# 3D графика и трасиране на лъчи v.5.0


http://raytracing-bg.net/

# Тема 9

# Структури за ускорено пресичане
# K-d Trees
# Релефни карти (heightfields)

# Съдържание

- Анонси
- Bounding boxes
- Структури за ускорено пресичане
  - Гридове (равномерни и неравномерни)
  - Осмични дървета (Octrees)
  - Двоични многомерни дървета (K-d trees) и реализацията им
- Релефни карти
  - Структура за ускорено пресичане с релефни карти

# Анонси

- Тест №1 е проверен, резултатите са на сайта на курса

# Bugfixing (от предната лекция)

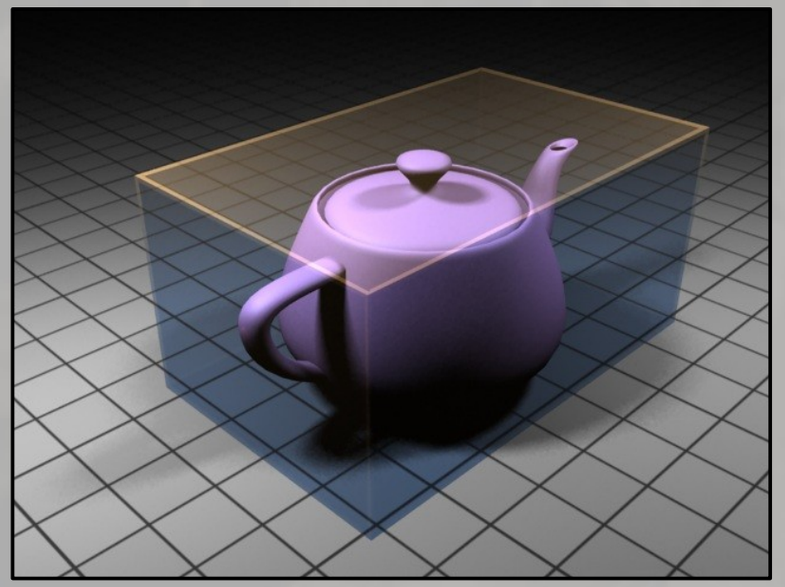
- Bump Maps вече работят
  - Няколко chars разлика
  - BumpTexture::bumpIntensity параметър

# Загражданща геометрия

- Досега, за загражданща геометрия (bounding volume) на триъгълните мрежи, ползвахме сфера
- За загражданща геометрия може да ползваме каквато си поискаме, стига тя:
  - Да наподобява формата на триъгълната мрежа добре
  - Да е лека за пресичане
- Но сферата има няколко недостатъка
  - Ще споменем за тях след малко

# Bounding box

- Bounding box-а представлява паралелепипед, чиито страни са успоредни на координатните оси
  - Дефиниция: всички точки (x, y, z), за които $X_{min} \leq x \leq X_{max}$, $Y_{min} \leq y \leq Y_{max}$ и $Z_{min} \leq z \leq Z_{max}$
  - Проверката за пресичане с лъч е проста
  - Оптималност: тривиално е да се намери най-малкият bounding box около дадена триъгълна мрежа (при сфера не е така)

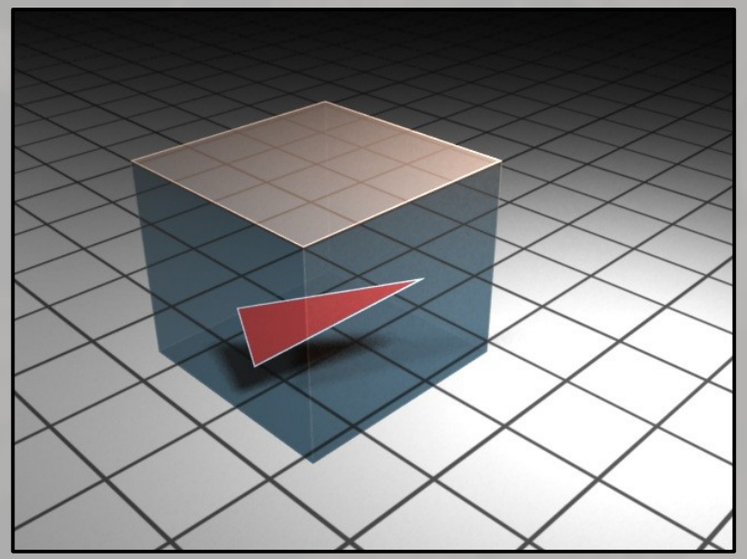


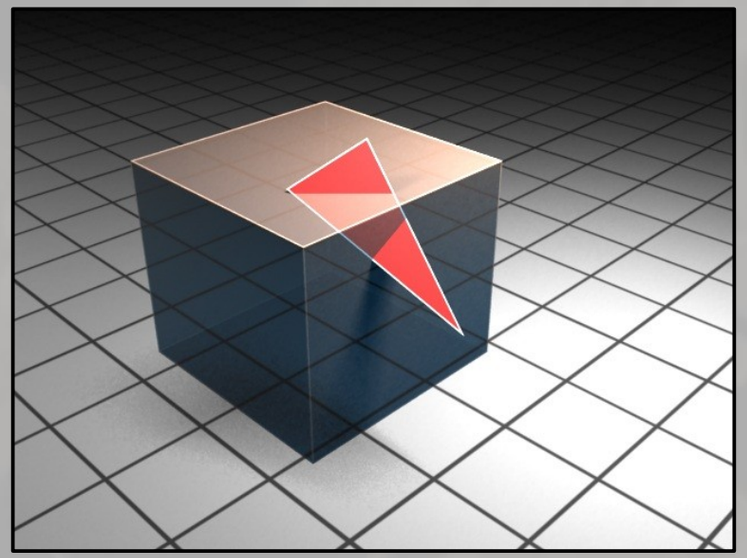
# Bounding Box

- Bounding box може да представим с двойка вектори – vmin и vmax – те са достатъчни да го опишат напълно
- Операции:
  - makeEmpty() - инициализира vmin с +∞, vmax с -∞
  - add(Vector) – добавя точка (потенциално разширява краищата)
  - inside(Vector) – проверява дали точка е в BBox-а
  - testIntersect(Ray) – проверява дали лъч пресича BBox-а
  - intersectTriangle(Triangle) – проверява дали триъгълник и bounding box имат обща част

# Проверка за обща част

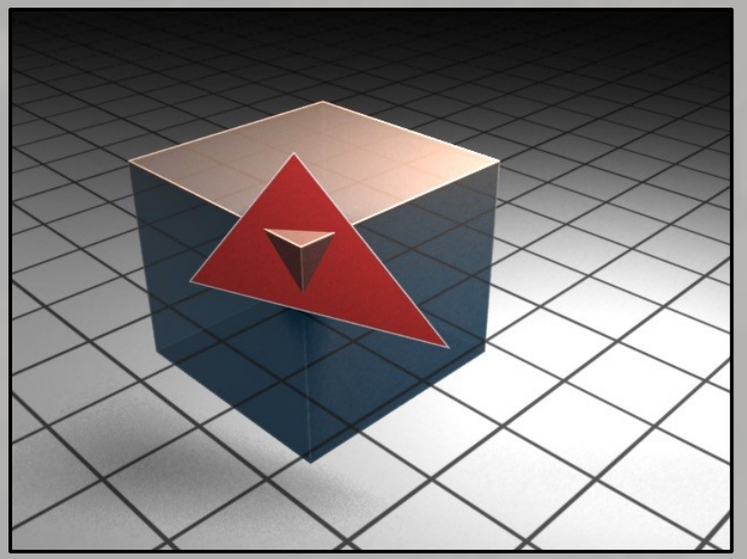
- Случай 1: някой от върховете на триъгълника е вътре в BBox-а
- Случай 2: някой от ръбовете на триъгълника пресича BBox-а.





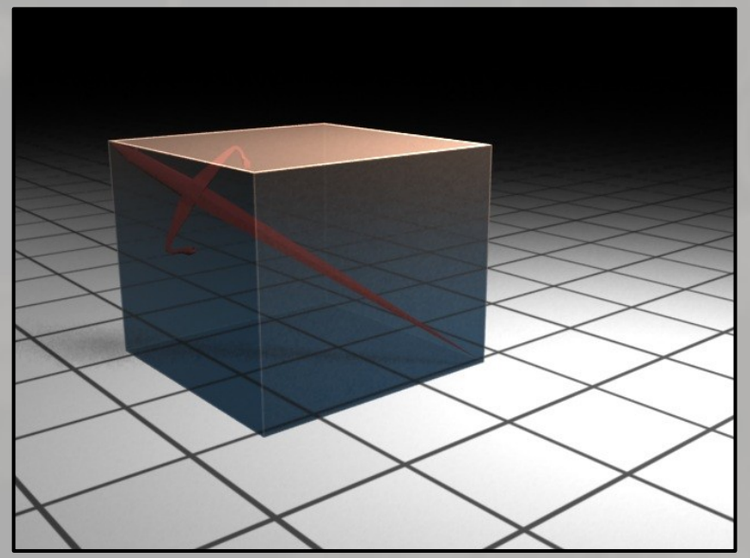
# Проверка за обща част

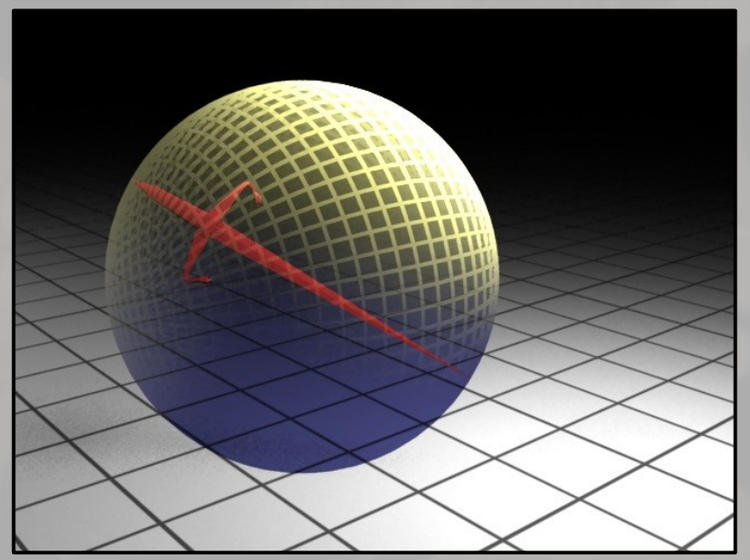
- Случай 3: някой от ръбовете на BBox-а пресича триъгълника



# Заграждане на геометрията

- За да получим bounding box-а на една триъгълна мрежа, просто добавяме (add()) всички върхове
  - Това генерира оптимален BBbox; той няма как да е по-малък
  - Най-лош случай: дълъг, тънък обект, успореден на диагонала (1, 1, 1)
  - Най-лош случай при сфера: произволен дълъг, тънък обект



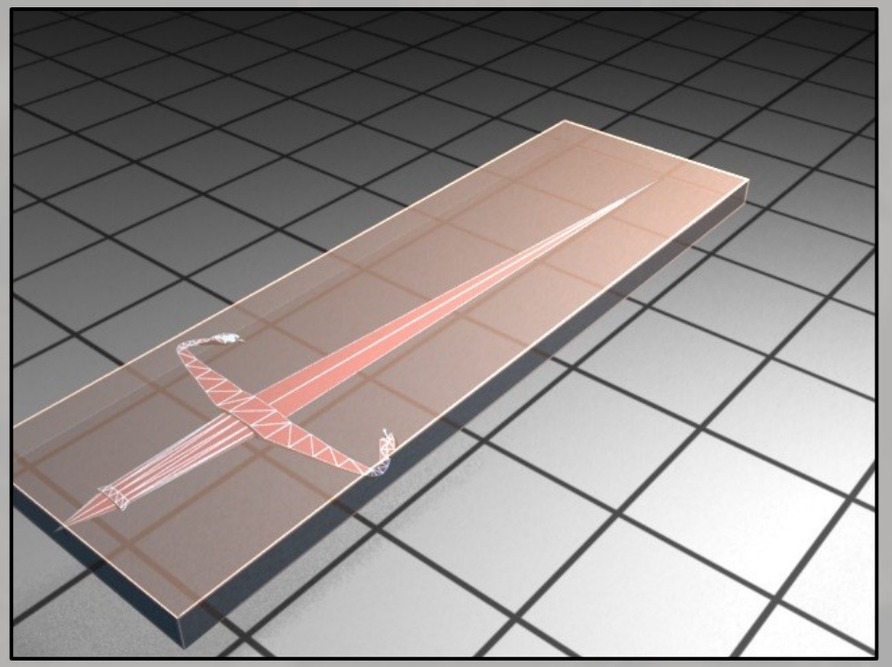


# Структури за ускорено пресичане

- Мотивация
  - Този рендер е отнел 4 часа
  - 4 часа са много. Това дори не е сложна сцена!

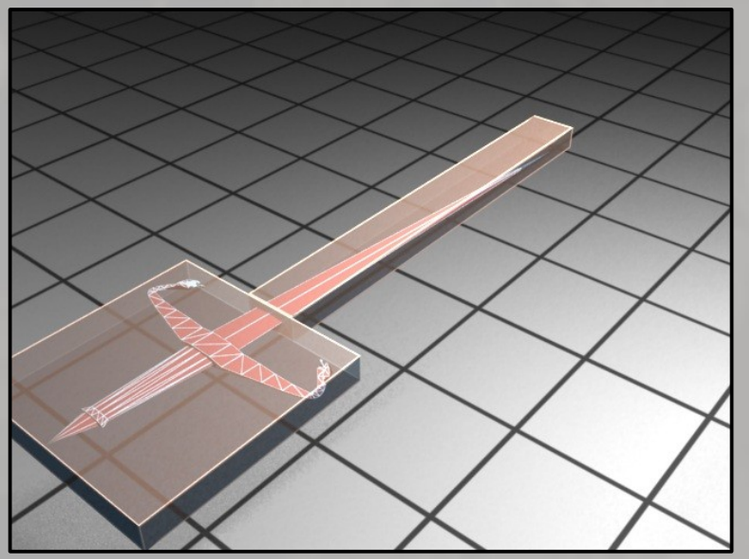
# Структури за ускорено пресичане

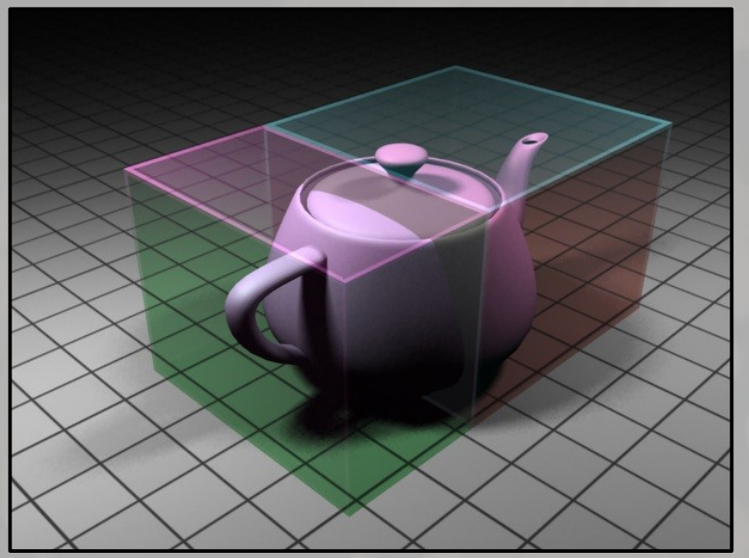
- Пример за неефективност:
  - Лъчите, които пресекат BBox-а, в повечето случаи ще се пресичат напразно с многото триъгълници по дръжката. По-голямата част от лъчите са в областта на острието, а то има само 4 триъгълника
  - Решението: може да разбием пространството на две парчета – дръжка и острие – всяко от които с отделен BBox



# Структури за ускорено пресичане

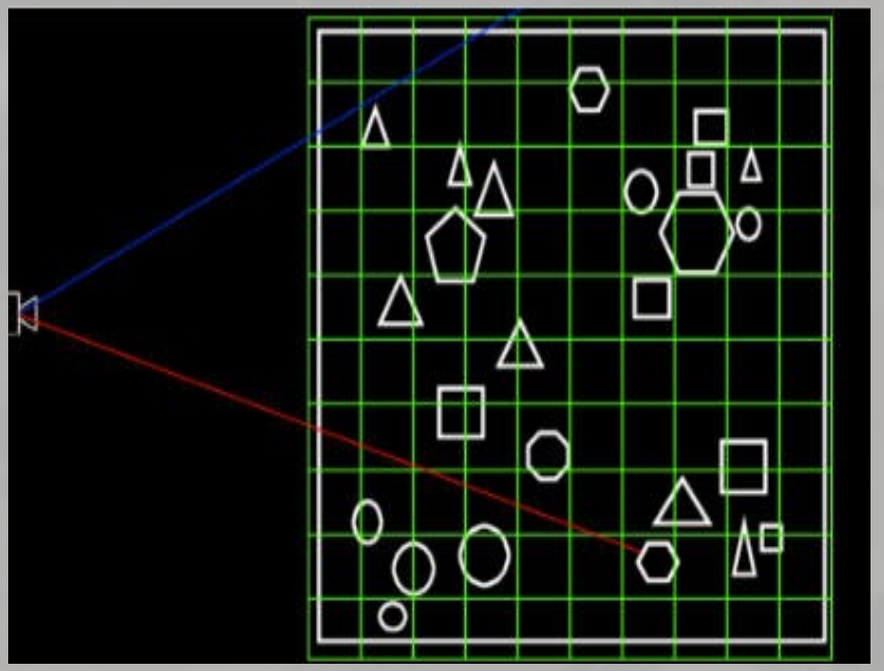
- Тоест, първо ще проверим кой от двата bounding box-а пресича лъча, и в зависимост от това, ще пресечем със съответното подмножество от триъгълници
- На практика ще искаме разделянето на под-обеми да става автоматично





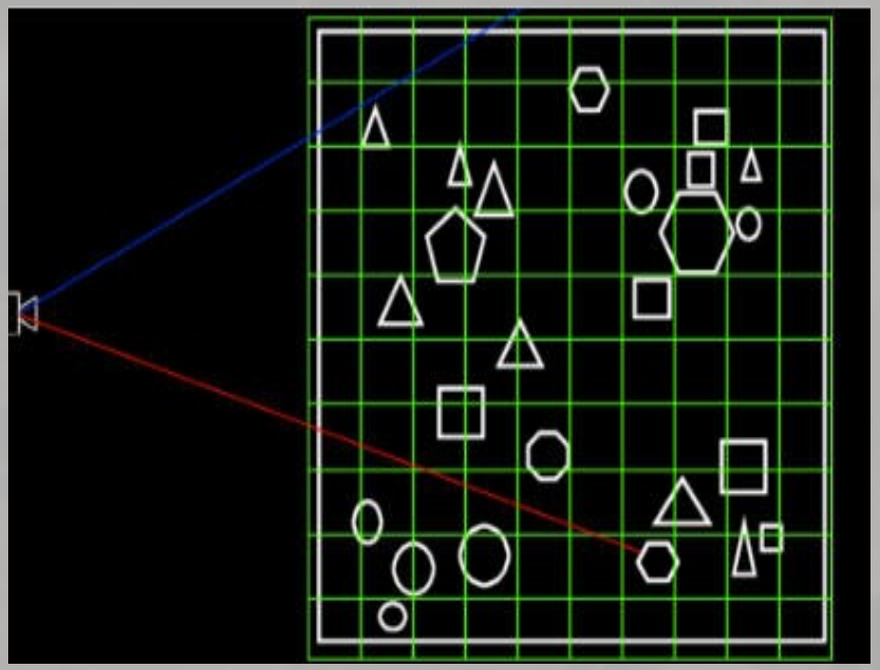
# Равномерен грид

- Равномерен грид
  - При него, разделяме цялото пространство на еднакви клетки, и за всяка клетка определяме кои примитиви (и/или триъгълници) имат общи части с нея
  - При пресичане, посещаваме клетките, през които минава лъча (в реда, определен от посоката на лъча), и пресичаме с прилежащите примитиви



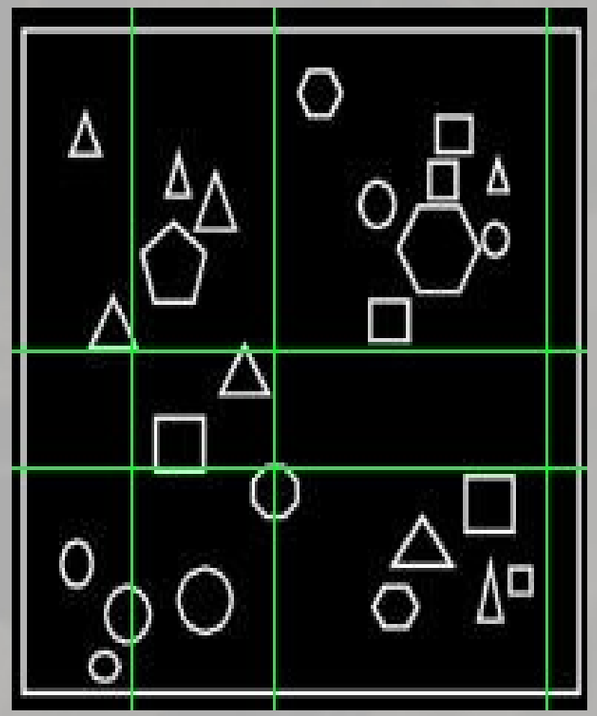
# Равномерен грид

- Така например, синият лъч ще бъде пресечен само с един примитив; червеният ще бъде опитан напразно с 3 примитиви, докато не стигне до истинския, който реално улучва
- Недостатък: в 3D, грида е тримерен, и паметта свършва много бързо ако искаме фин грид



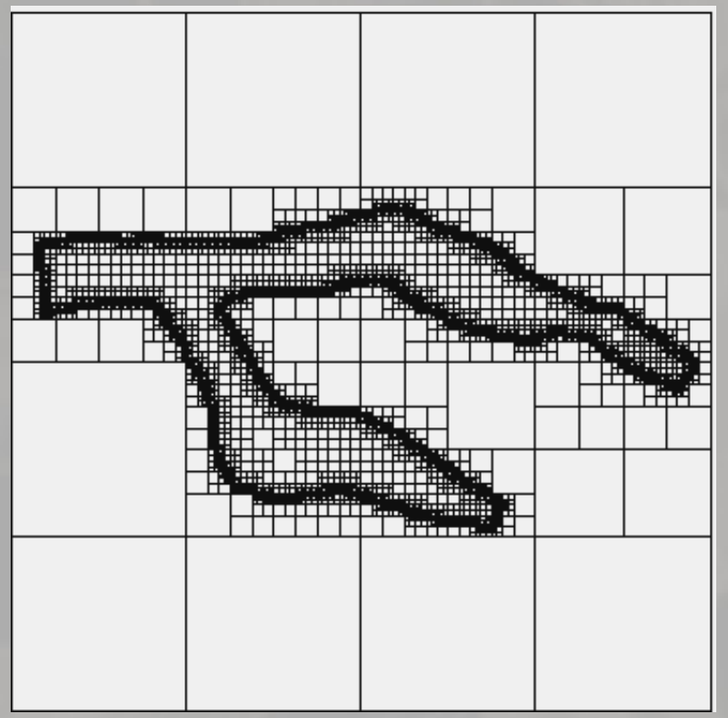
# Неравномерен грид

- Неравномерни гридове
  - Преминаването от клетка в клетка е трудно, но за сметка на това не се харчи памет (например, големите, празни области, се заемат от 1-2 големи клетки, вместо от много мънички, както е при равномерния грид)

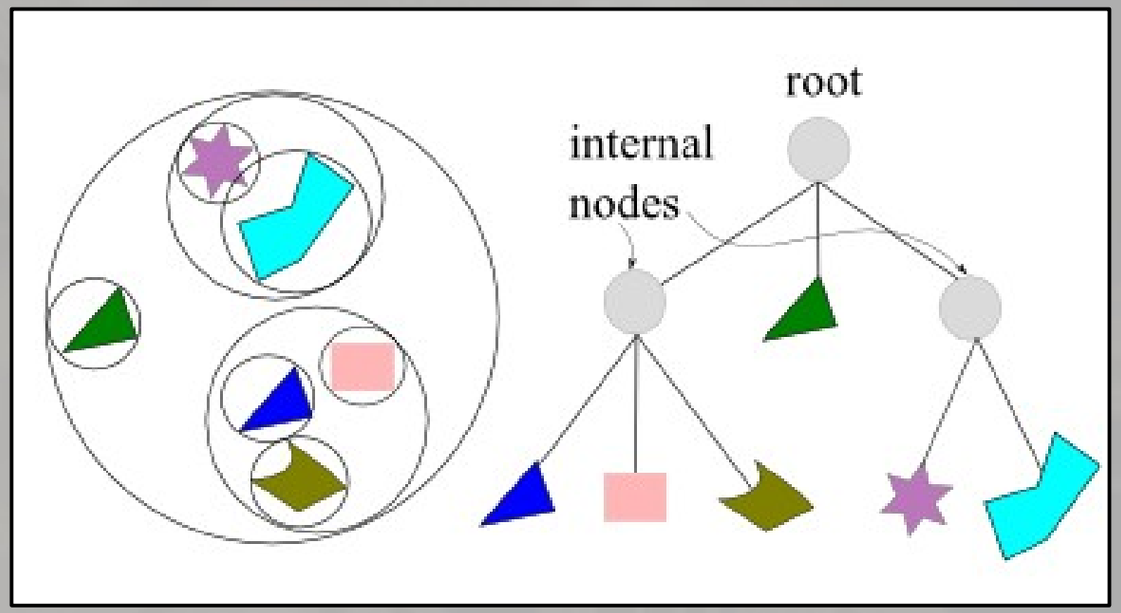


# Осмично дърво (octree)

- Осмични дървета (octrees)
  - Представляват йерархична структура, при която всеки възел от дървото или е листо (съдържа триъгълници), или е разцепен на 8 подвъзела (цепенето е по средата на страните)
  - Итеративна илюстрация: [applet]
  - Недостатък: разделянето по средата невинаги е оптимално (примера със сабята)



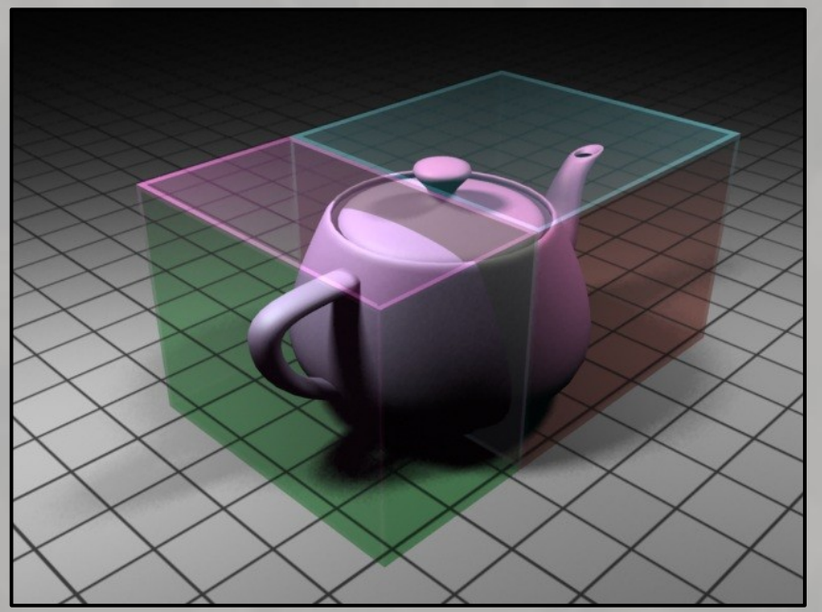
# Bounding Volume Hierarchy (BVH) дървета




- Подходящо за подразделяне на сцената
- Нерегулярно дърво от обеми и под-обеми, където всеки дъщерен възел е в пределите на родителя си

# K-d дървета

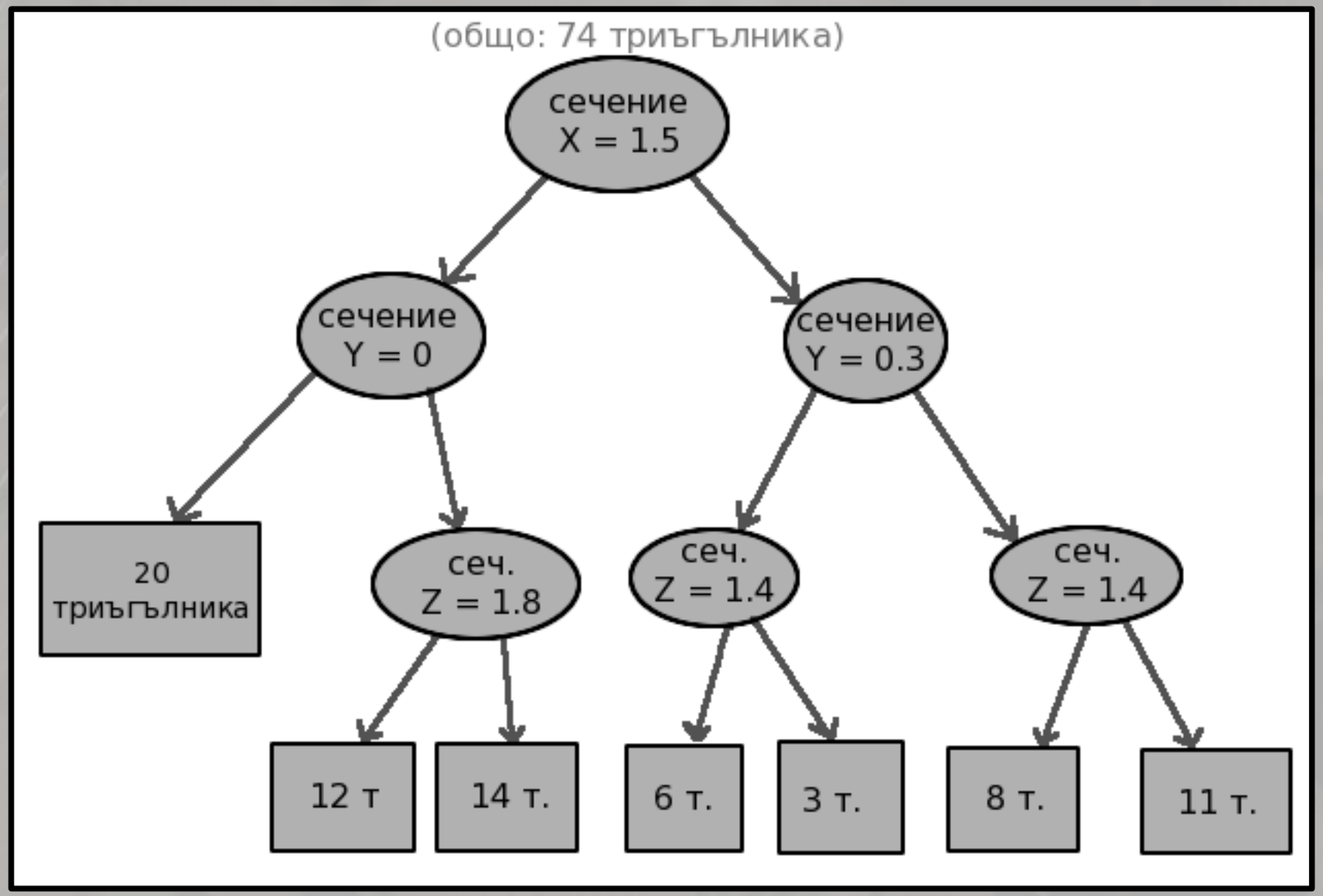
- Генерализация на двоичните дървета за претърсване в произволномерно пространство
- Всеки възел в дървото или е листо (съдържа списък с триъгълници), или има две поддървета, като те обхващат обемите, получени от BBox-а на възела след разцепване по някое направление в някаква позиция



# K-d дървета

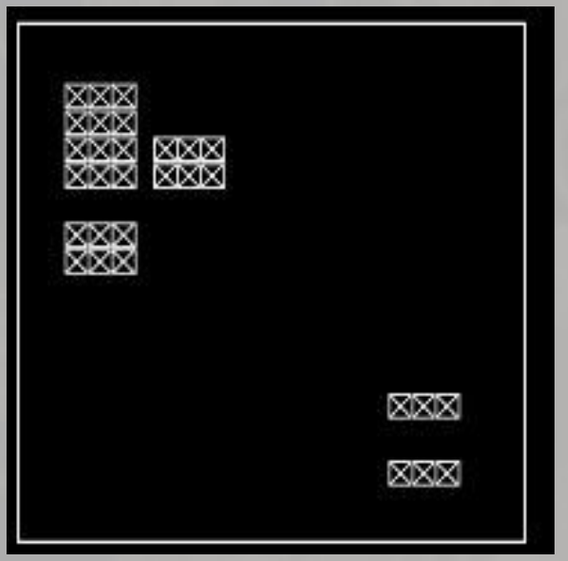
- Дървото представлява йерархична структура на пространството; всеки възел от дървото представлява някаква част от пространството, и „знае“ кои триъгълници влизат в нея
  - Ако областта е голяма, дефинираме разделителна равнина, която сече пространството на две – това са двата подвъзела – в които остават по по-малко триъгълници
  - Ако областта е достатъчно малка (т.е., съдържа малко триъгълници), може просто да ги опишем всички.

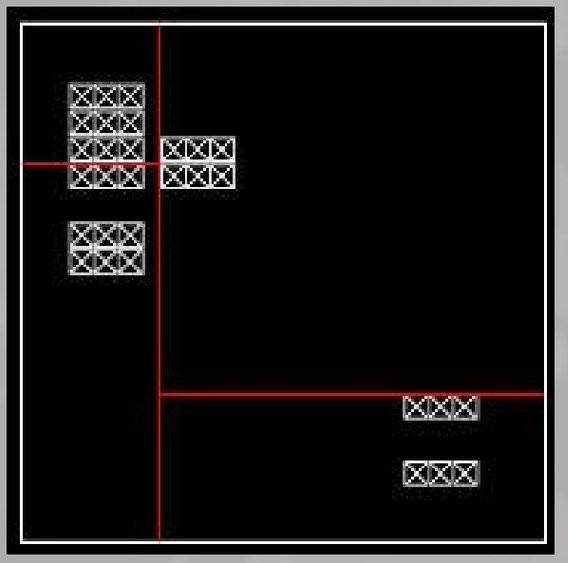
# K-d дървета


(общо: 74 триъгълника)
сечение
X = 1.5
сечение
Y = 0
сечение
Y = 0.3
20
триъгълника
сеч.
Z = 1.8
сеч.
Z = 1.4
сеч.
Z = 1.4
12 т
14 т.
6 т.
3 т.
8 т.
11 т.

# K-d дървета

- На всяка стъпка, сечението е само по една от осите
  - Т.е., за да наподобим осмичните дървета, трябват ни три нива разцепвания (по X, Y и Z), но имаме много по-голяма свобода, защото можем да избираме позициите на деление както си искаме
  - Една от възможностите е, да търсим такава разделяща равнина, че двете половини да са с приблизително еднакъв брой триъгълници (наполовина)



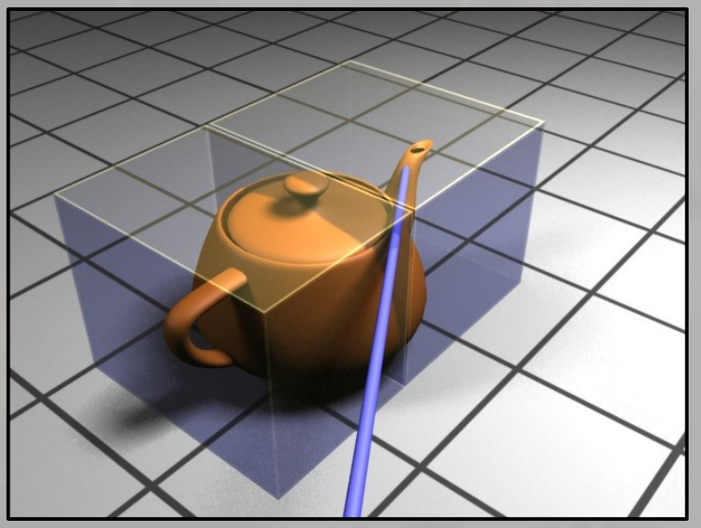


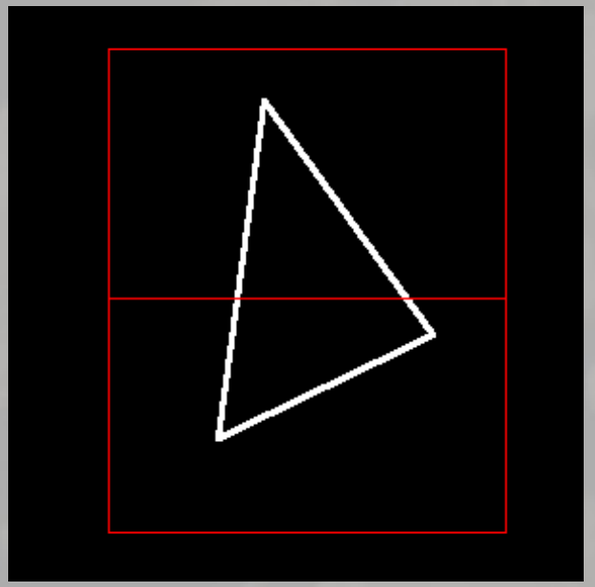
# Пресичане с K-d дърво

- Търсенето на пресечната точка започва от корена на дървото (целия обект) и на всяка стъпка „влизаме“ в този подвъзел, който лъчът уцелва. Броят на триъгълниците спада приблизително наполовина с всяка стъпка, така че след приблизително $\log_2(N)$ стъпки, ще стигнем до листо (идеално: с единствен триъгълник; на практика – с (малък) списък от триъгълници)
  - Т.е., реализирахме логаритмична сложност на търсенето!

# Пресичане с K-d дърво

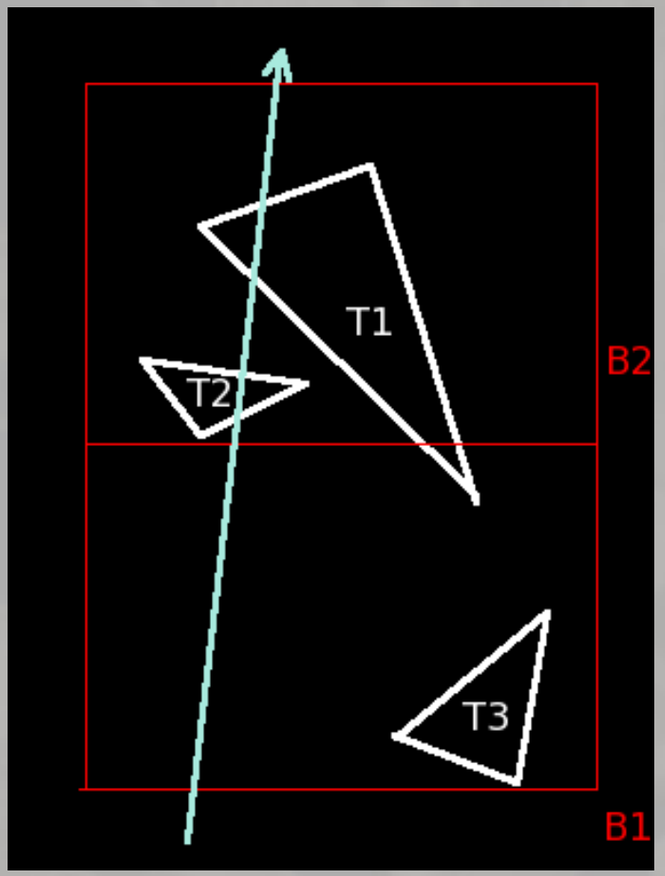
- Няколко проблема и частни случая:
  - Лъчът може да пресича и двата подвъзела
    - Можем да пресечем първо с по-близкия, и ако не намерим пресечната точка там, с по-далечния
  - Триъгълник може да принадлежи и на двете подпространства
    - Вариант 1: цепим триъгълника по границата
    - Вариант 2: добавяме го и в двата подвъзела





# Пресичане с K-d дърво

- При вариант 2, има един частен случай:
  - Търсенето ще влезе първо в B1. Тъй като T1 принадлежи едновременно на B1 и B1, то лъчът ще бъде пресечен с T1, и търсенето ще приключи дотук
  - Но имаме по-близка пресечна точка (с T2), само че я пропускаме, защото T2 се намира изцяло в по-далечния възел
  - Решението: ще изискваме пресечната точка да бъде вътре във възела, който текущо разглеждаме (B1). Само тогава ще считаме, че сме намерили пресечната точка



# Построяване на K-d дървото

```
Function build(triangles, bbox, depth):
    if triangles.size() < MAX_TRIANGLES_PER_LEAF or depth > MAX_DEPTH:
        return LeafNode(triangles)
    else:
        splitAxis = depth % 3
        sp = findOptimalSplitPlane(triangles, bbox, splitAxis)
        leftBBox = bbox.split(sp, LEFT)
        rightBBox = bbox.split(sp, RIGHT)
        leftTriangles = intersect(triangles, leftBBox)
        rightTriangles = intersect(triangles, rightBBox)
        curNode = BinaryNode(splitAxis, sp)
        curNode.left = build(leftTriangles, leftBBox, depth + 1)
        curNode.right = build(rightTriangles, rightBBox, depth + 1)
        return curNode
```

# Построяване на K-d дървото

- Намирането на оптималната разделяща равнина може да стане чрез най-различни евристики
  - Разделянето на триъгълниците на равни части не е най-добрата, но може да се намери лесно чрез двоично търсене
- В случая редуваме цепене по X, Y, Z, X, Y, Z ..., но може да прилагаме и друга схема, ако е подходящо
  - Например, цепим най-дългата страна на BBox-а
- Константите MAX_TRIANGLES_PER_LEAF и MAX_DEPTH могат да се тунинговат подходящо за конкретната геометрия. Примерни стойности – 20, 64.

# Пресичане с K-d дървото

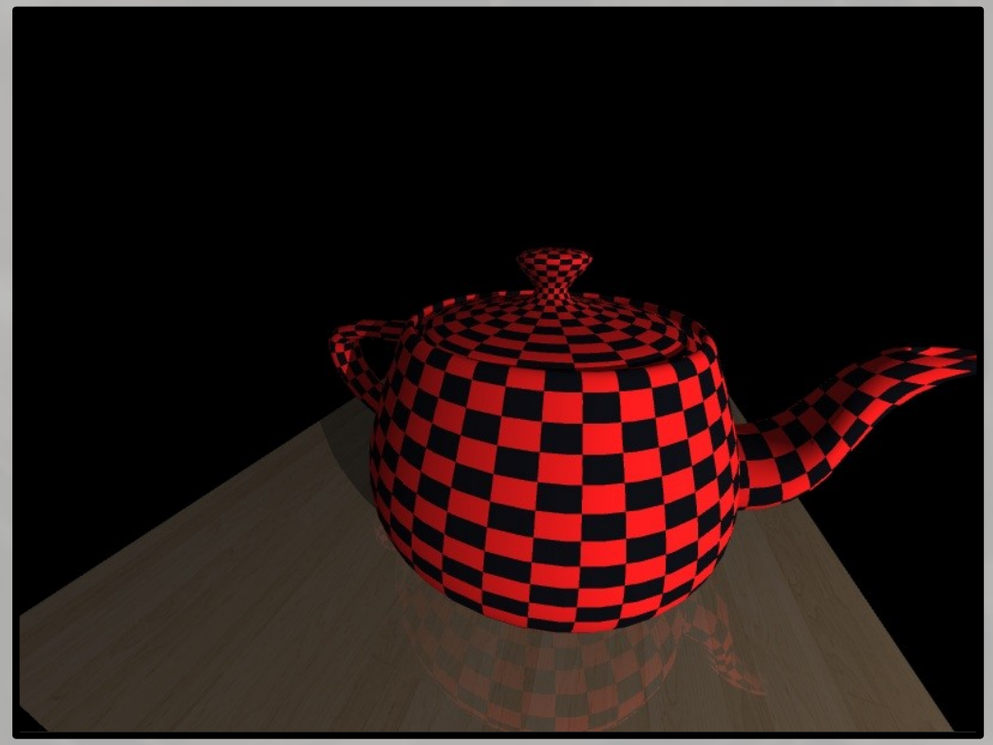
```
Function intersect(ray, info, node, bbox):
    if isLeaf(node):
        if node.triangles.intersect(ray, info) and bbox.inside(info.p):
            return info.dist
    else:
        for subnode in sorted(node.children, ray):
            subnodeBBox = bbox.split(subnode)
            if subnodeBBox.intersects(ray):
                dist = intersect(ray, info, subnode, subnodeBBox)
                if dist < ∞:
                    return dist
        return ∞
```

# Разискване

- Какво става със сложността, ако се налага да пресичаме и с двата подвъзела?
  - Тези случаи са рядкост. В средния случай, обхождането на поддърво е логаритмична операция
- Ами триъгълниците, които се повтарят и в двата възела?
  - Те вдигат височината на дървото, но не с много (например, за чайник с ~10000 триъгълника, средната дълбочина на дървото е 16).

# K-d дърво - резултати

- Идентична сцена с груб и с фин чайник
  - kdtree_test.qdmg
  - gcc 4.8, Haswell @3.3 GHz
- Резултати
  (M = брой пиксели; N = брой триъгълници)



| | **Naïve - O(N*M)** | **Tree build - O(N*logN)** | **Рендер - O(M*logN)** | **Общо** |
|---|---|---|---|---|
| Груб чайник (992 триъгълника) | 20.0s | 0.01s | 2.03s | 2.04s |
| Фин чайник (9120 триъгълника) | 166.4s | 0.09s | 2.54s | 2.63s |

# Резултати

Time: ~13 минути (AA, 6 refl, 6 refr, 1024x640)

# K-d дърво (заключение)

- Въпреки голямото подобрение от K-d дървото, с използването му трябва да се внимава
  - Потенциално заема огромно количество памет
  - K-d дърво за малки (<50 триъгълника) обекти е безсмислено
  - Строежа на дървото отнема време (особено при по-сложните алгоритми за оптимален избор на разделящата равнина)
    - Дървото се представя най-добре когато се строи веднъж, а после се ползва за пресичане с огромен брой лъчи
    - Трябва да се внимава времето за строеж да не надхвърля ползата от ускорението, идващо от използването му
      - Параметрите за строежа е добре да са на разположение на потребителя

# Ускоряване на цялата сцена

- В момента - 1 триъгълна мрежа = 1 K-d дърво
  - Пак сме бавни при множество малки геометрии
- Ако всички геометрии са триъгълни мрежи, може да се ползва глобално K-d дърво за цялата сцена
  - O(logN) за пресичане на лъч със сцената
- BVH дървета за общия случай