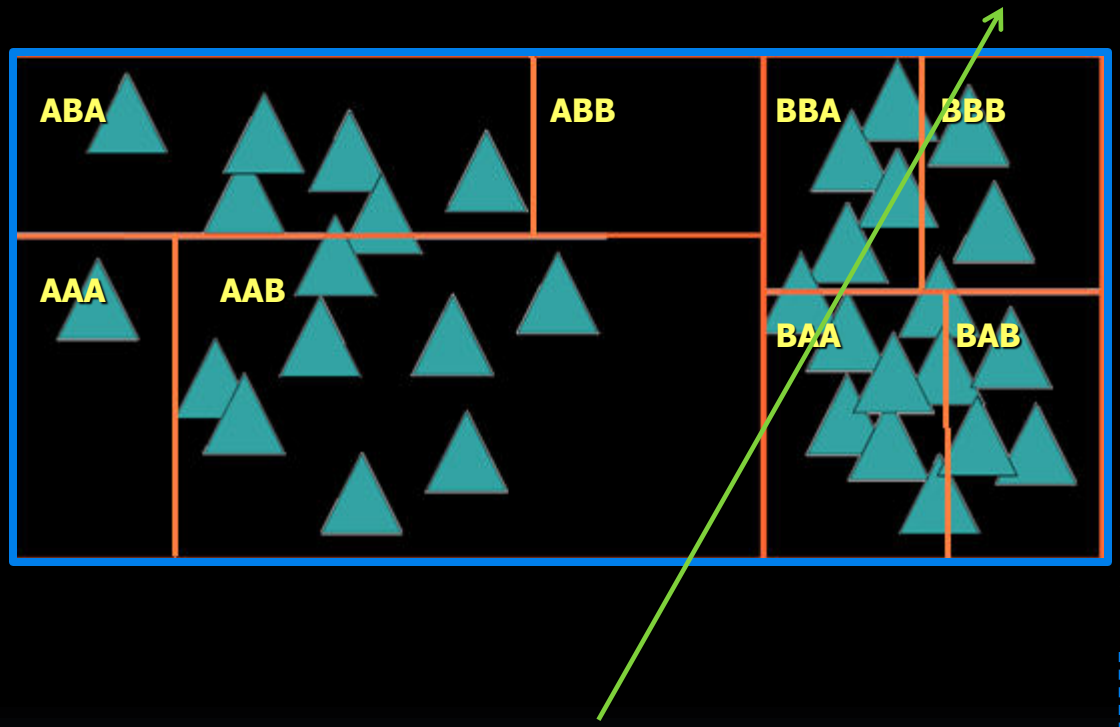
GRAPHICS & MEDIA LAB

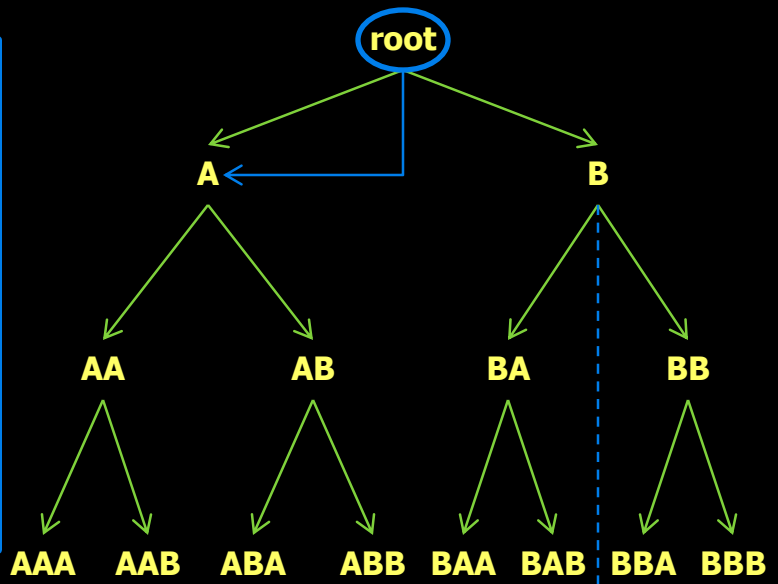
- Независимо от размерности пространства, всегда рассматриваются 3 случая

- На каждом шаге алгоритма прослеживания выполняется одно сложение и одно умножение



kd-tree traversal



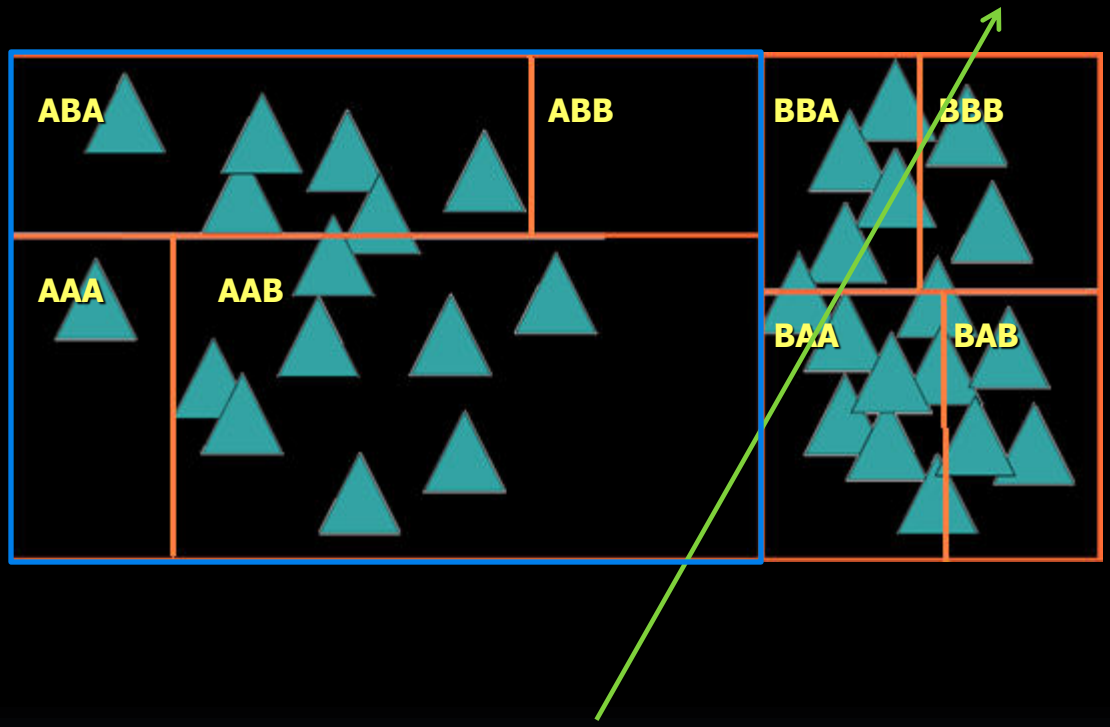


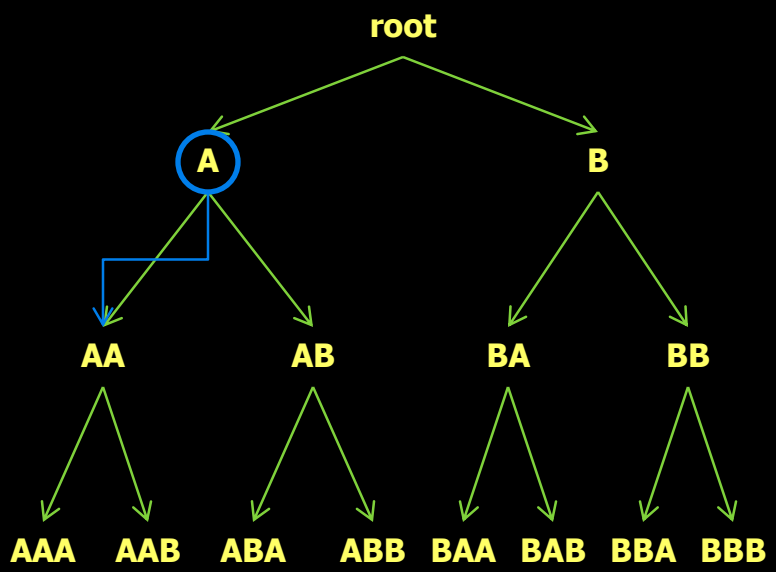
t_near < t_split < t_far

⇒node = near
⇒stack.push(far)

Стек: ()

kd-tree traversal



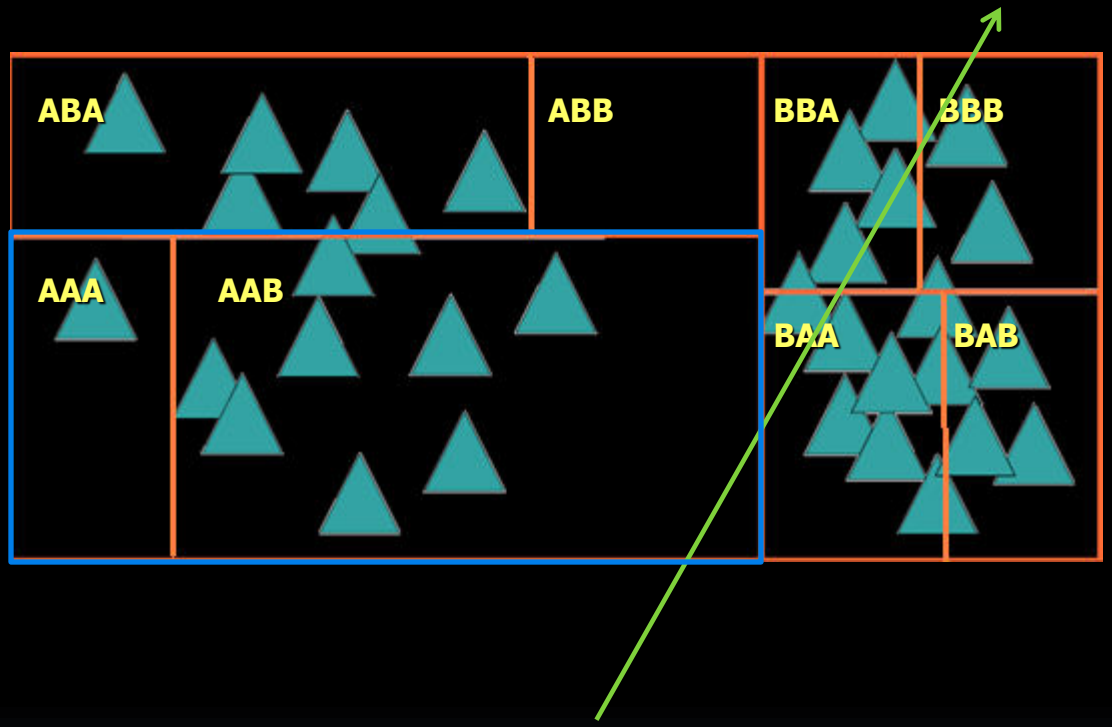


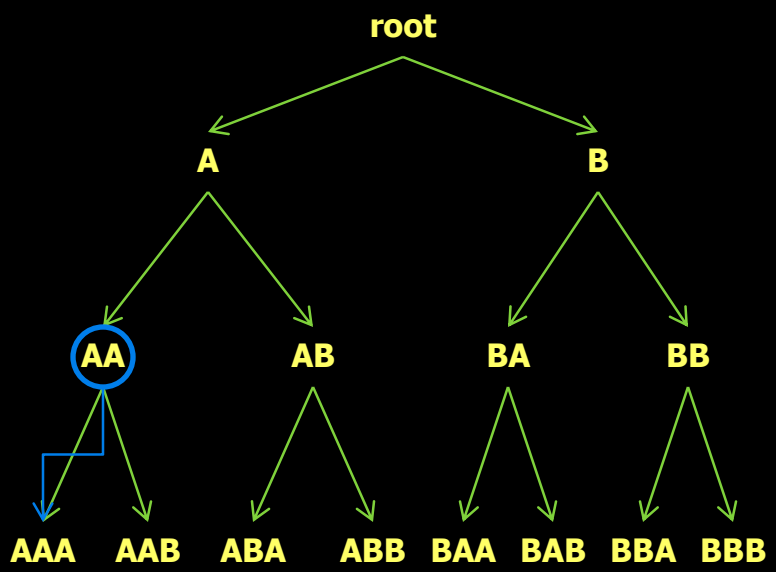
t_near < t_far < t_split

⇒node = near

Стек: (B)

kd-tree traversal



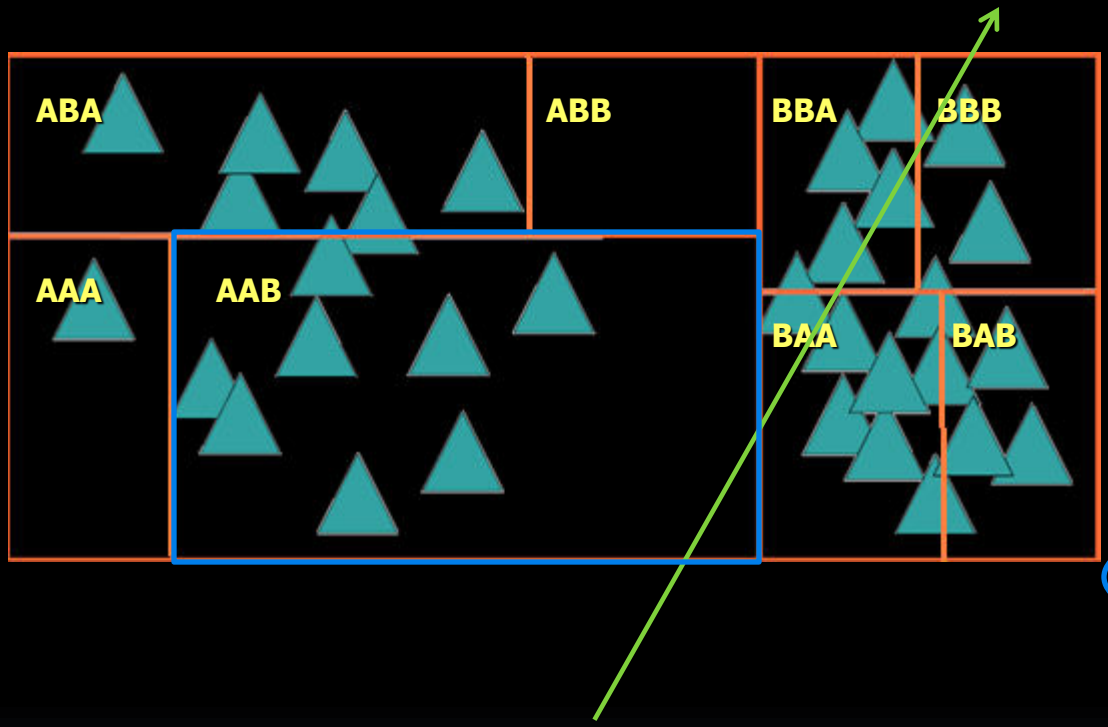


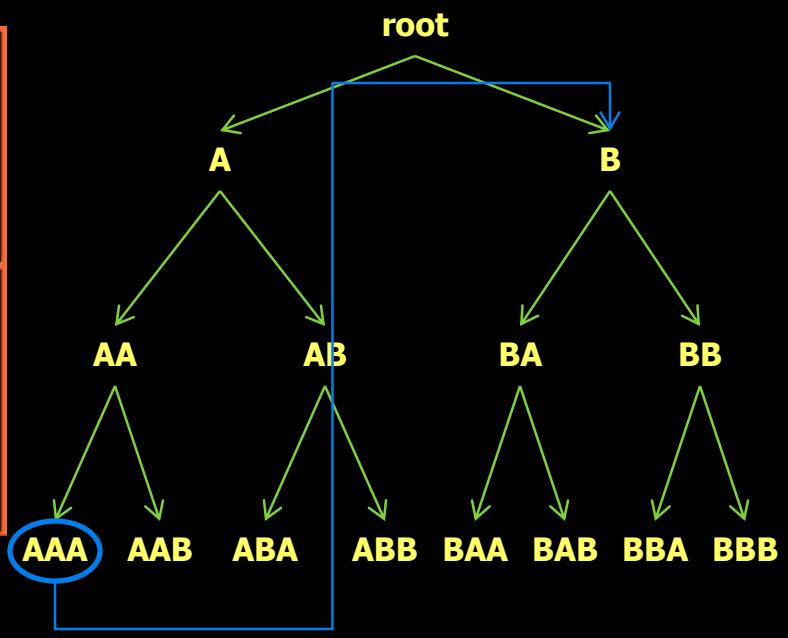
t_near < t_far < t_split

⇒node = near

Стек: (B)

kd-tree traversal



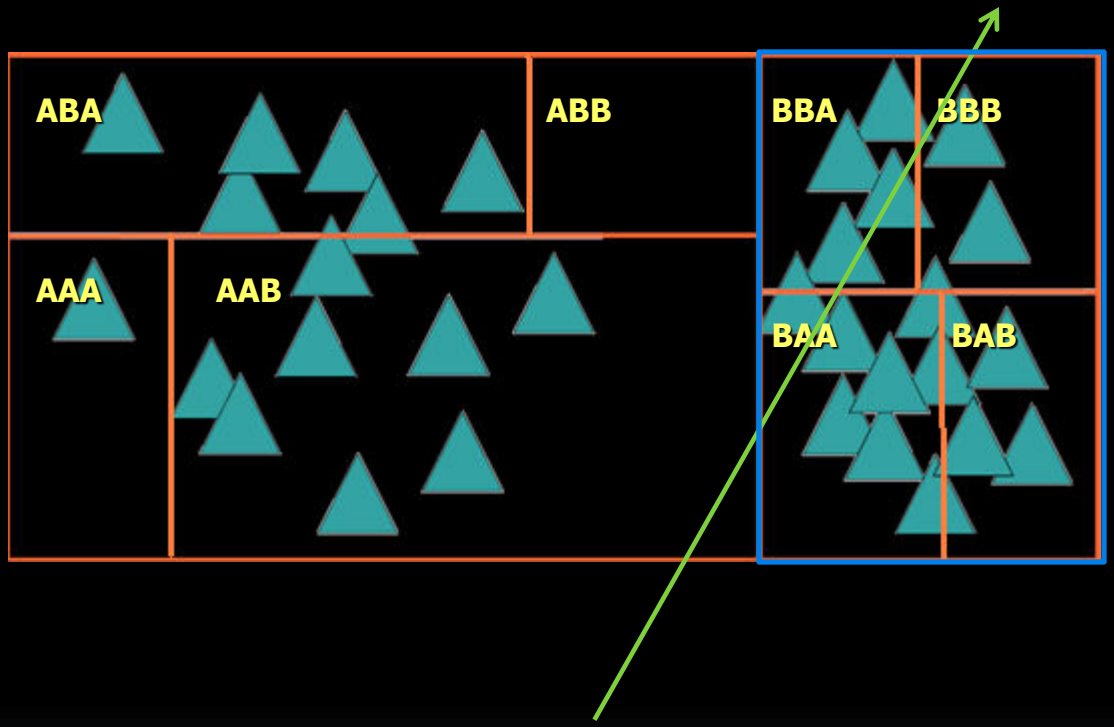


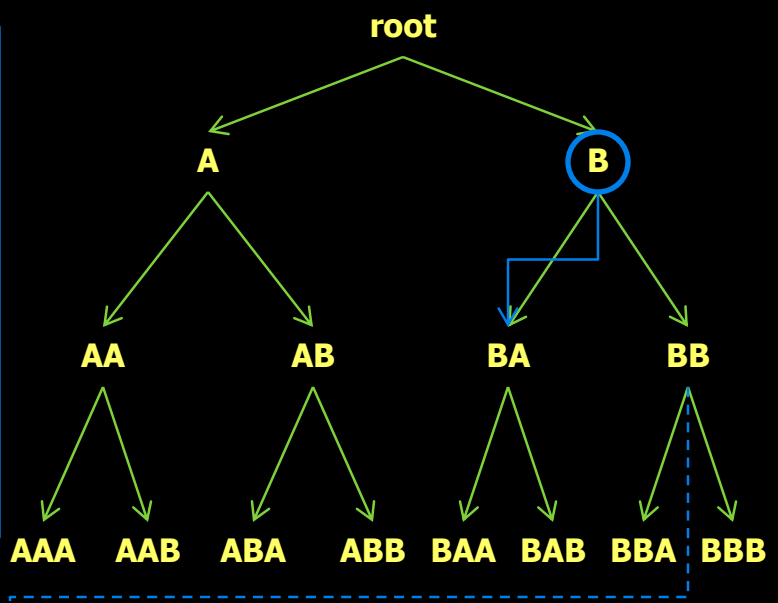
Leaf, no intersection found

⇒node = stack.pop()

Стек: (B)

kd-tree traversal



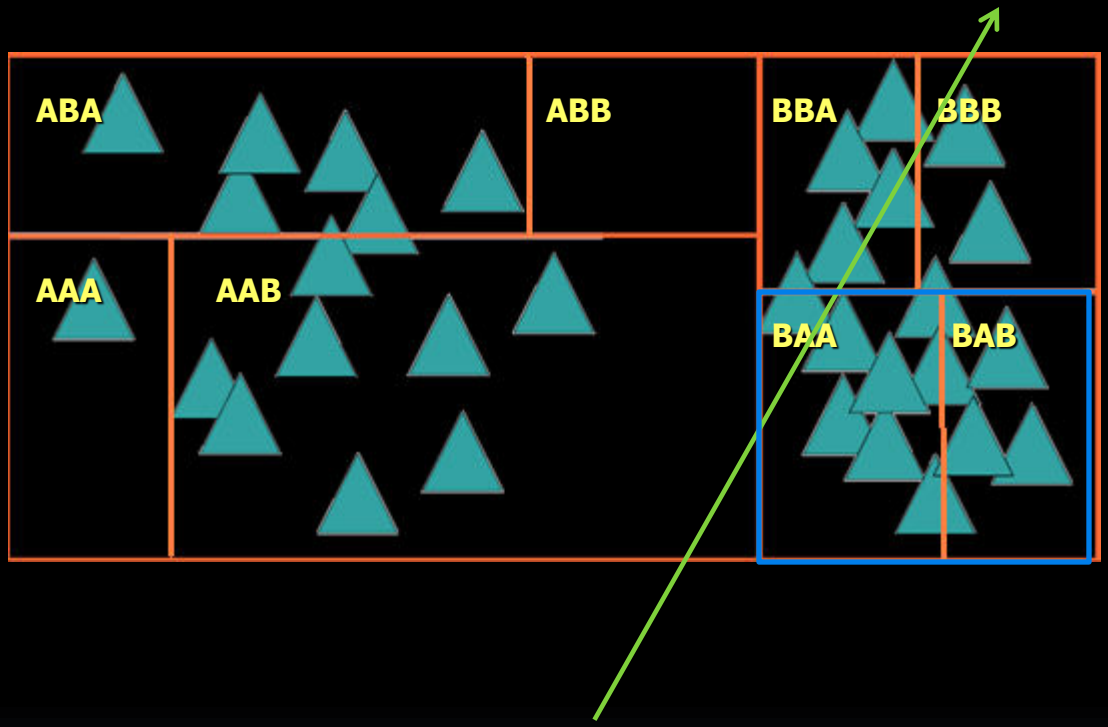


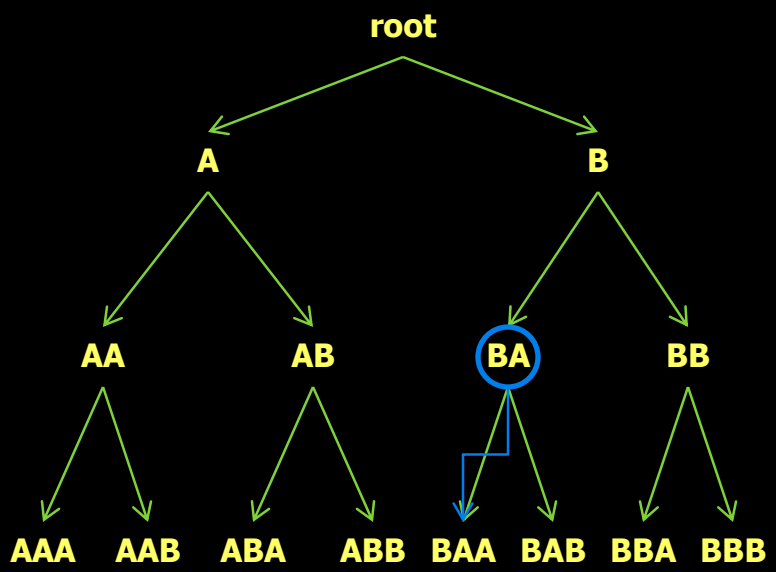
t_near < t_split < t_far

⇒node = near
⇒stack.push(far)

Стек: ()

kd-tree traversal



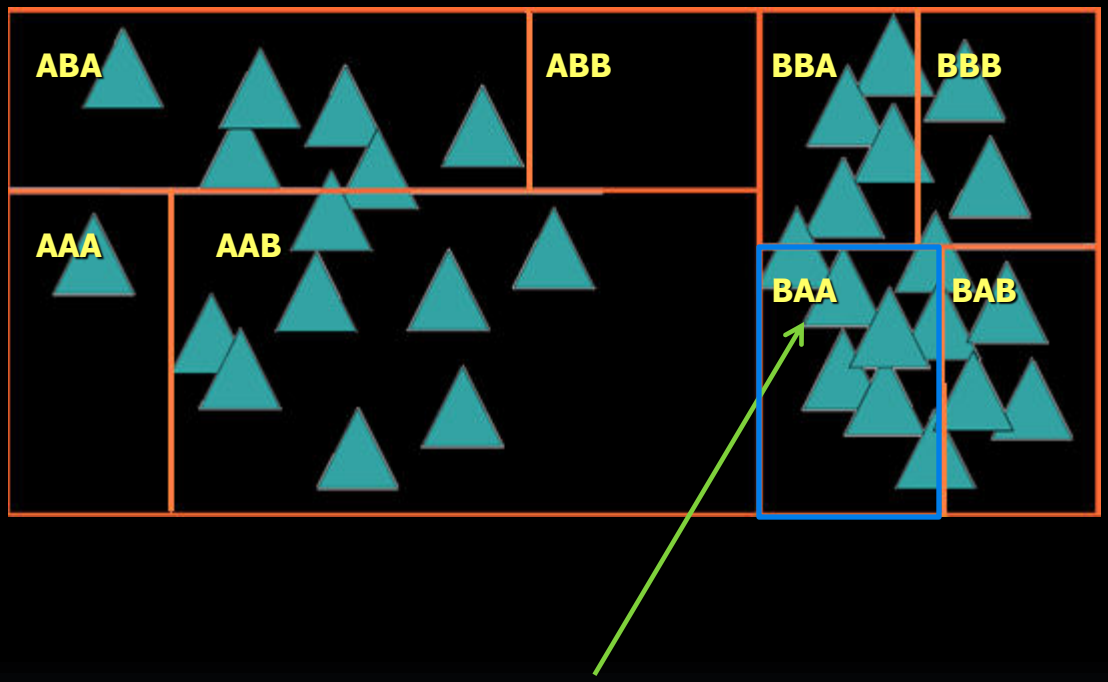


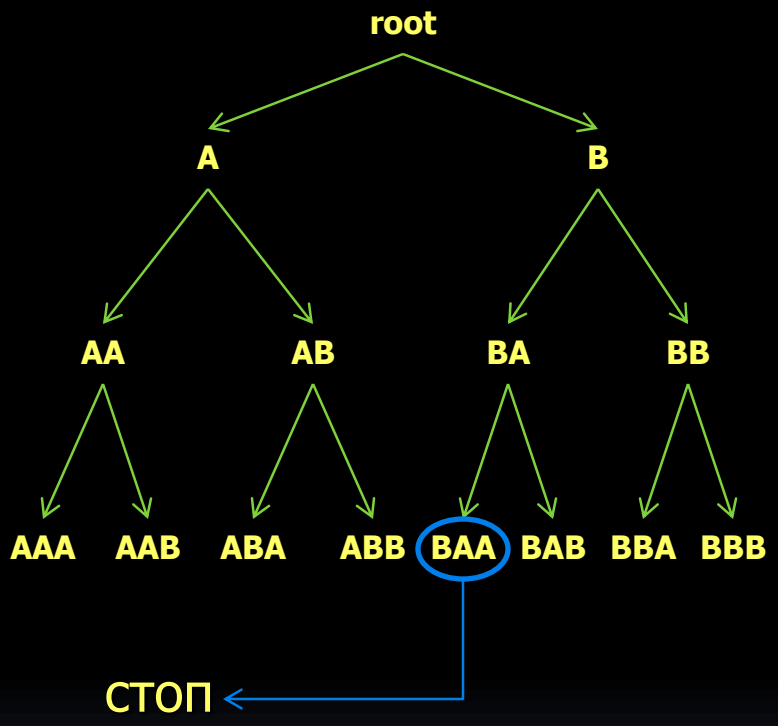
t_near < t_far < t_split

⇒node = near

Стек: (BB)

kd-tree traversal





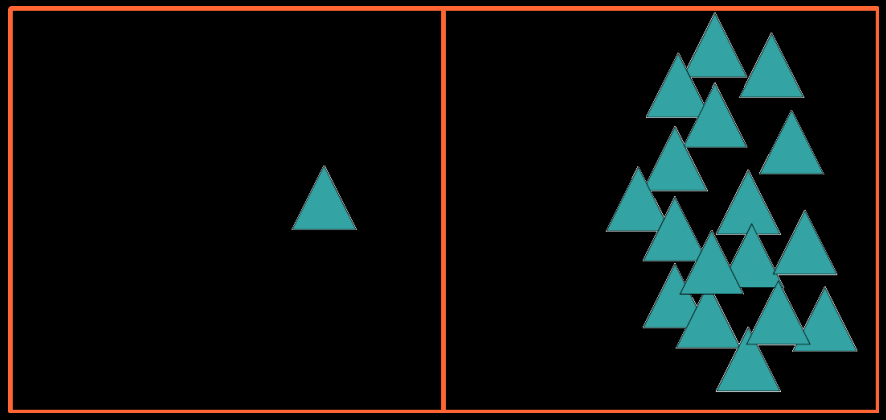
Leaf, intersection found

$\Rightarrow$ stop traversal

Стек: (BB)

Построение kd-tree

По центру

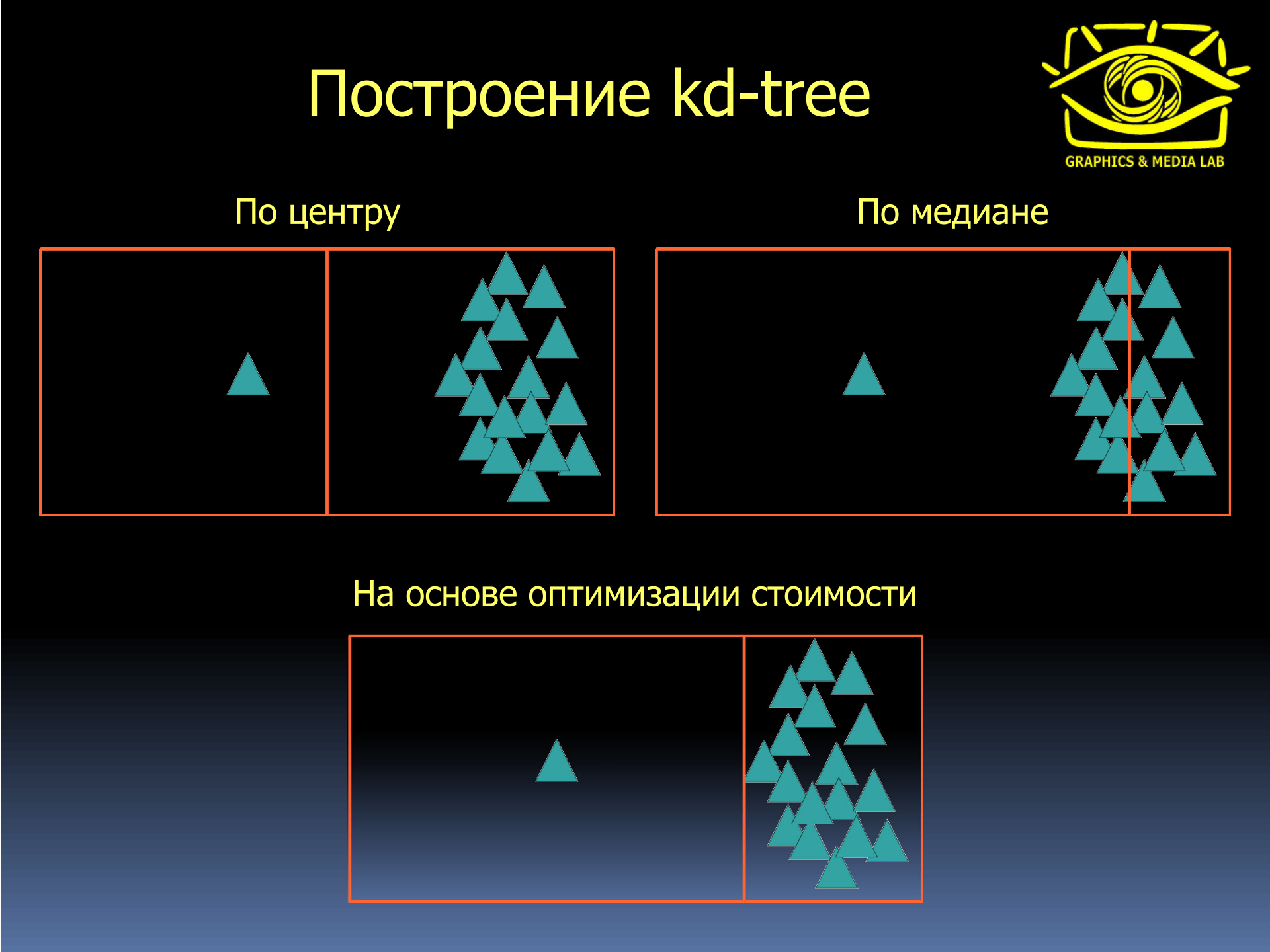


Построение kd-tree

По центру

По медиане

Построение kd-tree

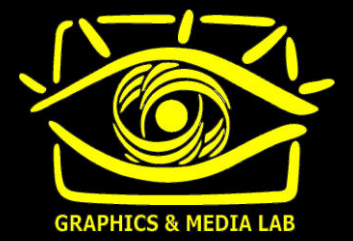


По центру

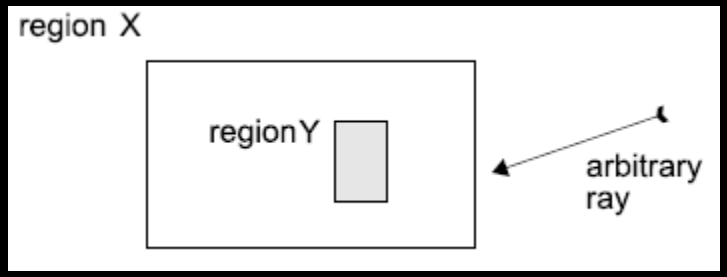
По медиане

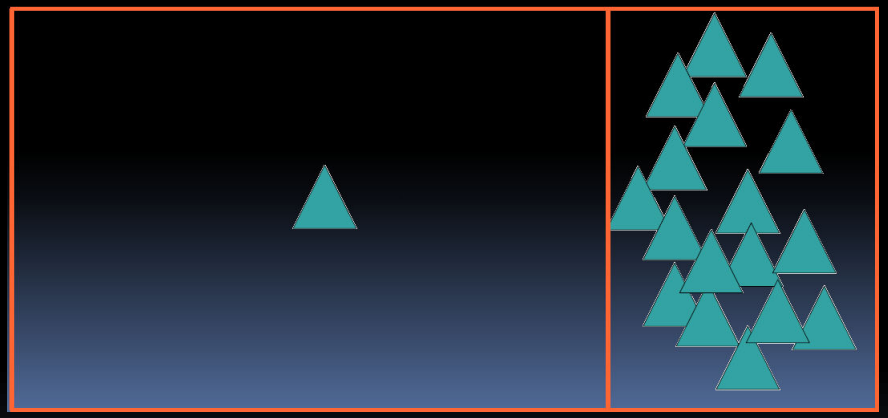
На основе оптимизации стоимости

Surface Area Heuristic (SAH)



Разбиение проводится на основе минимизации функции стоимости прослеживания луча в узле.





$$p_{Y|X} = \frac{SA(Y)}{SA(X)}$$

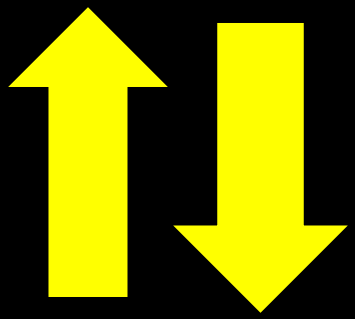
$$p_{Y|X} = \frac{(Y_w.Y_h + Y_w.Y_d + Y_h.Y_d)}{(X_w.X_h + X_w.X_d + X_h.X_d)}$$

SAH(x) = Empty_Cost + N_left*p(x) + N_right*(1-p(x))
p(x) = p_left

В фотонных картах:

$$VVH(x) = C_{ts} + \frac{C_L(x)V(d_L(x) \pm R)}{V(d \pm R)} + \frac{C_R(x)V(d_R(x) \pm R)}{V(d \pm R)}$$

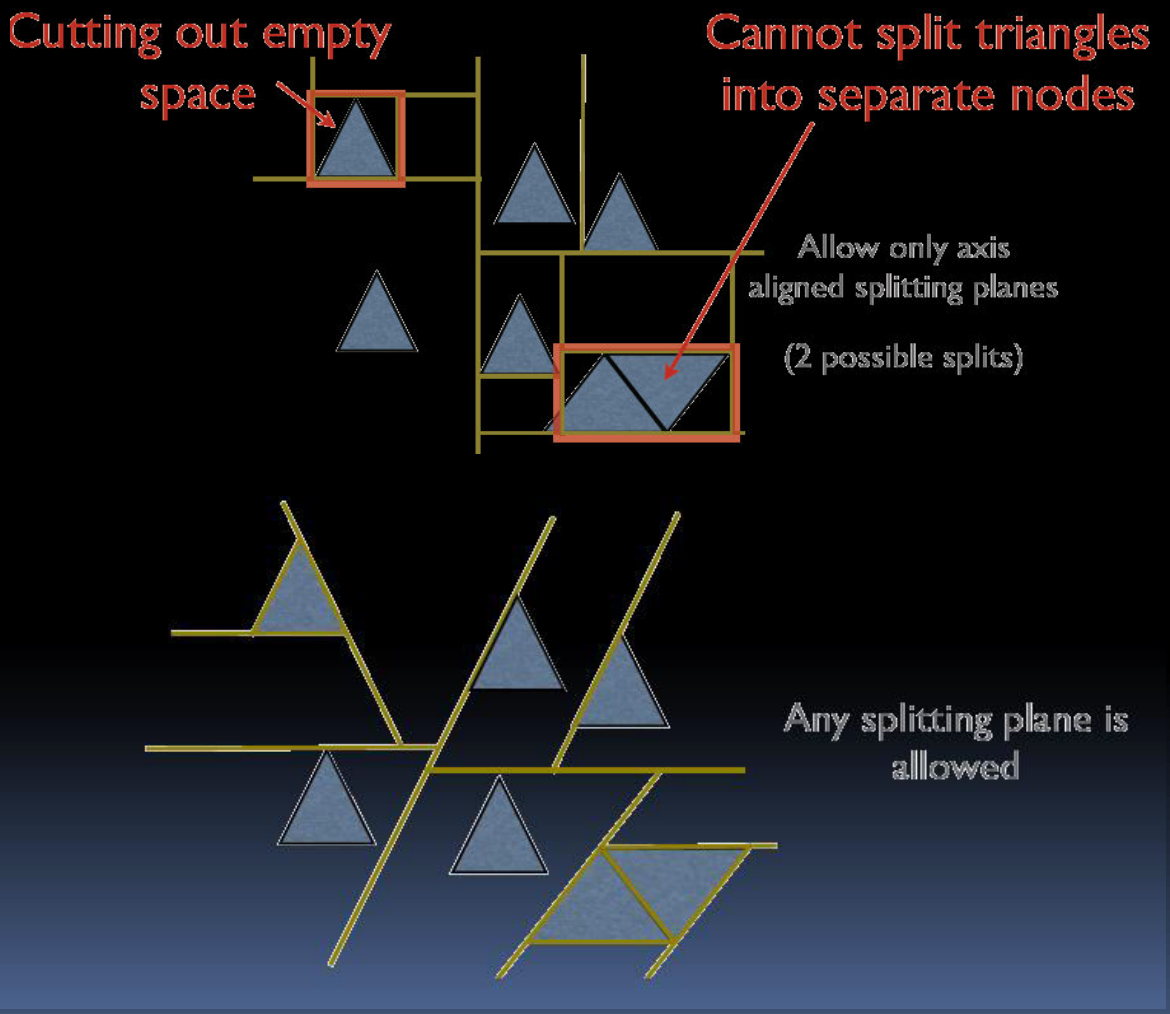
Kd tree



```
struct KdTreeNode
{
    unsigned int flagDimAndOffset;
    // bits 0..1 : splitting dimension
    // bits 2..30 : offset to left child
    // bit 31 : flag whether node is a leaf
    float splitCoordinates;
};
```

- Преимущества
  - Адаптивность
  - Компактное представление
  - Простые алгоритмы поиска, не зависят или слабо зависят от размерности пространства.
  - 8 байт kd tree лучше чем 8 байт BSP (почему?)
- Недостатки
  - Поиск на GPU: нужен стек
  - Нетривиальный алгоритм построения

Kd tree vs classic BSP

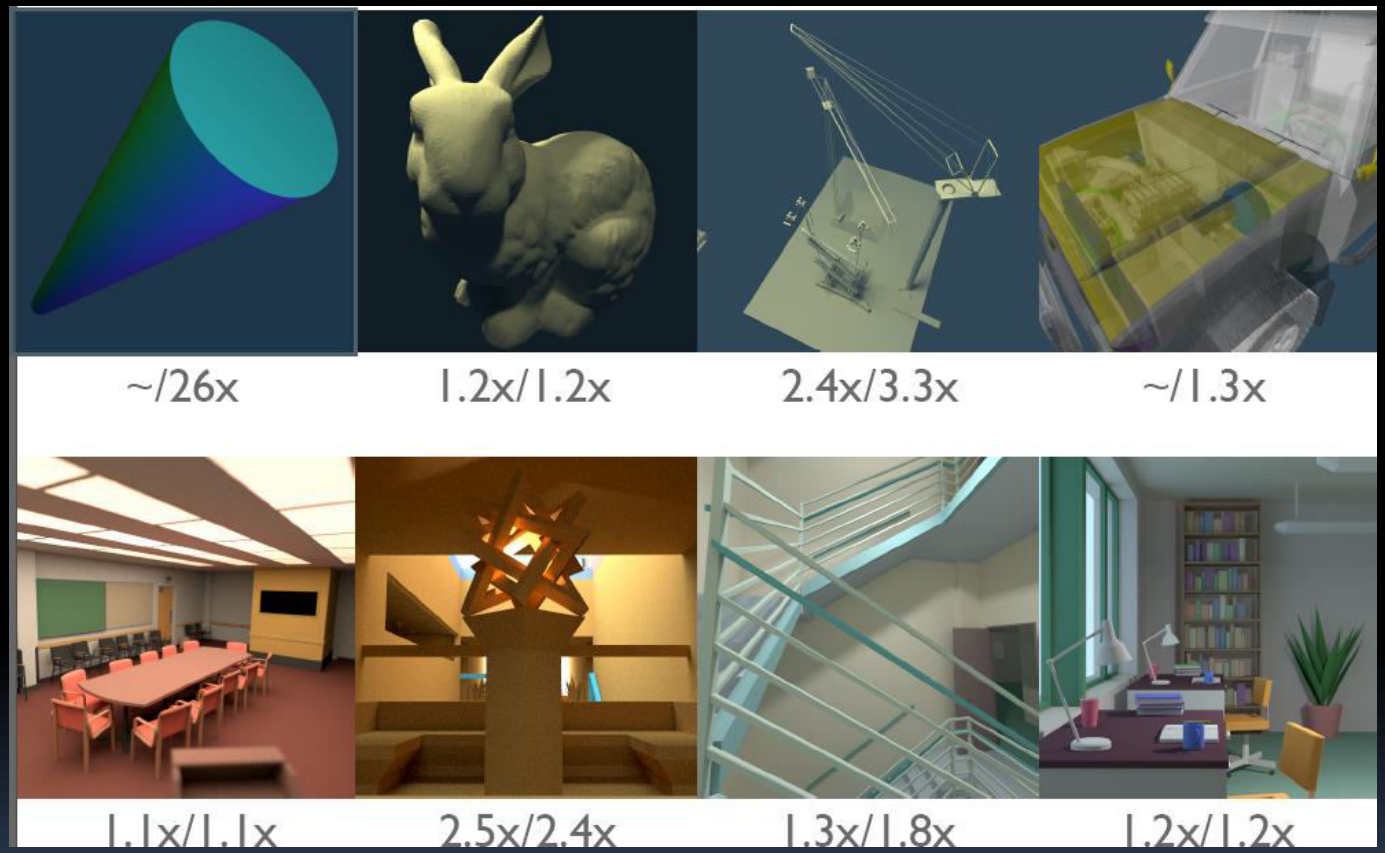


Kd tree vs classic BSP

GRAPHICS & MEDIA LAB

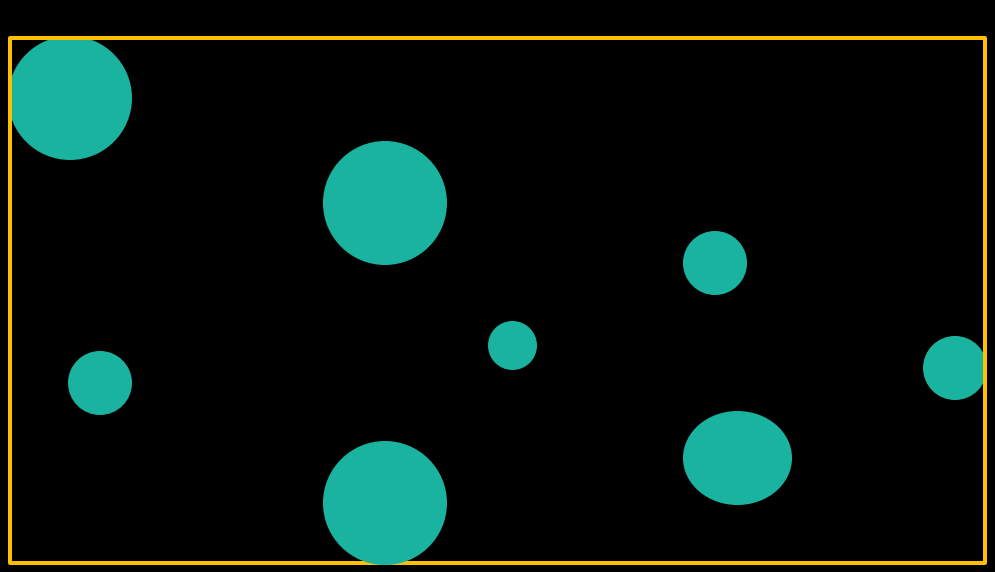
- kd-tree
  - Можно построить SAH дерево за O(n*log(n)) (+)
  - Компактное представление данных (+)
  - Простые алгоритмы поиска (+)
  - Не всякая геометрия хорошо разбивается (-)
- BSP
  - Теоретически, BSP – более эффективно (+)
  - SAH дерево строится очень долго (-)
  - Узлы имеют сложную форму (-)

Kd tree vs classic BSP



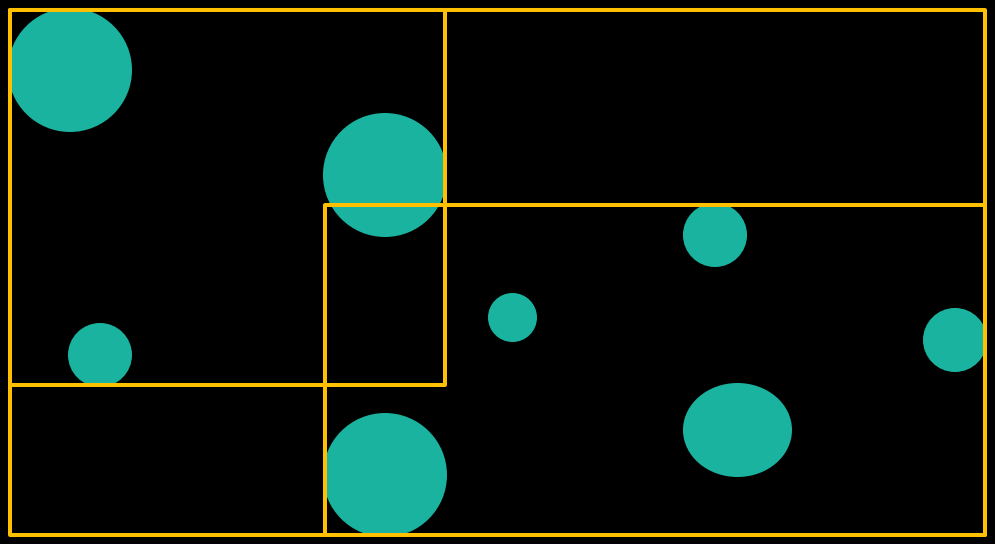
Ускорение: raytraced/raycasted

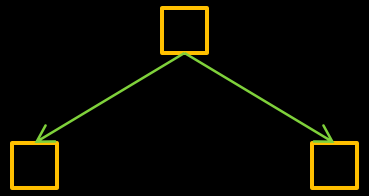
Bounding Volume Hierarchy (BVH)



- Иерархия вложенных (возможно пересекающихся) объемов.
- В каждом узле дерева необходимо хранить информацию об ограничивающем объем примитиве.

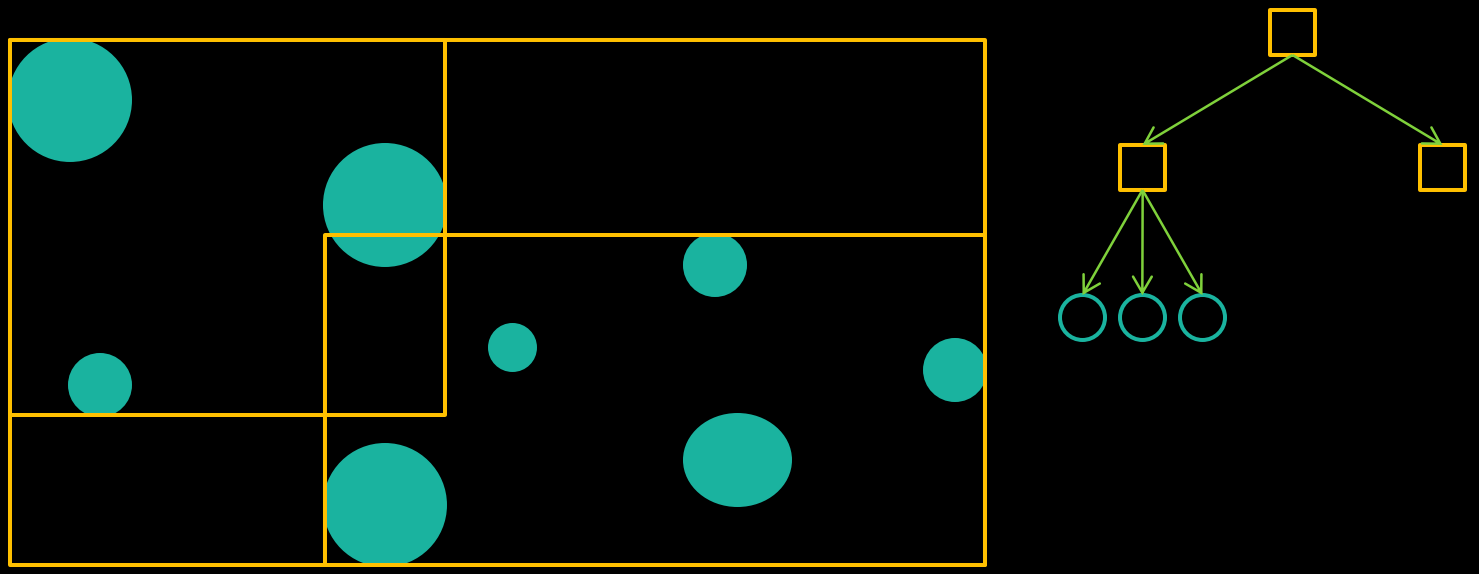
Bounding Volume Hierarchy (BVH)





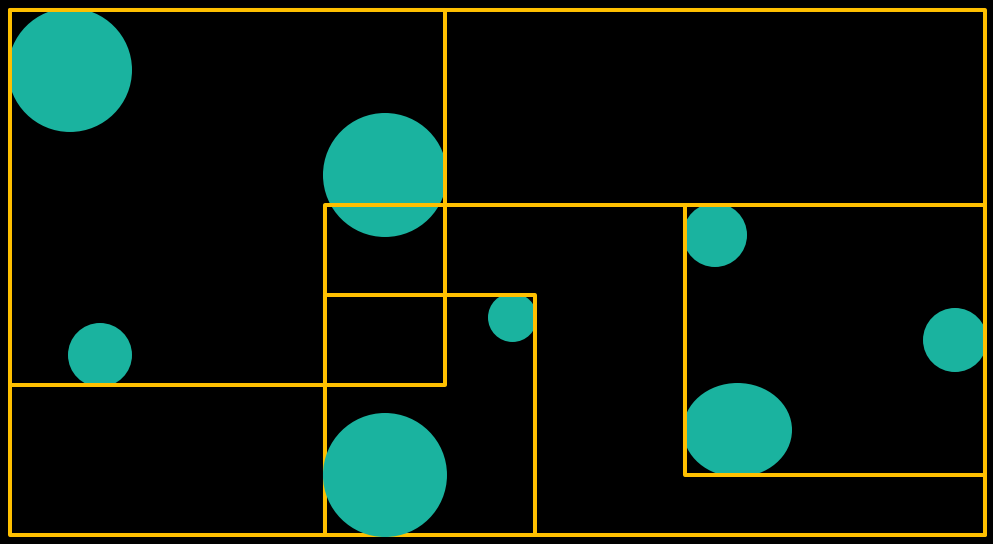
• Иерархия вложенных (возможно пересекающихся) объемов.
• В каждом узле дерева необходимо хранить информацию об ограничивающем объем примитиве.

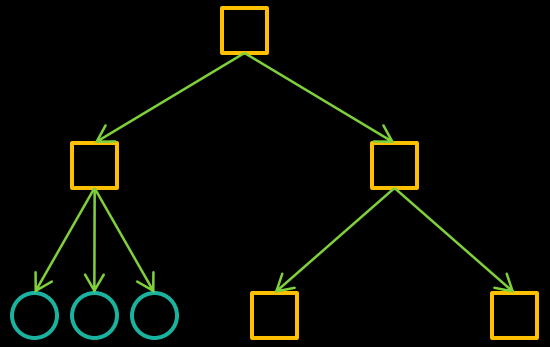
Bounding Volume Hierarchy (BVH)



- Иерархия вложенных (возможно пересекающихся) объемов.
- В каждом узле дерева необходимо хранить информацию об ограничивающем объем примитиве.

Bounding Volume Hierarchy (BVH)

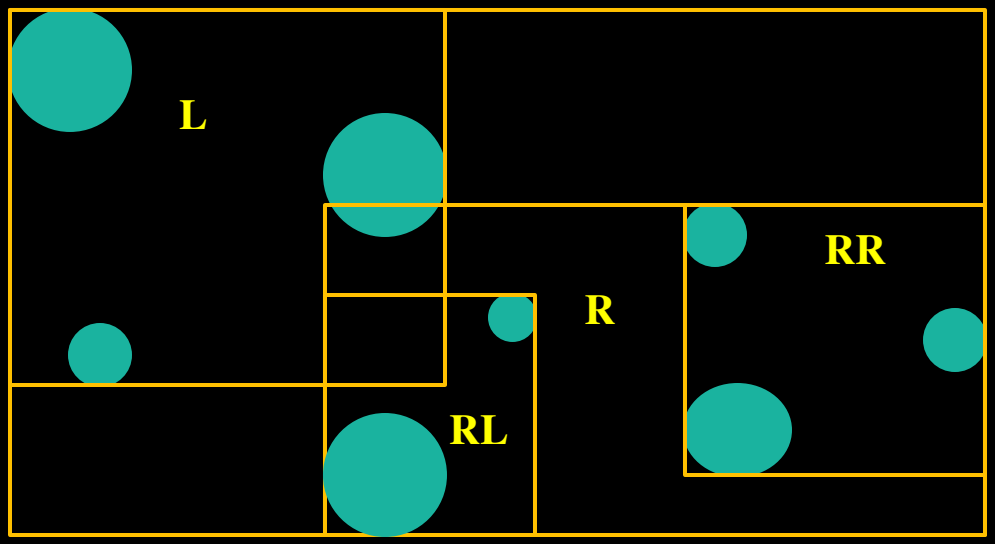


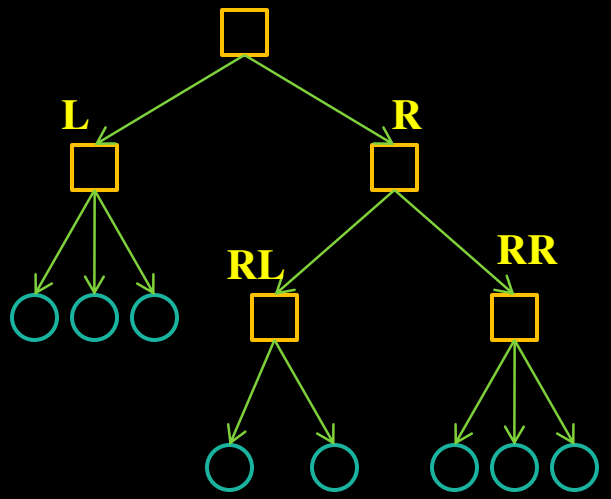


• Иерархия вложенных (возможно пересекающихся) объемов.

• В каждом узле дерева необходимо хранить информацию об ограничивающем объем примитиве.

Bounding Volume Hierarchy (BVH)

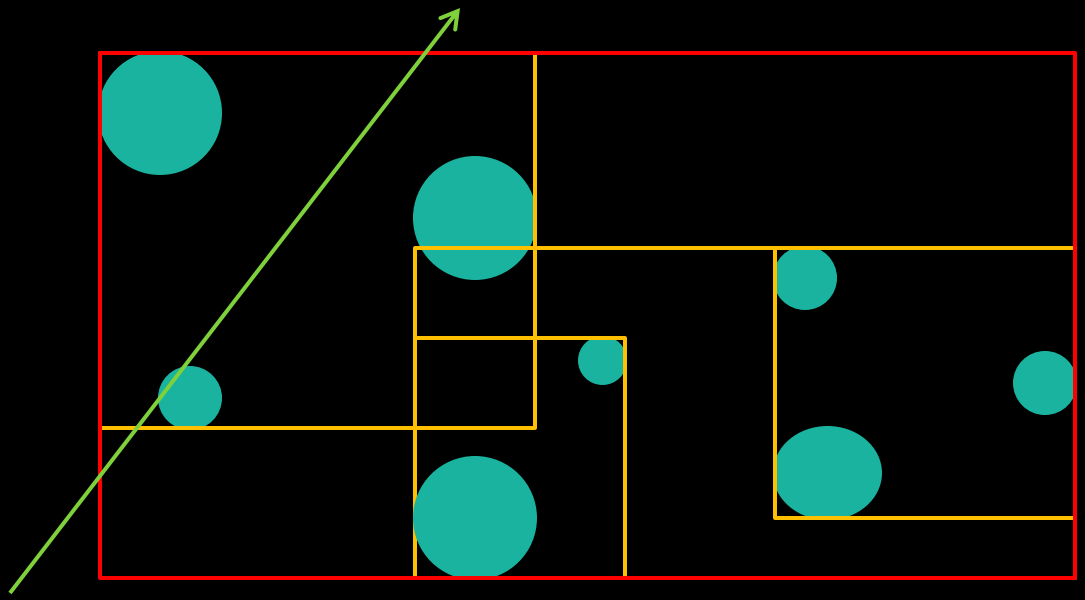


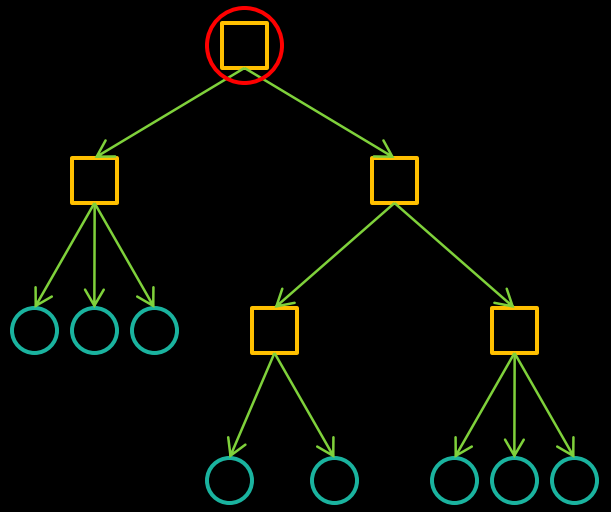


- Иерархия вложенных (возможно пересекающихся) объемов.
- В каждом узле дерева необходимо хранить информацию об ограничивающем объем примитиве.

Bounding Volume Hierarchy (BVH)

GRAPHICS & MEDIA LAB

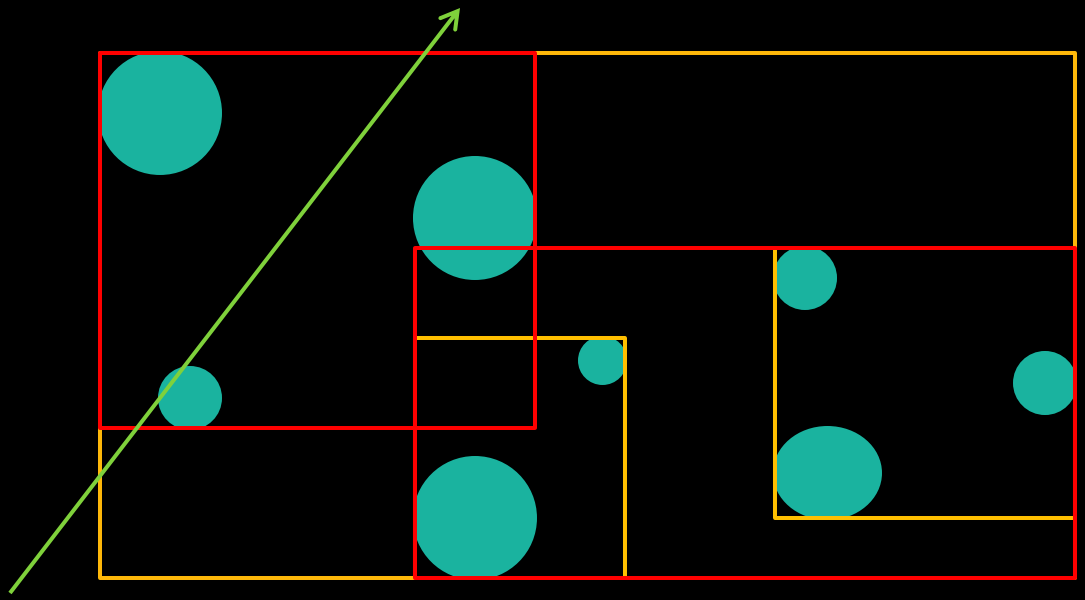


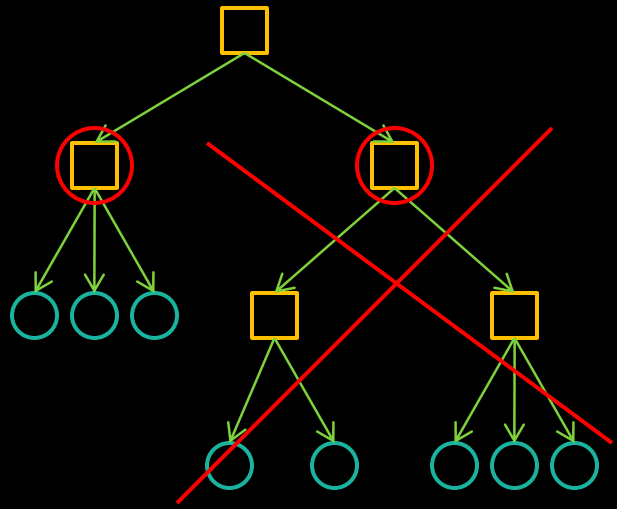


На каждом уровне ищем пересечение нашего объекта (например луча) со всеми дочерними узлами.

Bounding Volume Hierarchy (BVH)

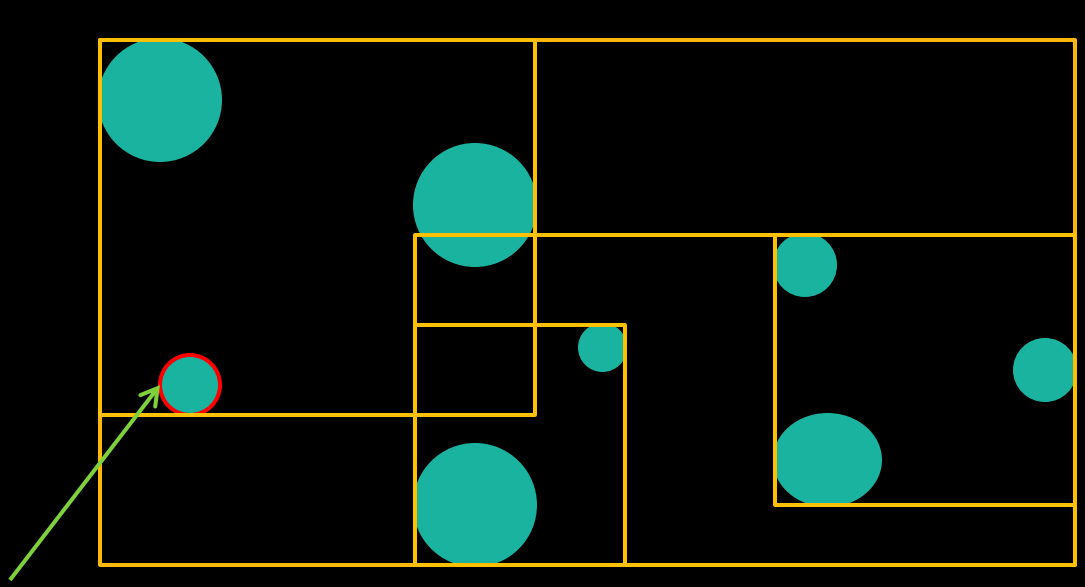
GRAPHICS & MEDIA LAB

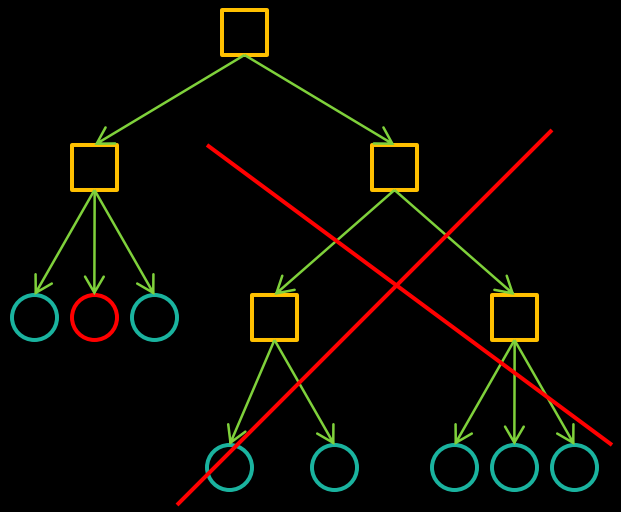




На каждом уровне ищем пересечение нашего объекта (например луча) со всеми дочерними узлами.

Bounding Volume Hierarchy (BVH)





Если лист, пересекаем с примитивами

Bounding Volume Hierarchy (BVH)

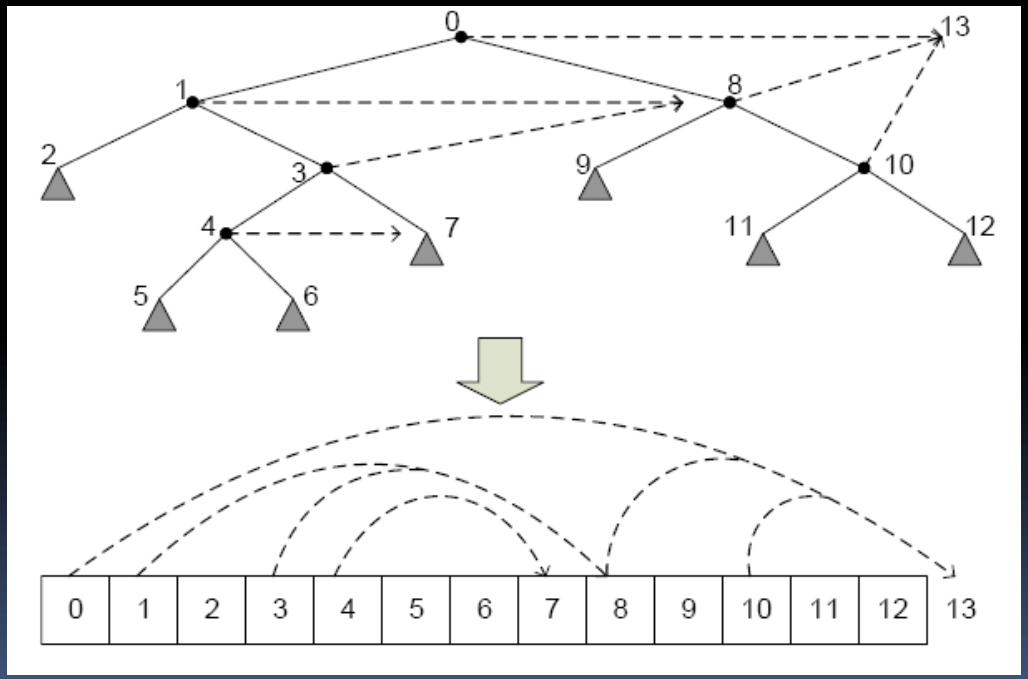
- Преимущества
  - Быстро локализует геометрию на пустых пространствах
  - Трассировка на GPU: для поиска не нужен стек
  - Растеризация: легко отсекать по пирамиде видимости

- Недостатки
  - BVH относительно требовательны по памяти в случае сложной геометрии
  - Несколько сложнее использовать SAH при посторении

BVH, почему не нужен стек?

GRAPHICS & MEDIA LAB

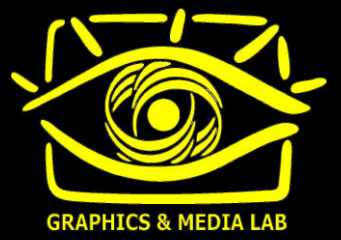
- Основная идея
  - ✓ Траверсим дерево строго слева направо
  - ✓ Сохряем в узлах ссылки “наверх”, вернее, “направо”



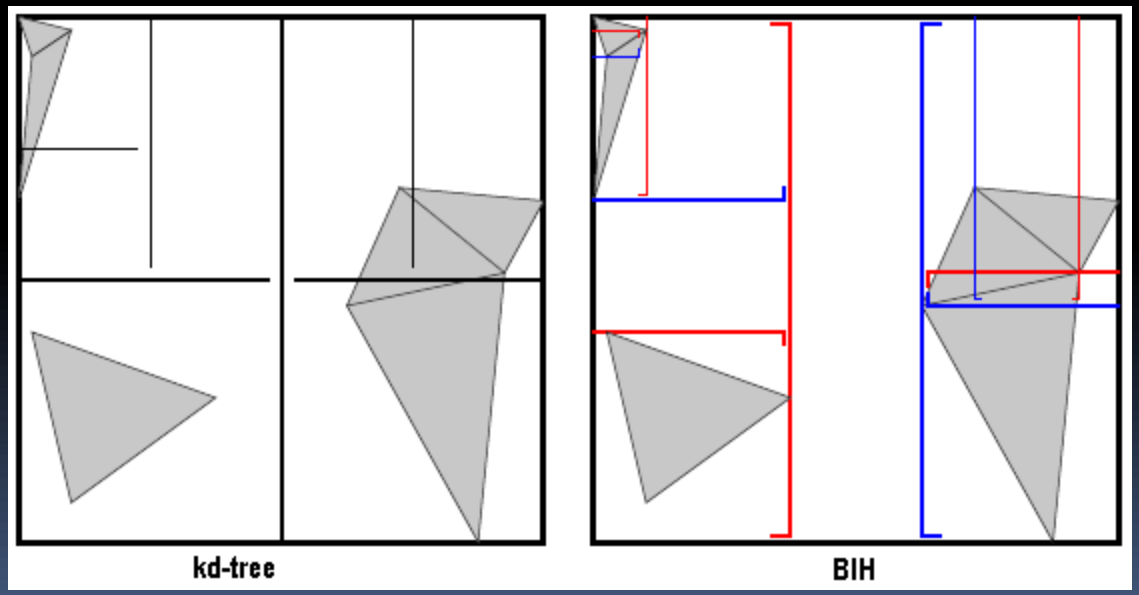
Bounding Volume Hierarchy (BVH)

Virtual Ray на одноядерном Pentium4 (2.4 Ghz)
в разрешение 1024x768 - 20 fps

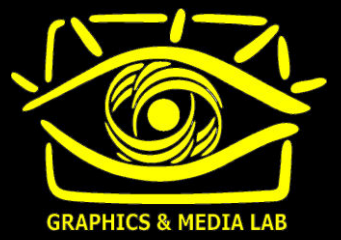
Bounding Interval Hierarchy (BIH)



- Основные моменты
  - Гибрид BVH и kd-tree
  - В каждом узле храняться 2 плоскости
  - Геометрия ограничивается «внутрь» или «наружу»



Bounding Interval Hierarchy (BIH)



- Преимущества (те же, что и в kd-tree)
  - Компактное представление (как в kd-tree)
  - Простые алгоритмы поиска
  - Меньшая глубина по сравнению с kd-tree

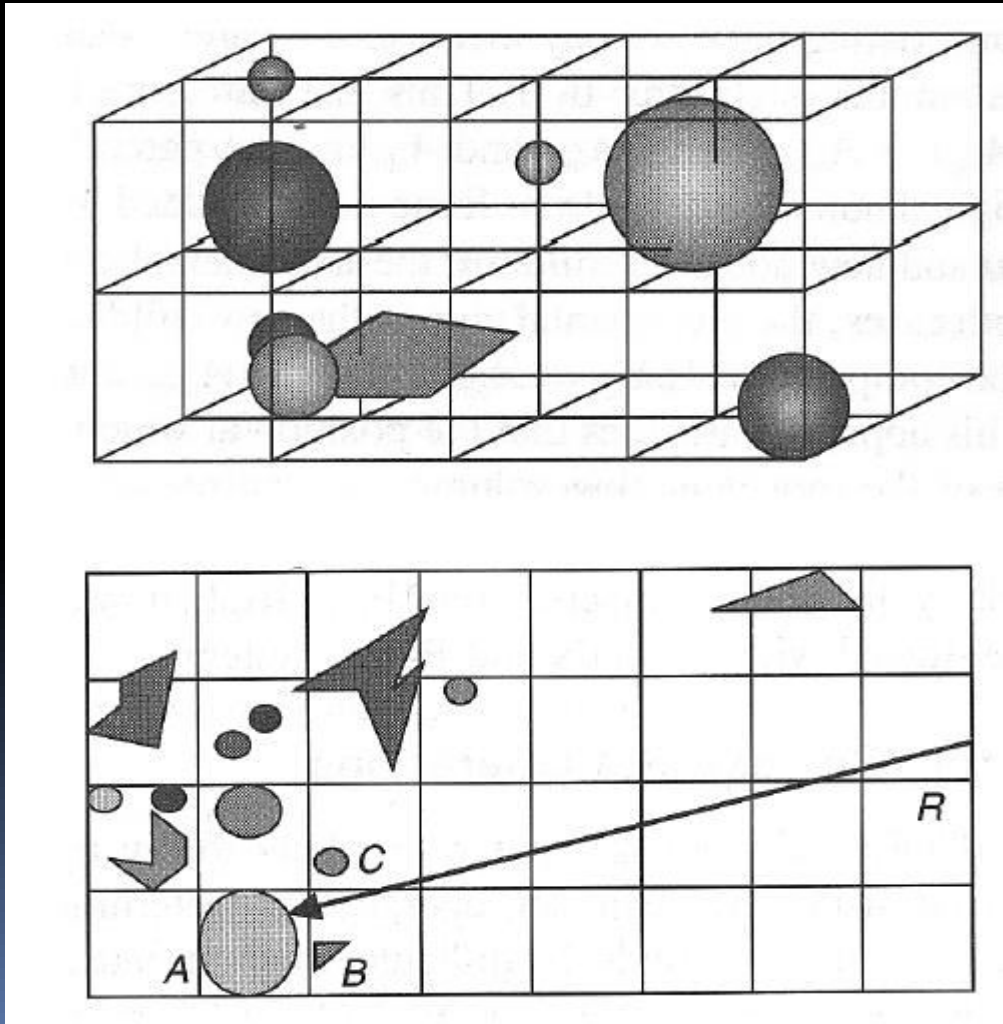
- Недостатки (те же, что и в kd-tree)
  - Трассировка лучей на GPU : нужен стек.
  - Нетривиальный алгоритм построения

BART Museum (75,884 triangles)	*kd*	BIH
a) Average fps (primary rays only)	0.39	2.04
Rendering time for complete animation (msec)	776,568	147,002
b) Average fps (average 4.024 rays per pixel)	0.28	0.49
Rendering time for complete animation (msec)	1,057,259	614,503

Регулярная сетка

Поиск сводится к вычислению индексов

i = kx*(point.x – b.min.x)
j = ky*(point.y – b.min.y)
k = kz*(point.z – b.min.z)

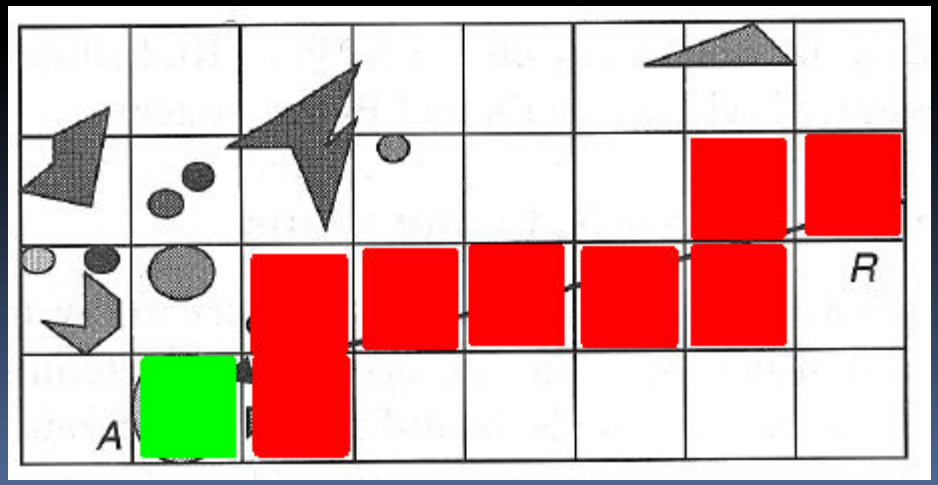


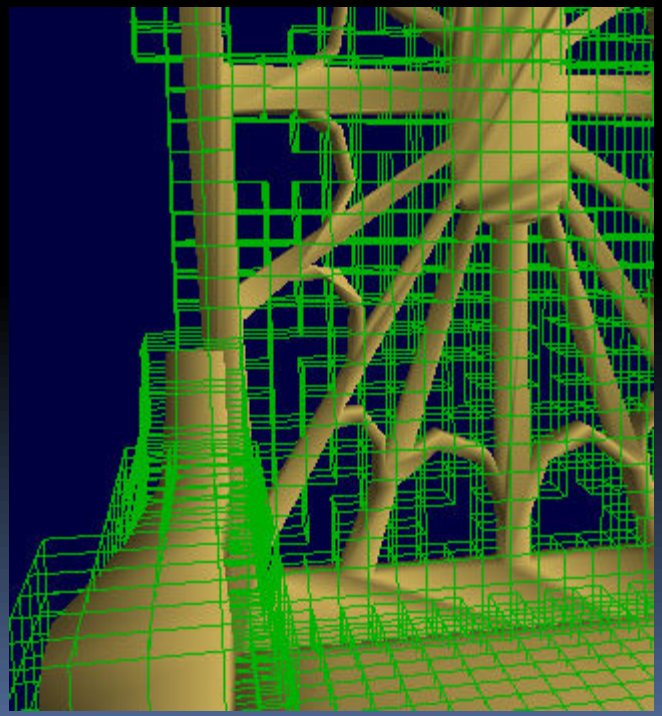
Регулярная сетка и трассировка лучей

GRAPHICS & MEDIA LAB

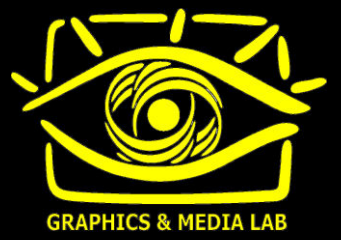
• Идея: использовать трехмерный аналог алгоритма Брезенхема

• Проблема точности: луч задан в координатах с плавающей точкой





Регулярная сетка

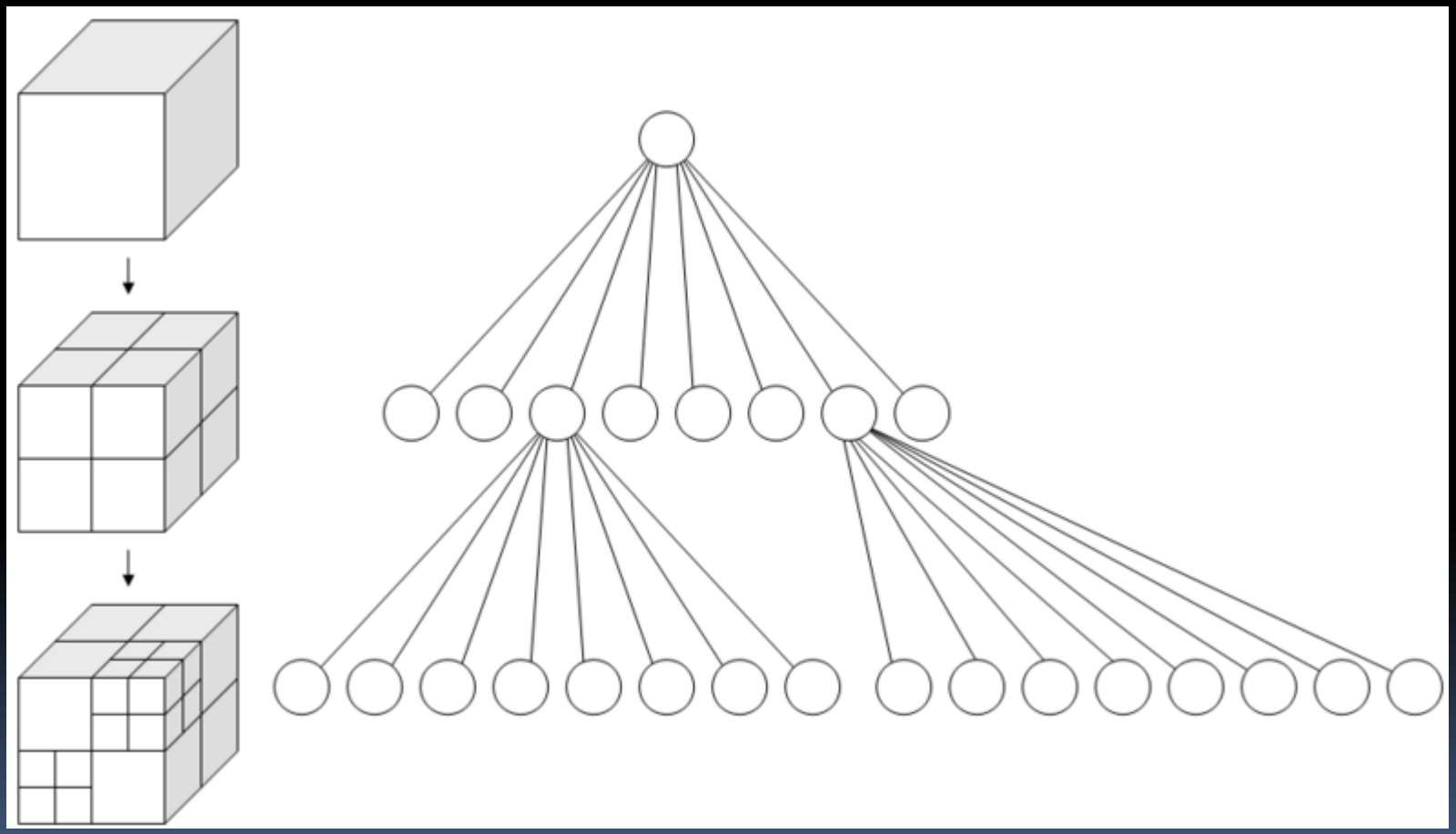


- Преимущества
  - Единственная из всех структур – неиерархическая!
  - Просто и бысто строится
  - Индексация, сложность поиска = O(1)
  - Трассировка лучей: алгоритм прослеживания, использующий только сложения и сдвиги.

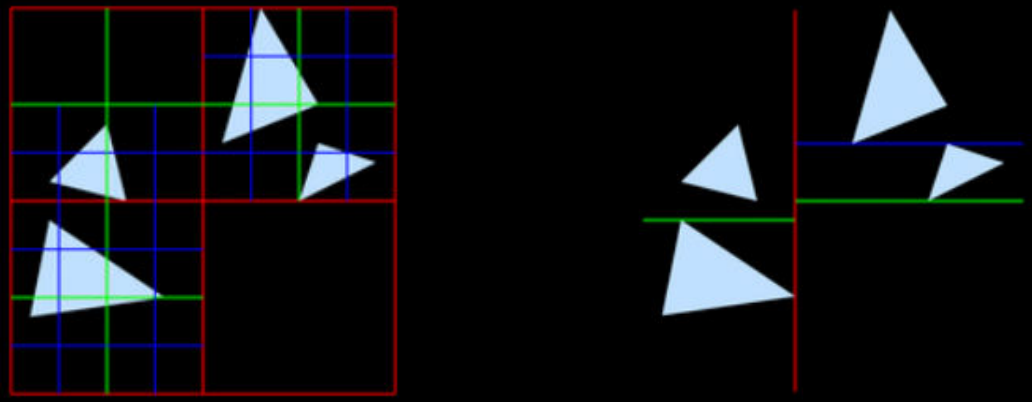
- Недостатки
  - Требует очень много памяти
  - Неадаптивна, проблема чайника на стадионе
  - Трассировка лучей: нет учета кэша ?
  - Растеризация: малопригодна

Октодерево

GRAPHICS & MEDIA LAB

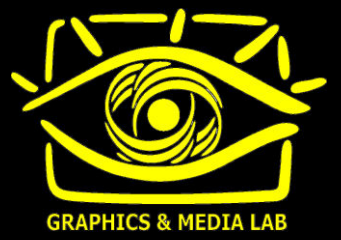


Октодерево



- Глубина дерева меньше ?
- Много пустых узлов
- Один и тот же примитив во многих листах
- Разбиение всегда производится по центру.

Иерархический Z-буффер



• Разбиваем проектную плоскость на некоторое число прямоугольников (тайлов), размером, например 32x32 пикселя.

• Если некий примитив полностью закрывает собой этот тайл, записываем в этот тайл среднее z-значение данного примитива

• При выводе очередного узла определяем, находятся ли его лицевые грани полностью в пределах тайла. Если это так и z-значение ближайшей вершины узла-куба больше z-значения тайла, то узел является скрытым, а значит, вся его геометрия тоже скрыта.

Traversal algorithm

GRAPHICS & MEDIA LAB

Current sub-node (state)	Exit plane YZ	Exit plane XZ	Exit plane XY
0	4	2	1
1	5	3	End
2	6	End	3
3	7	End	End
4	End	6	5
5	End	7	End
6	End	End	7
7	End	End	End

«End» означает, что луч покидает текущий узел

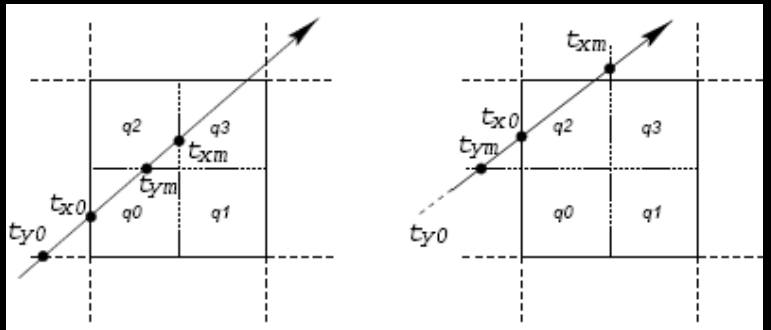


Figure 2: Sub-nodes crossed when $t_{x0} > t_{y0}$ (2D case).

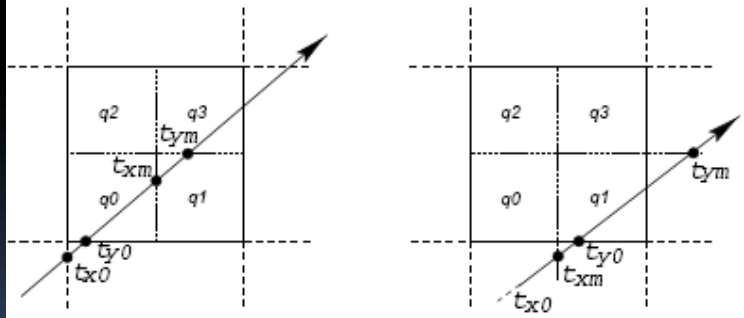
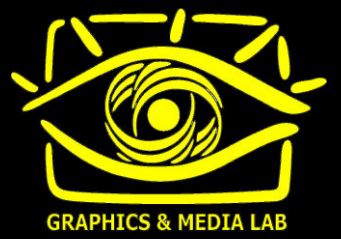


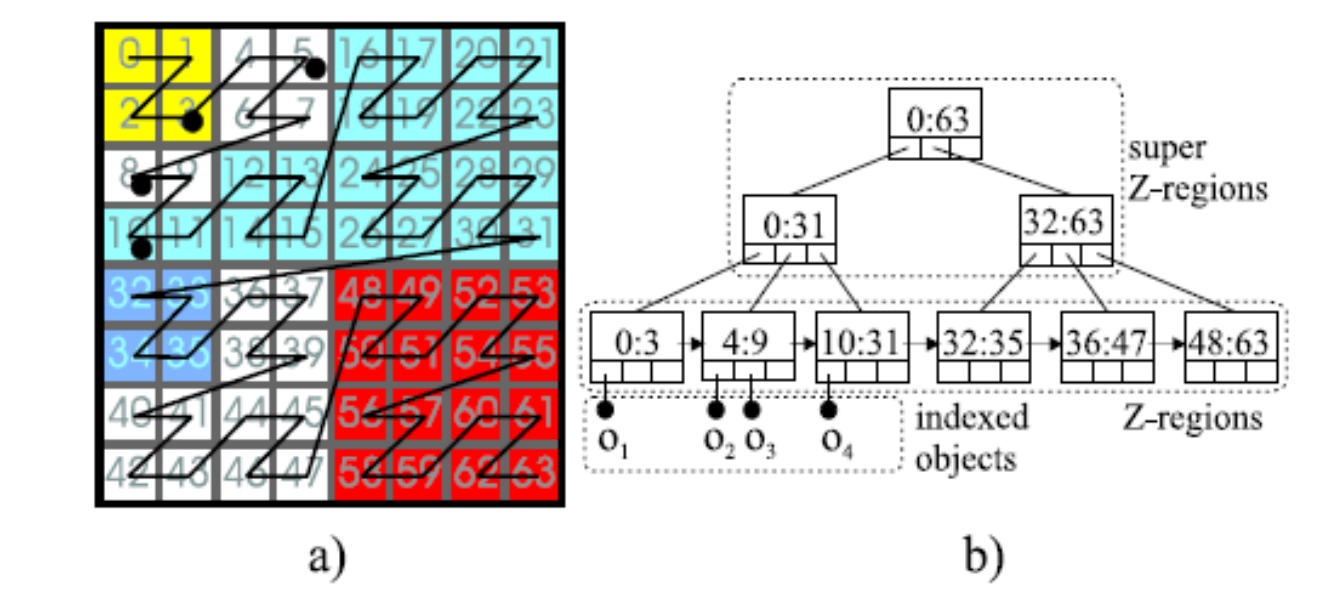
Figure 3: Sub-nodes crossed when $t_{y0} > t_{x0}$ (2D case).

Октодерево



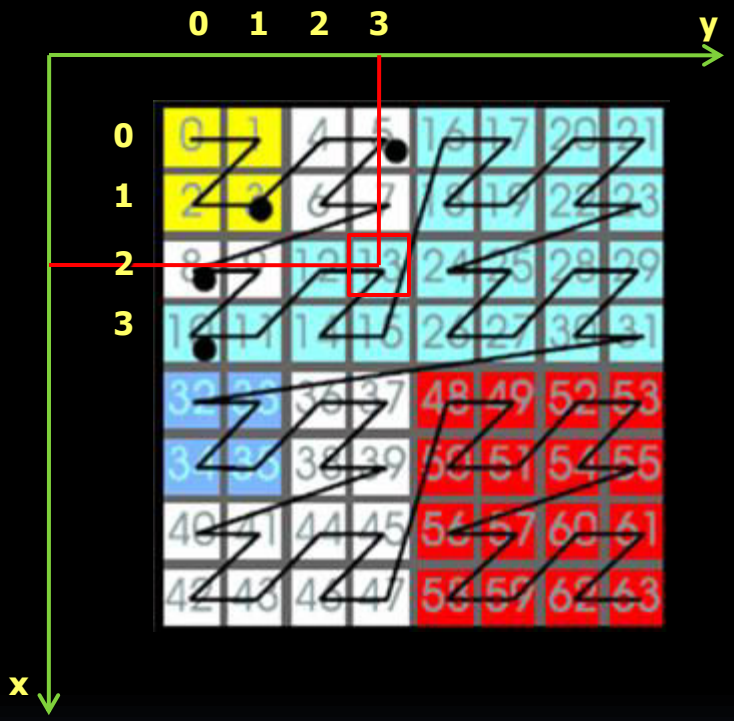
- Преимущества
  - Просто и быстро строится
  - Растеризация: простой алгоритм отсечения по пирамиде видимости
  - Растеризация: Иерархический Z-buffer
  - Воксельная визуализация
- Недостатки
  - Много пустых узлов
  - Не учитывает SAH – хуже чем kd-дерево
  - Учет одного и того-же примитива много раз
  - Трассировка лучей: сложный алгоритм обхода

Z-ordering (UB-tree)



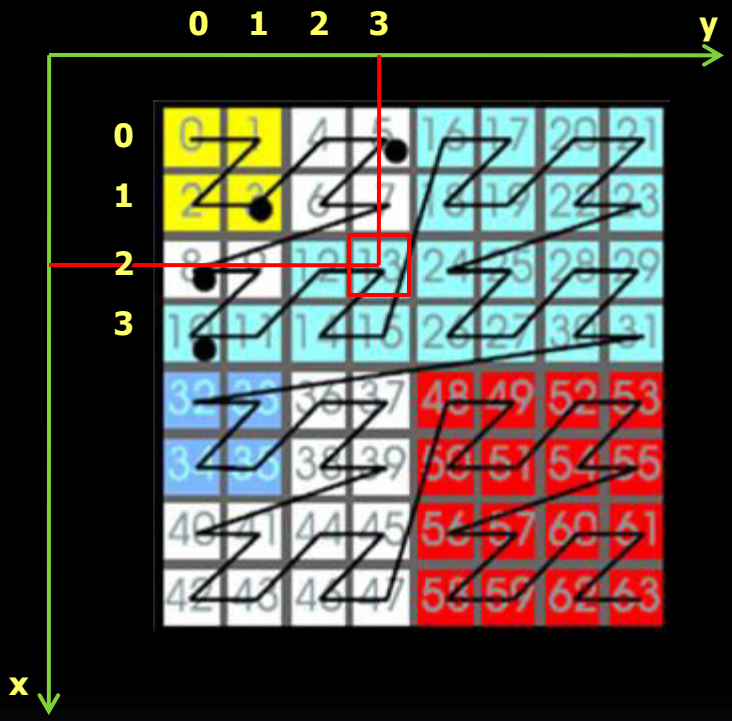
- Один из видов B+ дерева – маленькая глубина, путь от корня к листу занимает фиксированное число шагов
- Отображение 2D в 1D: Z-адресация
- Позволяют минимизировать количесто кэш промахов

Вычисление Z-индекса



2 op 3 == 13, op == ?

Вычисление Z-индекса

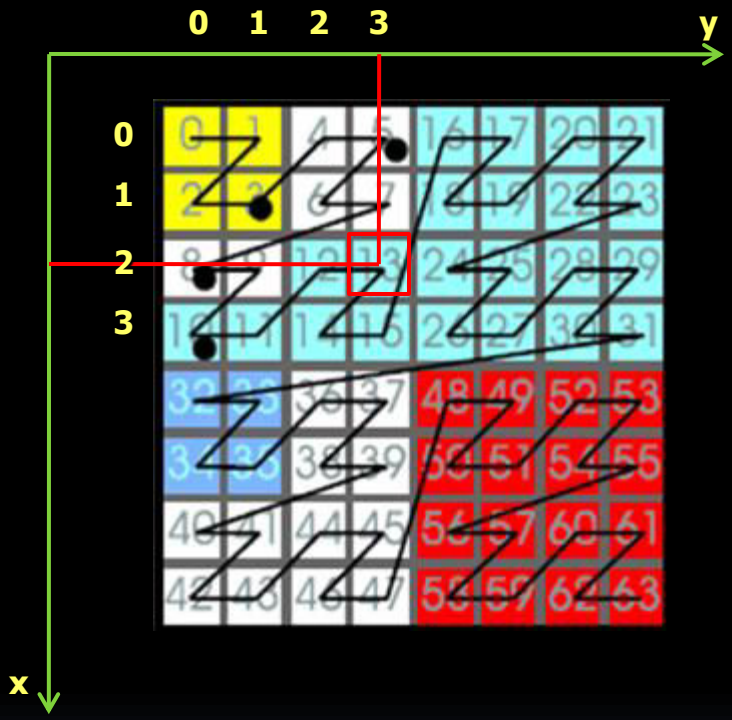


2 op 3 == 13, op == ?

2 → 010

3 → 011

Вычисление Z-индекса



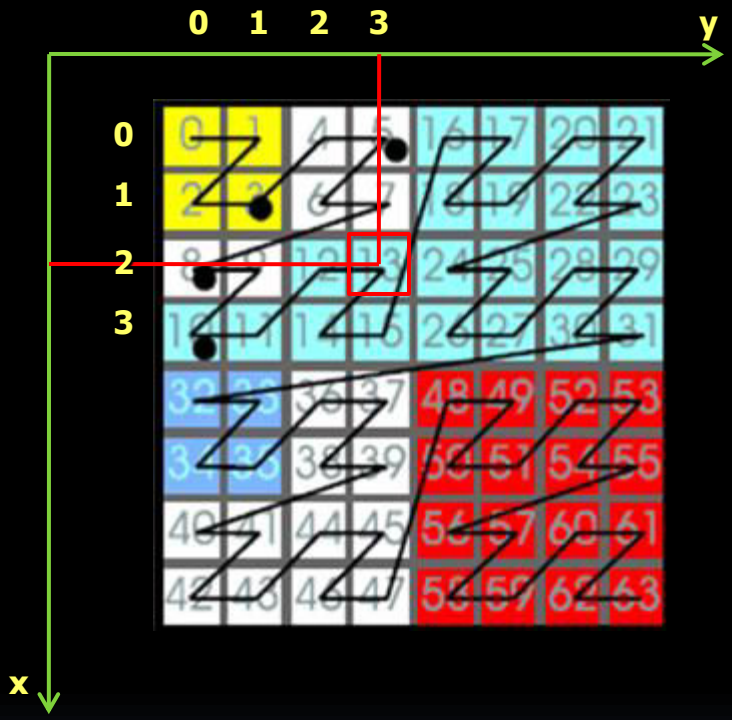
2 op 3 == 13, op == ?

2 → 010

3 → 011

0 1 0

Вычисление Z-индекса



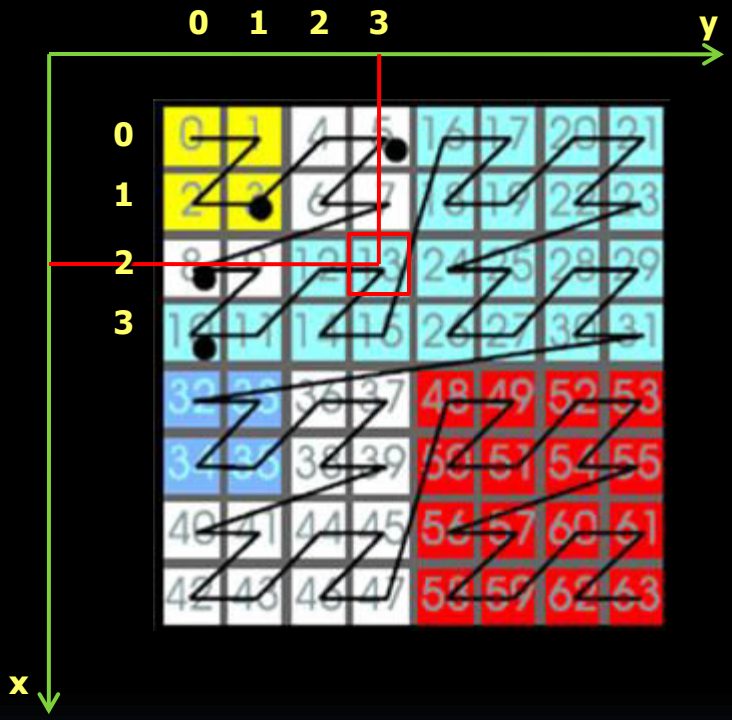
2 op 3 == 13, op == ?

2 → 010

3 → 011

0 0 1 1 0 1

Вычисление Z-индекса



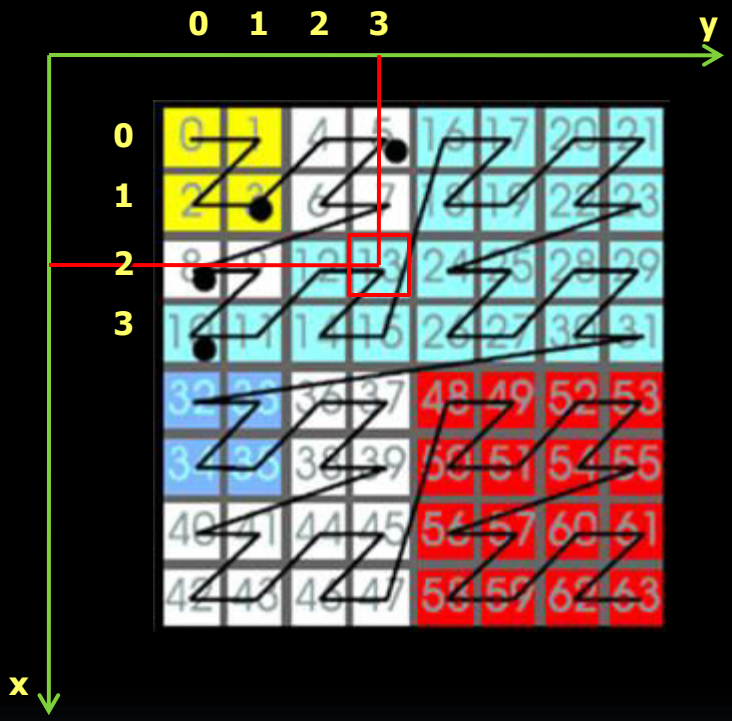
2 op 3 == 13, op == ?

2 → 010

3 → 011

13 ← 0 0 1 1 0 1

Вычисление Z-индекса



2 op 3 == 13, op == ?

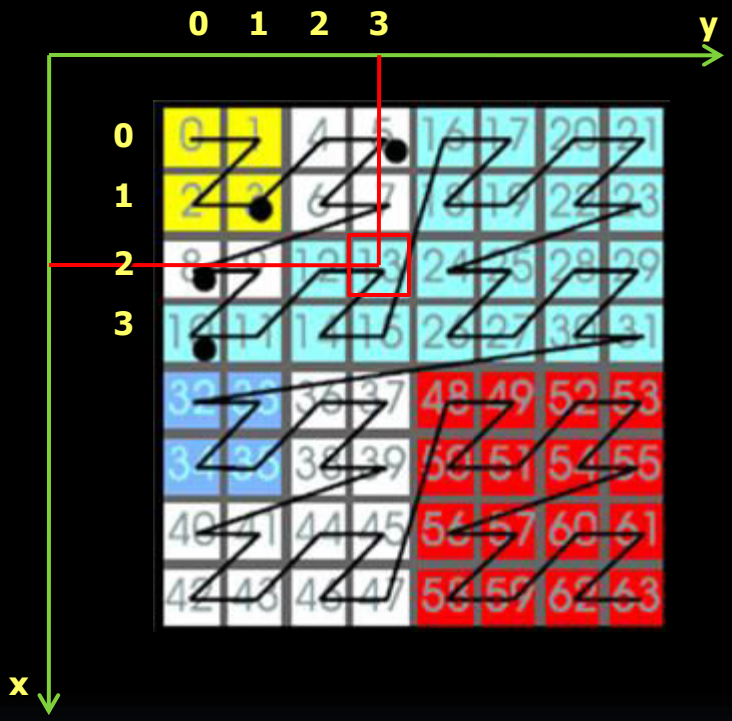
2 → **010**

3 → **011**

13 ← **0 0 1 1 0 1**

```
return   MortonTable256[y >> 8]    << 17 |
         MortonTable256[x >> 8]    << 16 |
         MortonTable256[y & 0xFF]  <<  1 |
         MortonTable256[x & 0xFF];
```

Вычисление Z-индекса



2 op 3 == 13, op == ?

2 → 010

3 → 011

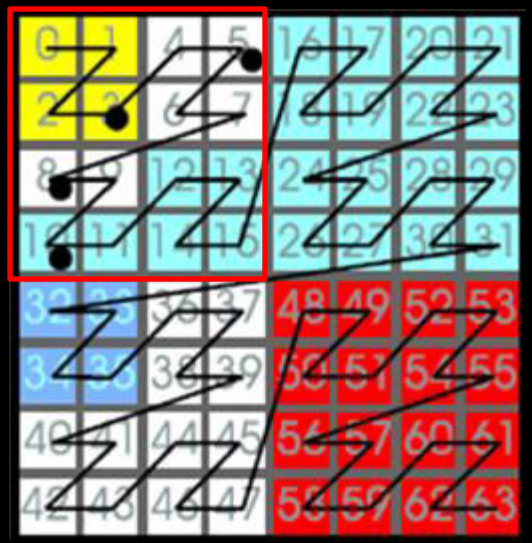
13 ← 0 0 1 1 0 1

```
return   MortonTable256[y >> 8]    << 17 |
         MortonTable256[x >> 8]    << 16 |
         MortonTable256[y & 0xFF]  <<  1 |
         MortonTable256[x & 0xFF];
```

Пора поработать:

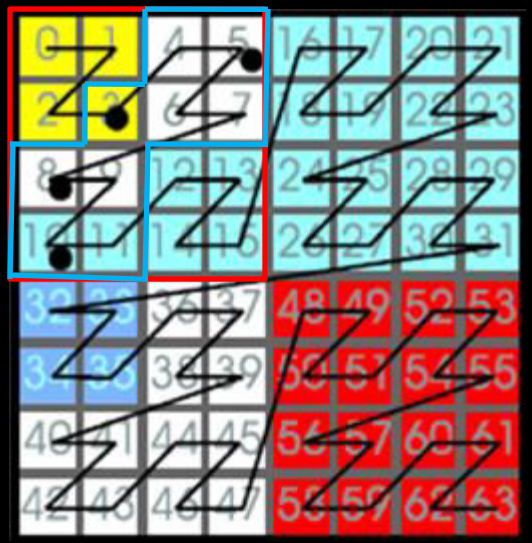
```
class Image;
template <class T> class Matrix;
```

Что такое Z-регионы?



0 – 15
3 – 11
6 – 48

Что такое Z-регионы?



0 – 15

3 – 11

6 – 48

Что такое Z-регионы?

0 – 15

3 – 11

6 – 48

Что такое Z-регионы?

0 – 15
3 – 11
6 – 48

Пусть мы обратились к 6-ому элементу и
Размер кэша равен 43

Тогда зеленая область находится в кэше

Что такое Z-регионы?

0 – 15
3 – 11
6 – 48

Пусть мы обратились к 6-ому элементу и
Размер кэша равен 43

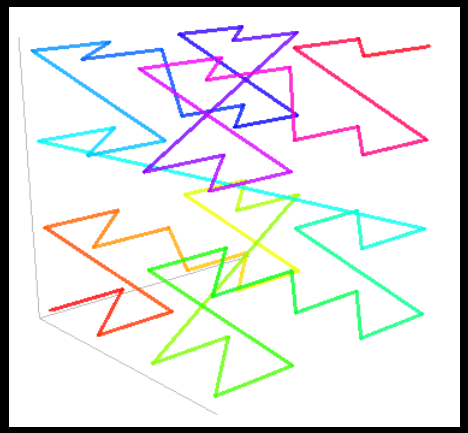
Тогда зеленая область находится в кэше

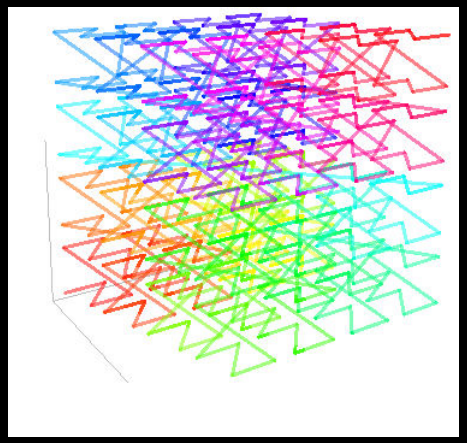
Z-регионы пересекать очень просто – как отрезки, но зачем?

Цель – найти все точки в прямоугольнике, запросов много.

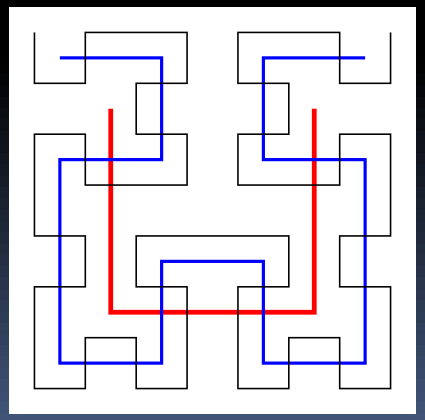
Обработаем сначала те запросы, прямоугольники которых содержаться в Z-регионах, находящихся в кэше

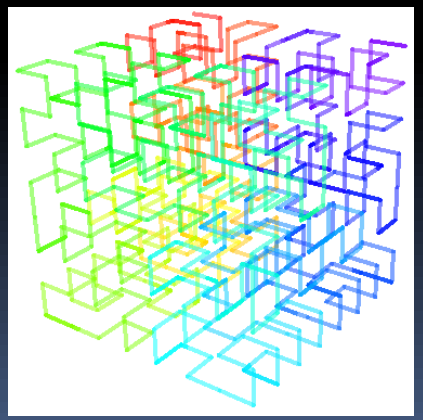
Z-ordering, 3D аналог

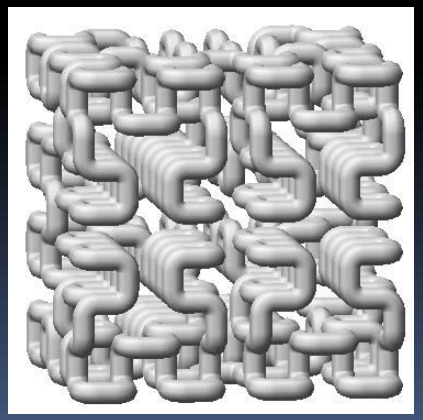




Альтернатива: кривые Гилберта



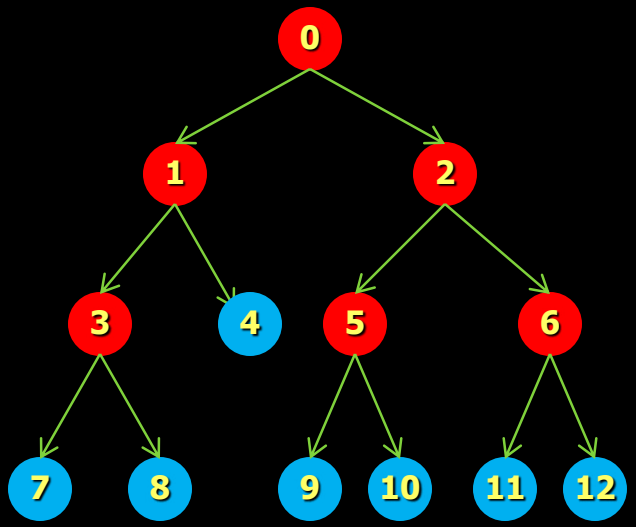




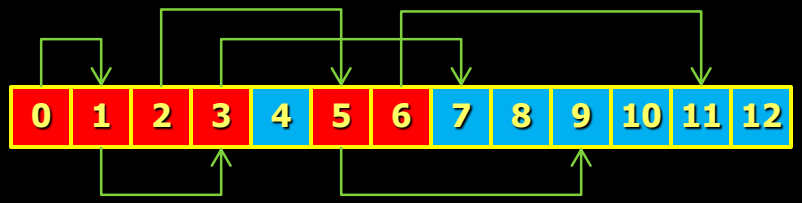
Деревья, оптимизация под кэш

GRAPHICS & MEDIA LAB

Простое расположение



Улучшенное расположение



- Смещения вместо указателей
- Сохраняем только left_offset
- right_offset = left_offset + 1
- Обход дерева “в ширину”

- Улучшенное расположение
  - ✓ ведет к меньшему количеству кэш промахов
  - ✓ Экономит память (4 байта на узел в 32 разрядной системе)

Спасибо за внимание!

GRAPHICS & MEDIA LAB