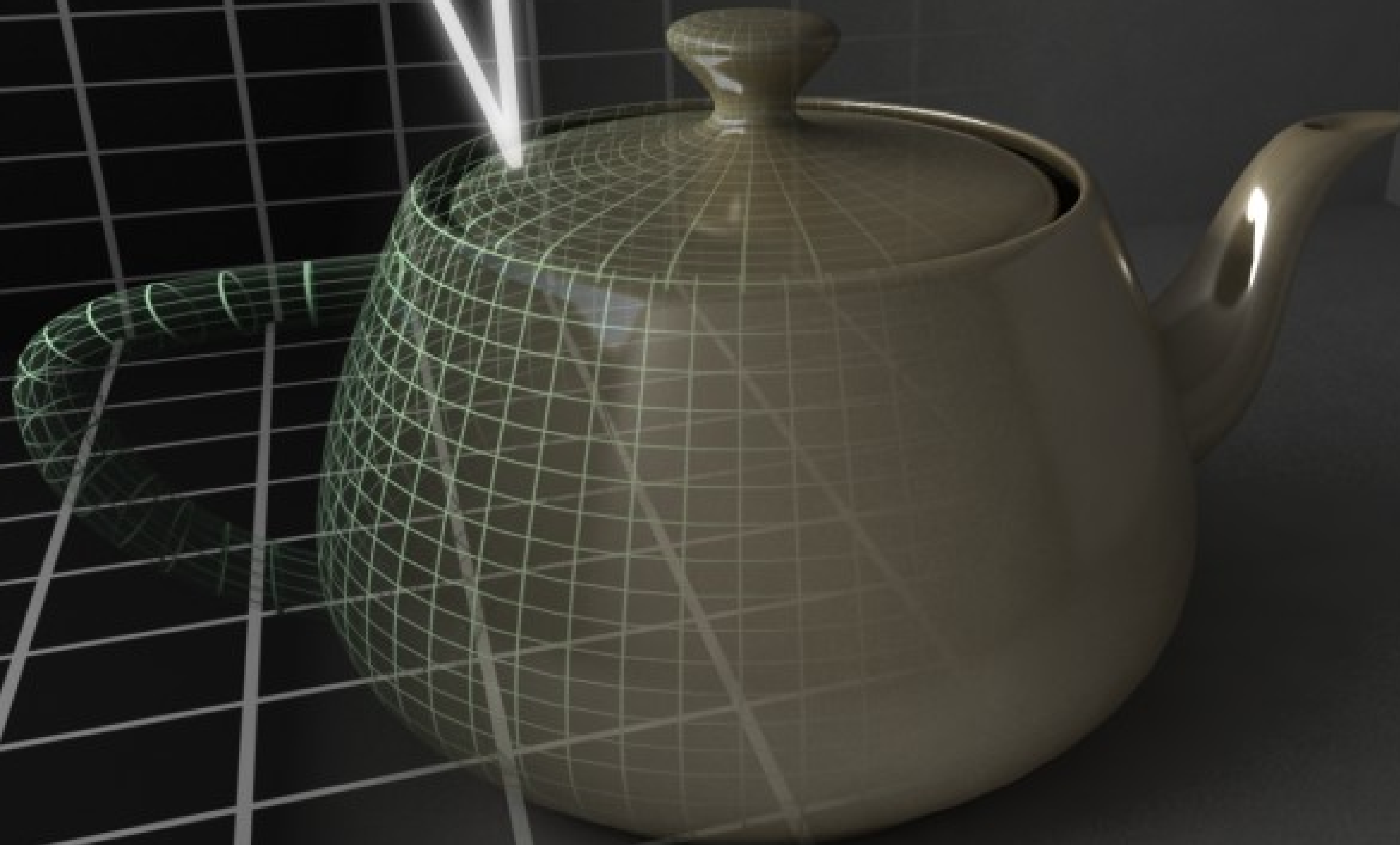
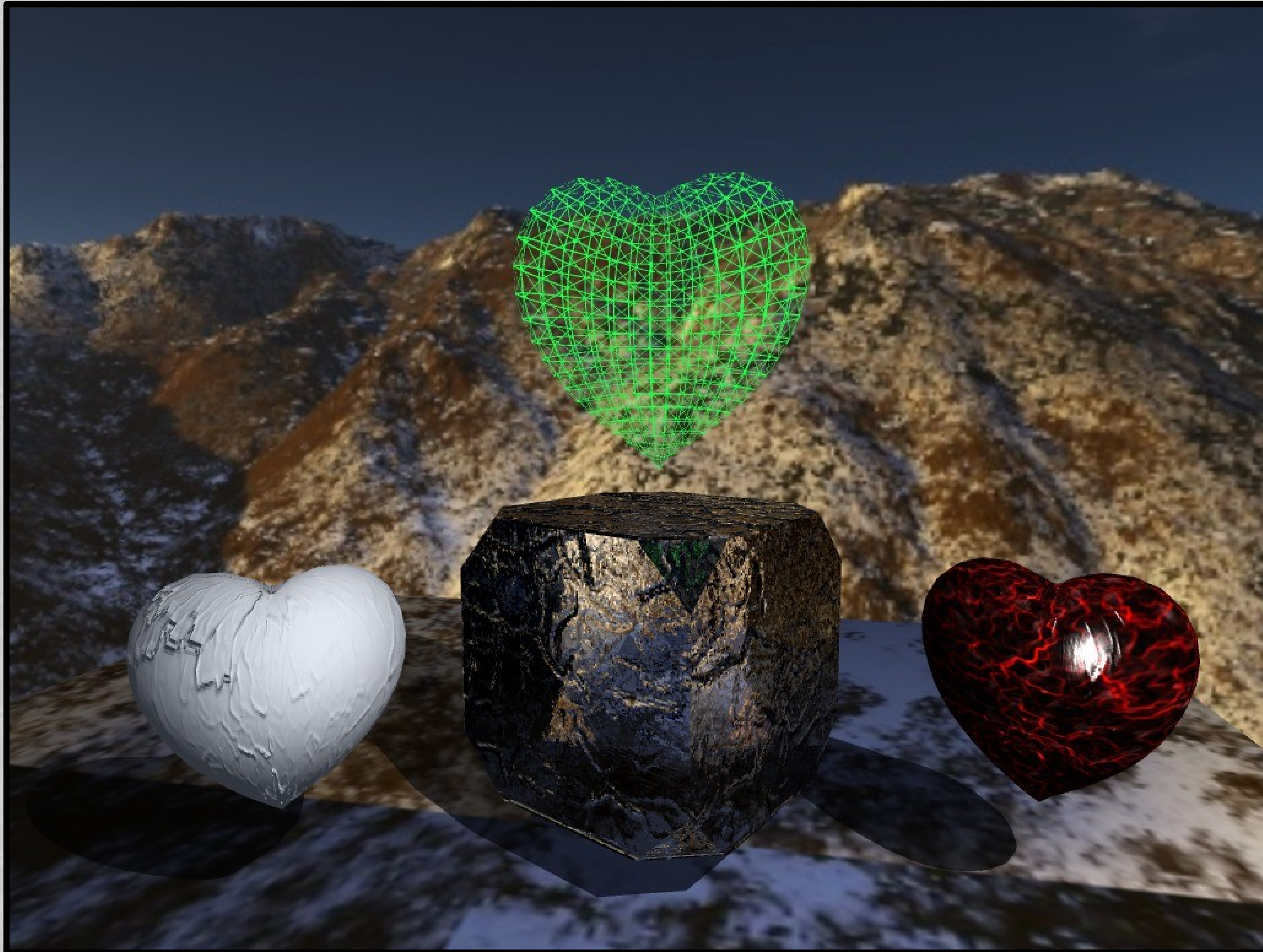
3D графика и трасиране на лъчи



http://raytracing-bg.net/

BofH



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

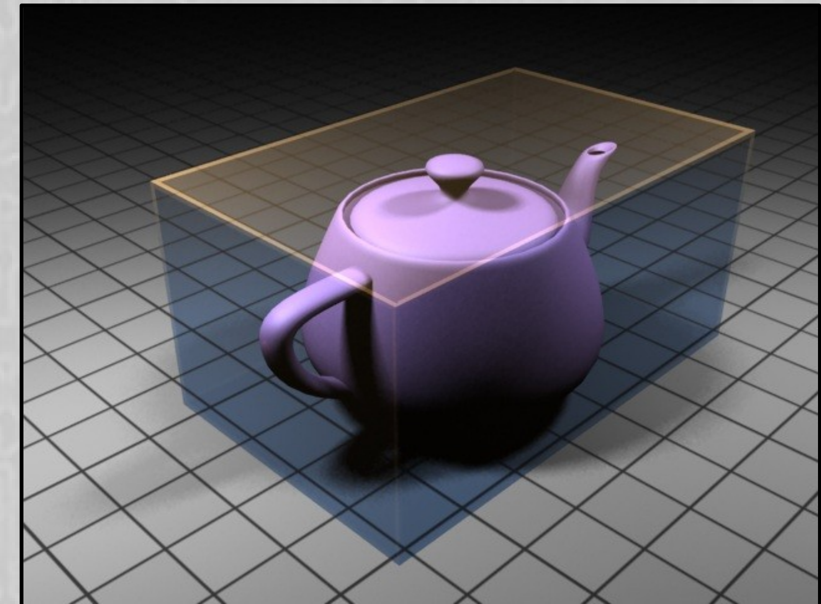
- Тест №1 е проверен, резултатите са на сайта на курса
  - Почти 13т. среден резултат
- No more spoilers in homework descriptions
  - Пълни указания (като сегашните) ще даваме при поискване
  - Стимулираме ви да откривате сами кое как става
- Семинар - „Real Time Ray Tracing“ с гости от университета в Saarbrücken
  - 14ти Декември, 15:00, 325/ФМИ

Заграждаща геометрия

- Досега, за заграждаща геометрия (bounding volume) на триъгълните мрежи, ползвахме сфера
- За заграждаща 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)

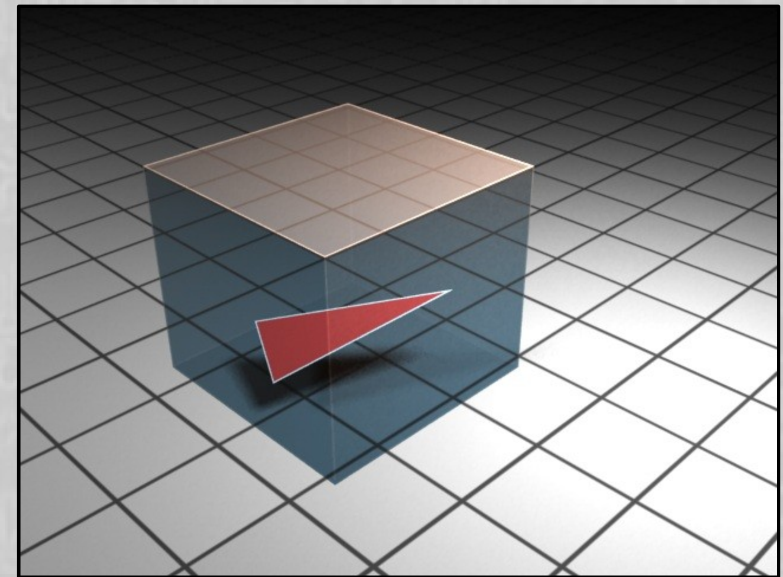


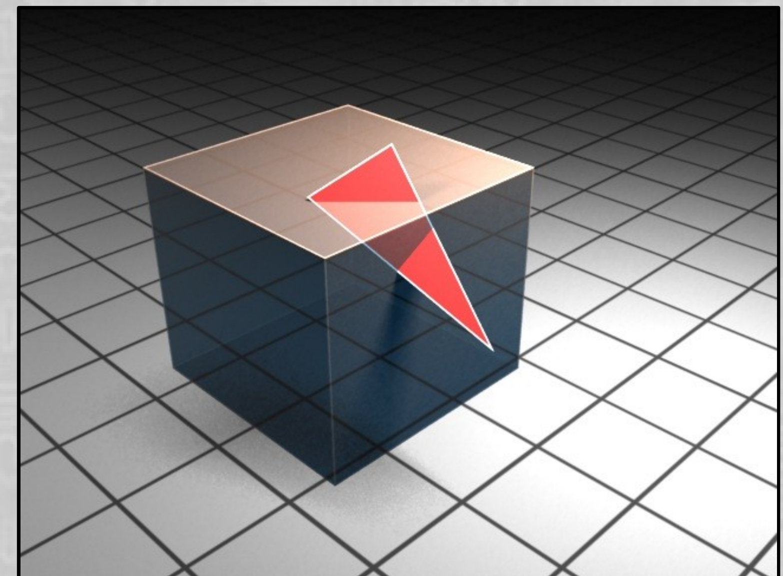
Bounding Box

- 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

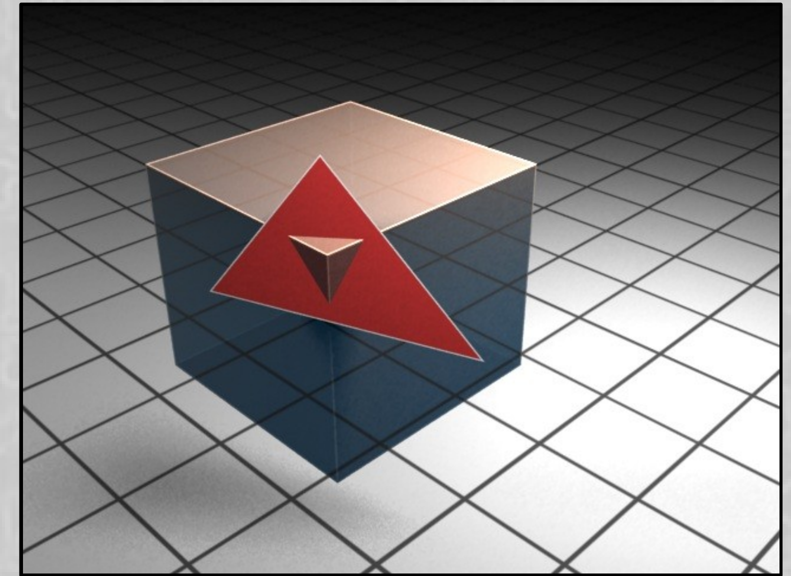
- Случай 1: някой от върховете на триъгълника е вътре в BBox-а
- Случай 2: някой от ръбовете на триъгълника пресича BBox-а.





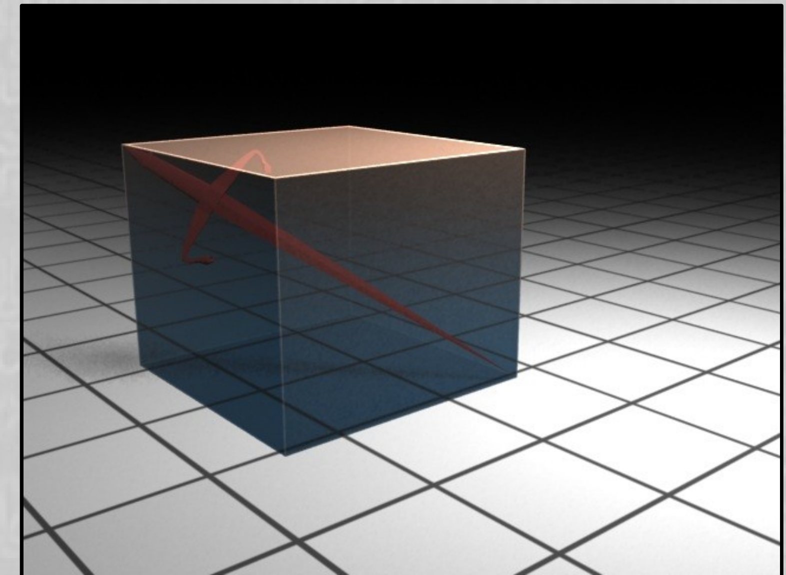
Проверка за обща част

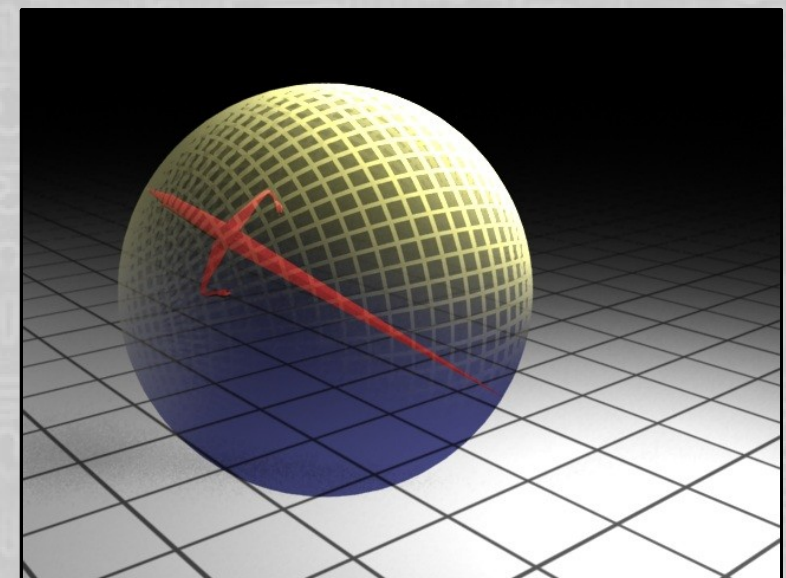
- Случай 3: някой от ръбовете на BBox-а пресича триъгълника



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



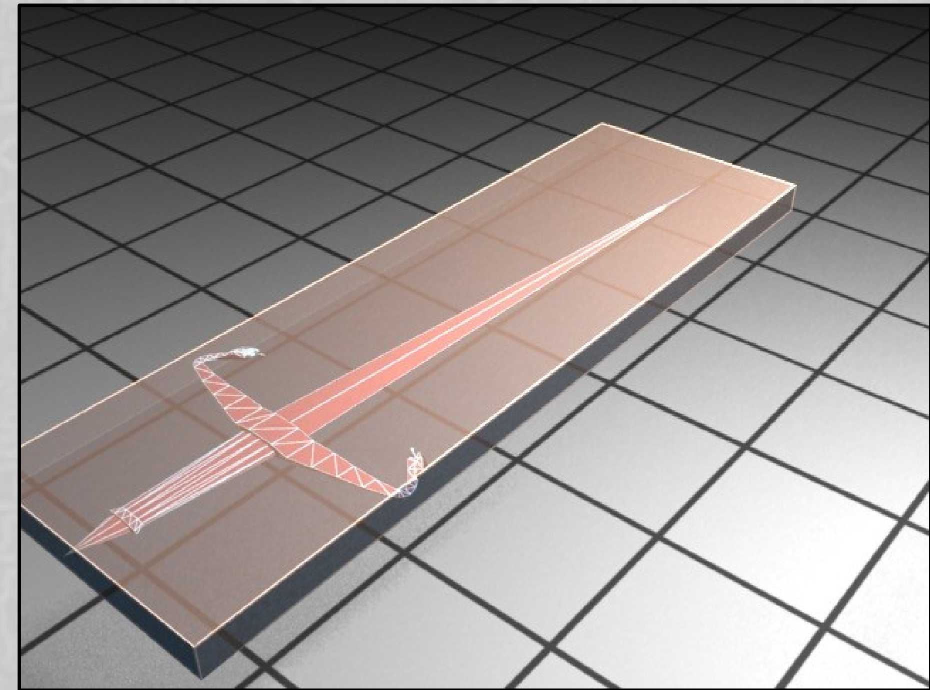


Структури за ускорено пресичане

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

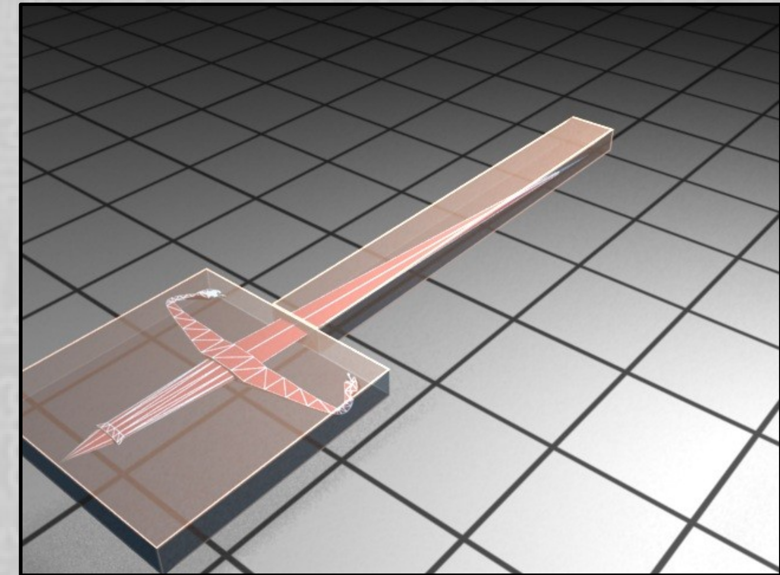
Структури за ускорено пресичане

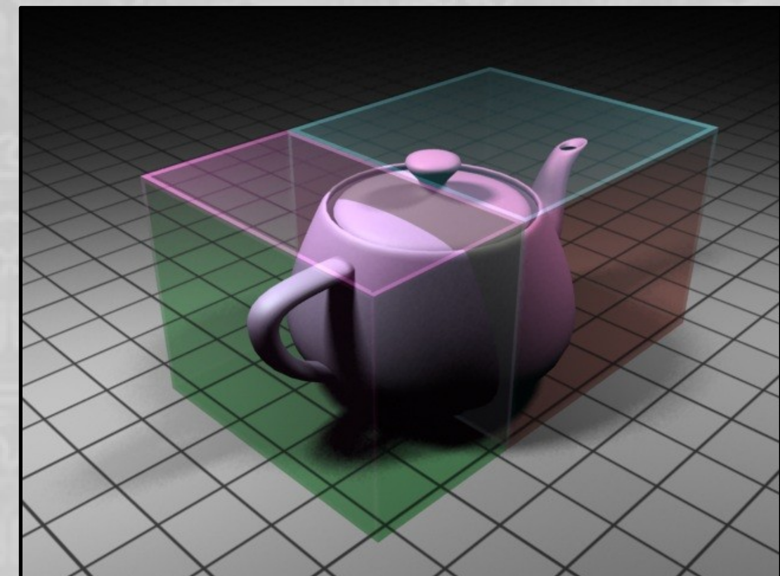
- Пример за неефективност:
  - Лъчите, които пресекат BBox-а, в повечето случаи ще се пресичат напразно с многото триъгълници по дръжката. По-голямата част от лъчите са в областта на острието, а то има само 4 триъгълника
  - Решението: може да разбием пространството на две парчета – дръжка и острие –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



Структури за ускорено пресичане

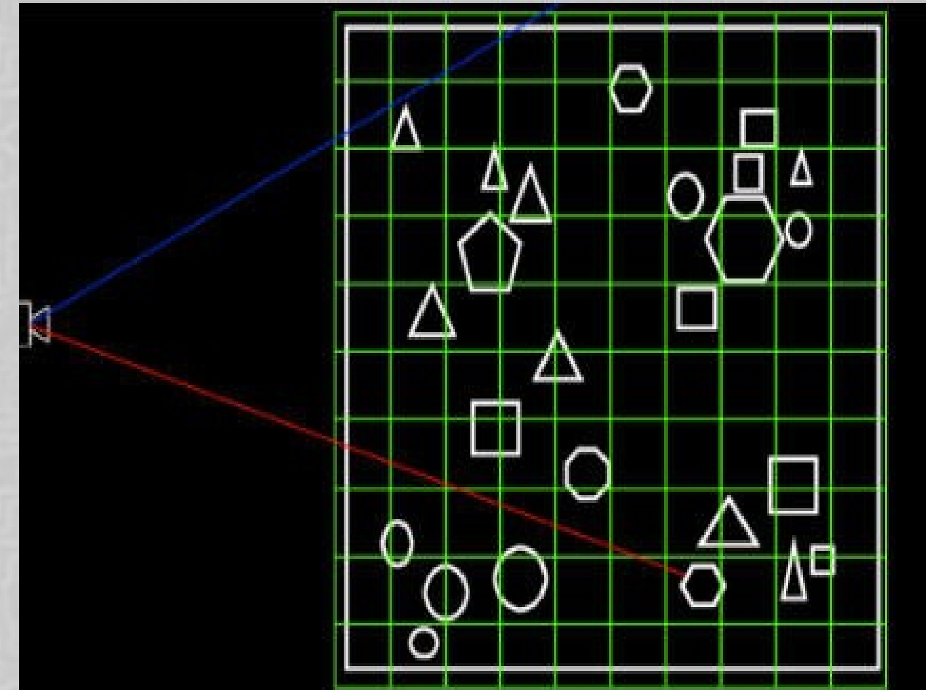
- Тоест, първо ще проверим кой от двата bounding box-а пресича лъча, и в зависимост от това, ще пресечем със съответното подмножество от триъгълници
- На практика ще искаме разделянето на под-обеми да става автоматично





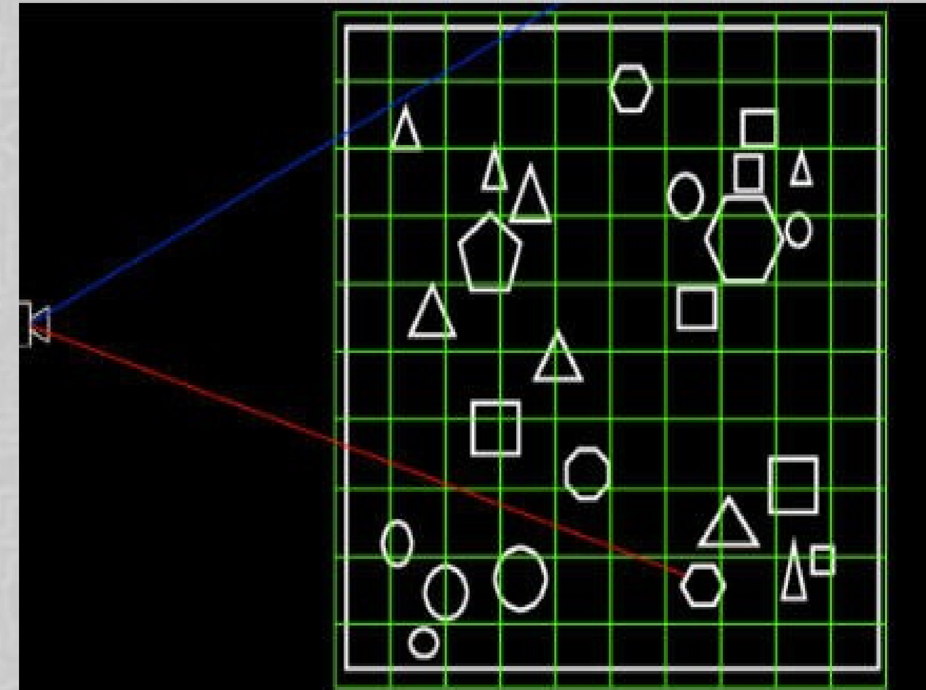
Равномерен грид

- 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



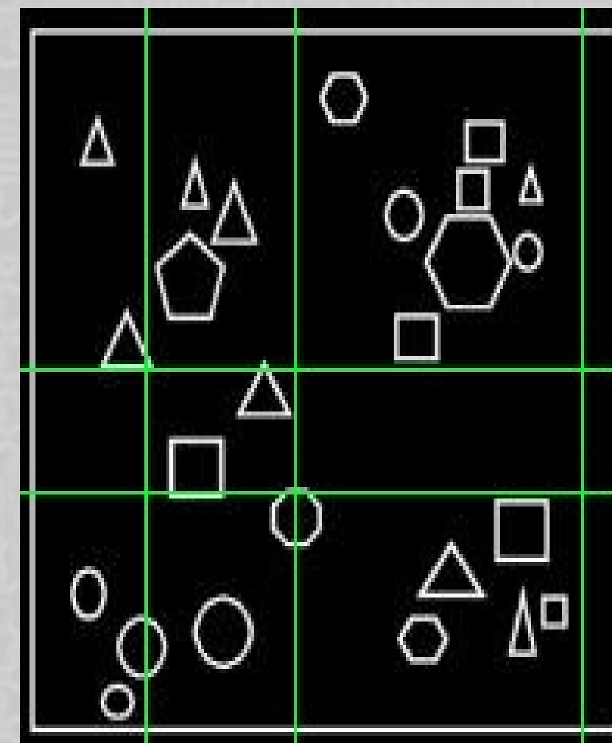
Равномерен грид

- 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
- Примерна (и чиста) реализация: [Jacco.Bikker@DevMaster]



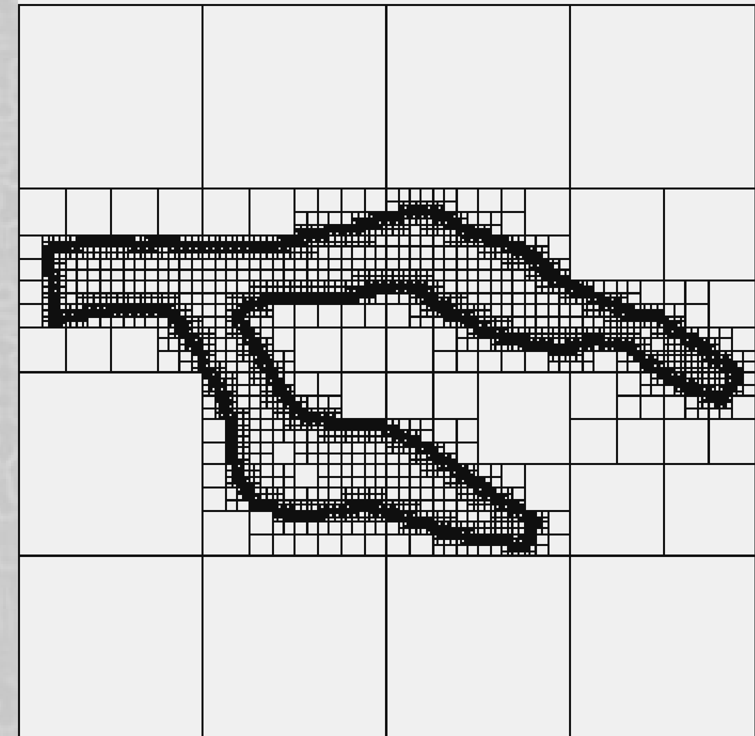
Неравномерен грид

- 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)



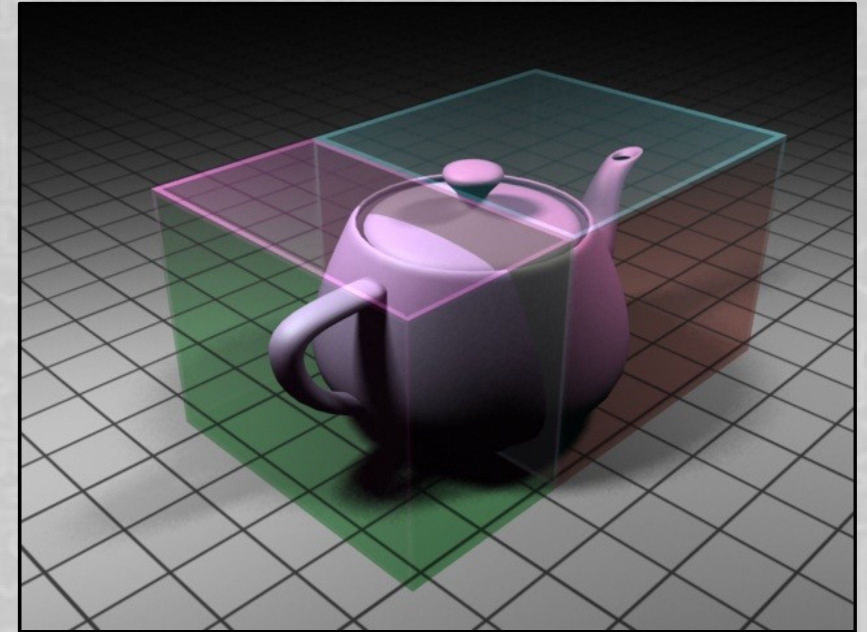
Осмично дърво (octree)

- 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)



K-d дървета

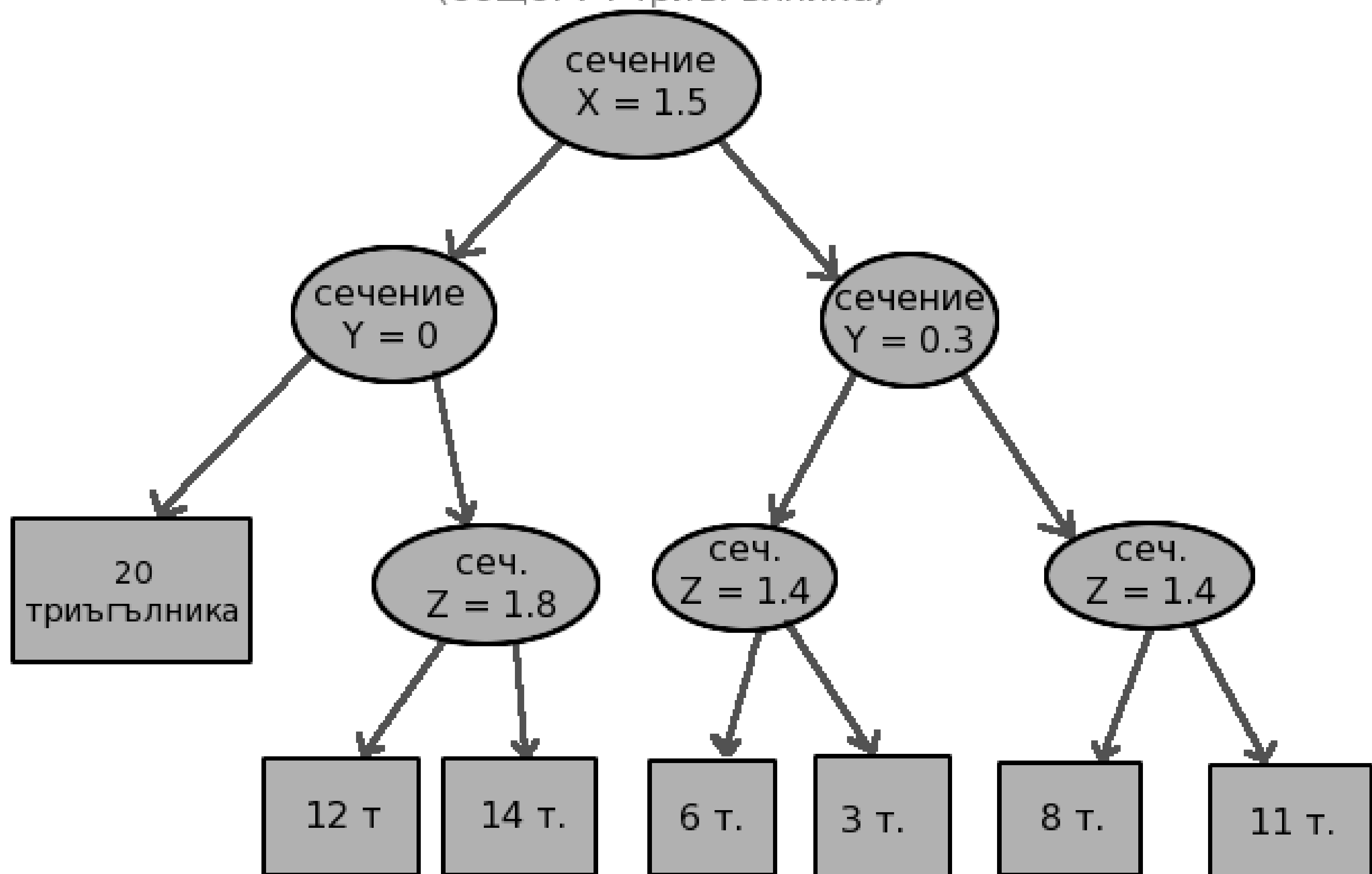
- Генерализация на двоичните дървета за претърсване в произволномерно пространство
- Всеки възел в дървото или е листо (съдържа списък с триъгълници), или има две поддървета, като те обхващат обемите, получени от BBox-а на възела след разцепване по някое направление в някаква позиция



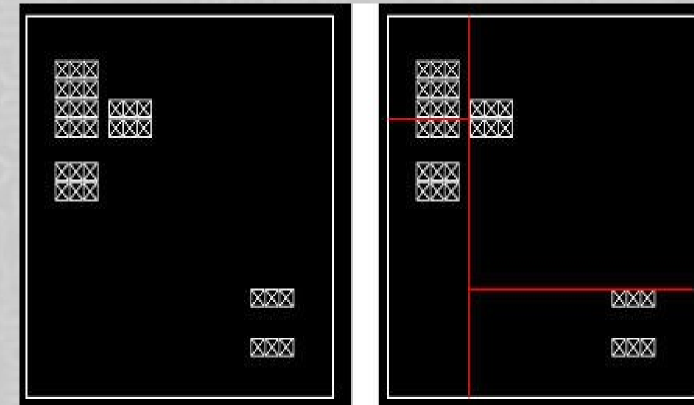
K-d дървета

- 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



K-d дървета



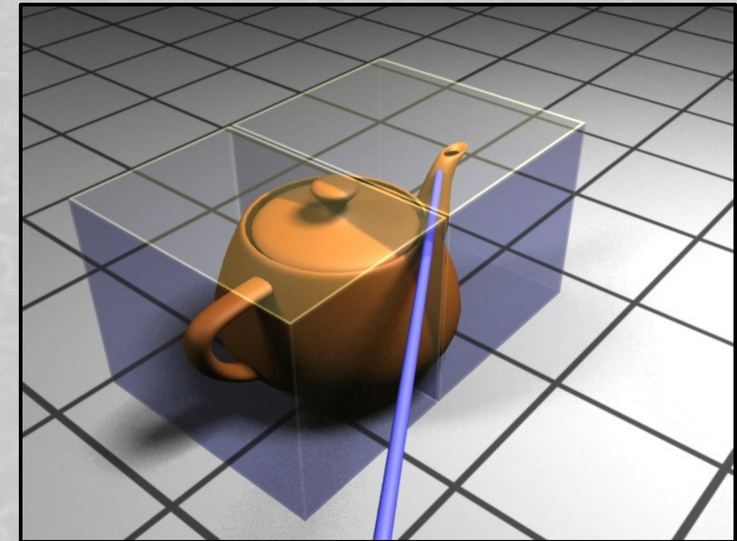
- На всяка стъпка, сечението е само по една от осите
    - Т.е., за да наподобим осмичните дървета, трябват ни три нива разцепвания (по X, Y и Z), но имаме много по-голяма свобода, защото можем да избираме позициите на деление както си искаме
    - Една от възможностите е, да търсим такава разделяща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

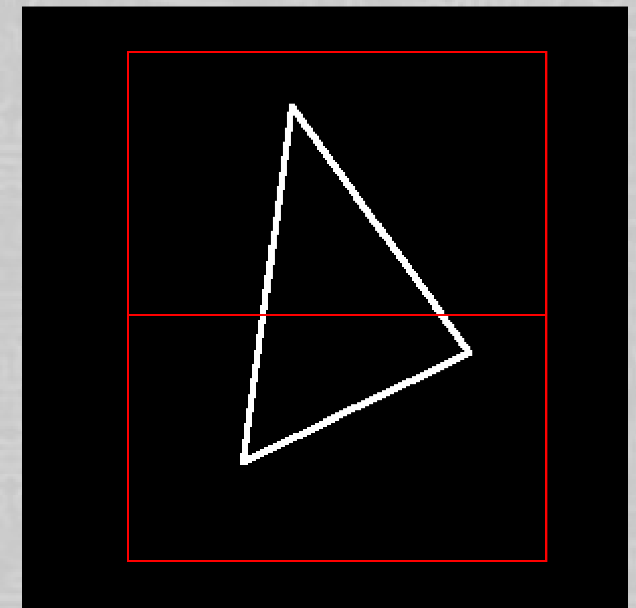
Пресичане с K-d дърво

- 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

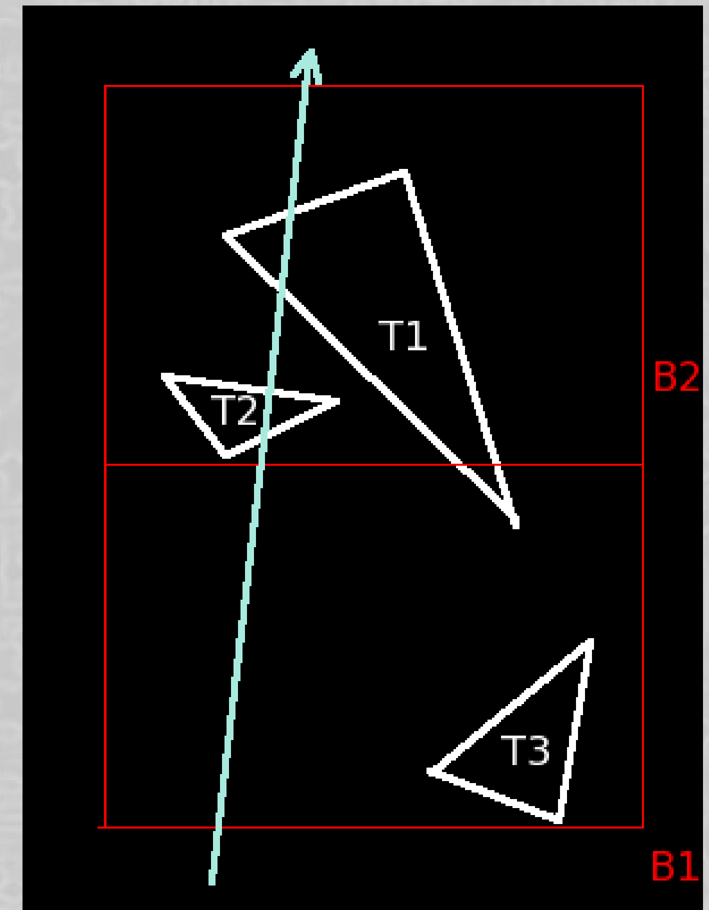
- Няколко проблема и частни случая:
  - Лъчът може да пресича и двата подвъзела
    - Можем да пресечем първо с по-близкия, и ако не намерим пресечната точка там, с по-далечния
  - Триъгълник може да принадлежи и на двете подпространства
    - Вариант 1: цепим триъгълника по границата
    - Вариант 2: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

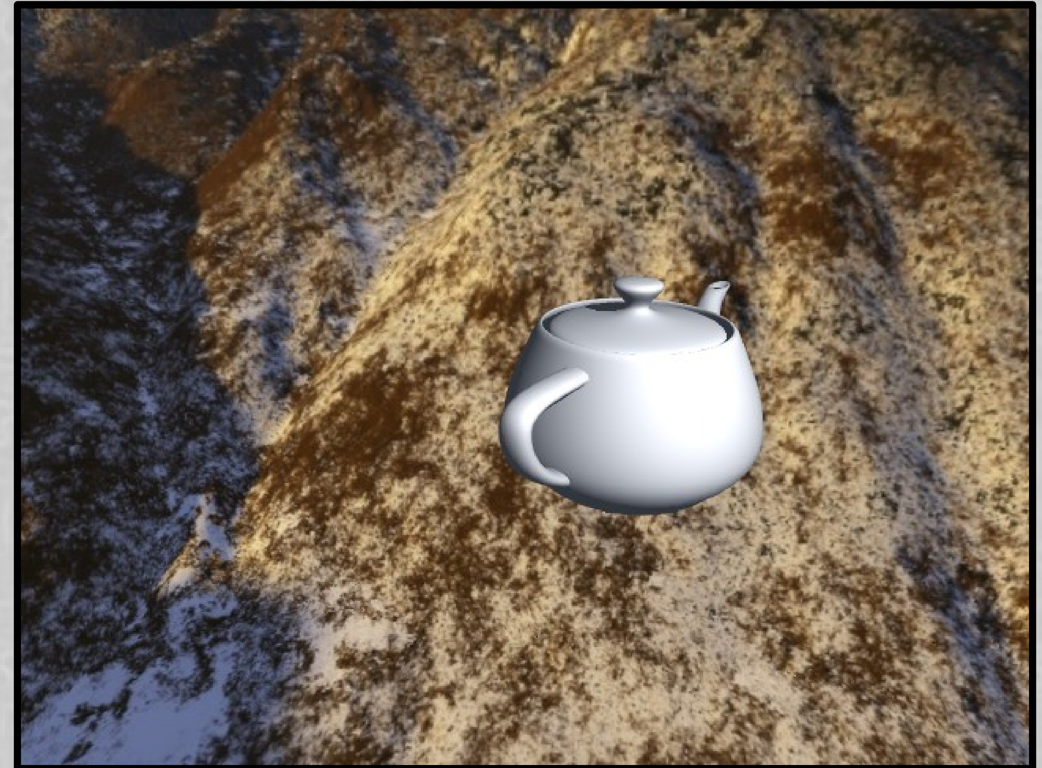
```
Function intersect(ray, info, node, bbox):
    if isLeaf(node):
        if node.triangles.intersect(ray, info) and bbox.inside(info.p):
            return info.dist
    else:
        for subnode in sorted(node.children, ray):
            subnodeBBox = bbox.split(subnode)
            if subnodeBBox.intersects(ray):
                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. Ако в някой възел се наложи да обходим и двете поддървета, то имаме log(n) + log(n) за този връх
- 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
- Резултати:



	Naïve - O(N*M)	Tree build - O(N*logN)	Рендер - O(M*logN)	Общо
Груб чайник (992 триъгълника)	17.7s	0.1s	2.3s	2.4s
Фин чайник (9120 триъгълника)	192.7s	0.7s	3.0s	3.7s

Резултати

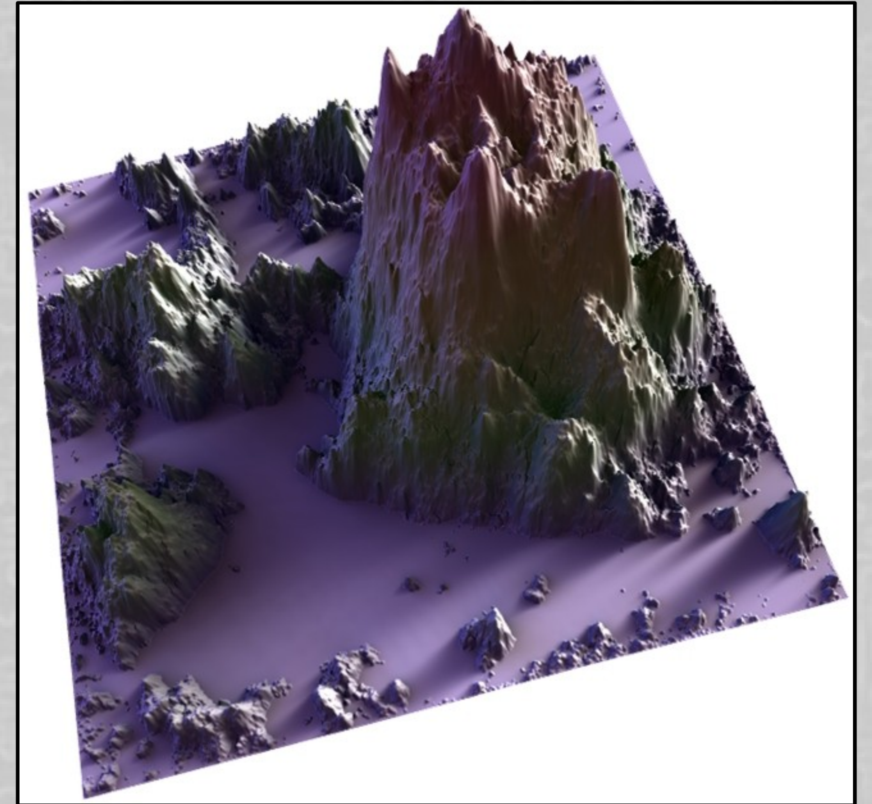
Time: ~13 минути (AA, 6 refl, 6 refr, 1024x640)

Обобщение

- Така реализираното K-d дърво ускорява пресичането с един обект – трябва да прилагаме някаква друга схема за пресичане с цялата сцена
  - Вариант 1: превръщаме всичко в триъгълни мрежи и строим едно голямо K-d дърво около всички триъгълници
  - Вариант 2: всеки обект разглеждаме като отделна единица (потенциално със собствено K-d дърво), а всичките обекти вкарваме в някаква глобална структура (равномерен грид, octree, ...). Пресичането със сцената реализираме като първо търсим в глобалната структура, и за всеки обект, преравяме в K-d дървото му

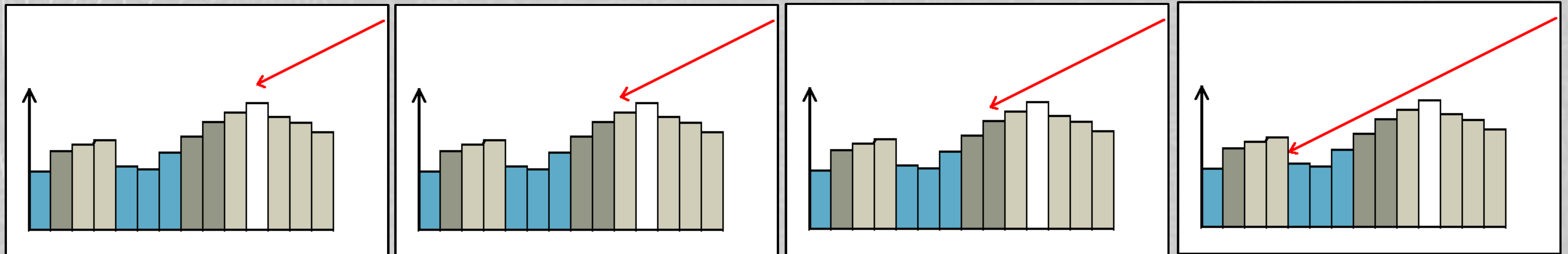
Релефни карти (heightfields)

- Релефните карти са текстури, чрез които задаваме някакъв релеф; стойността на текстурата в (x, y) указва височината на релефа в тази точка
  - Прилича на bump map, само дето указва реална геометрия
  - Не могат да бъдат „облепени“ върху обект; основата е плоска



Пресичане с релефни карти

- Наивен алгоритъм: трасираме лъча на малки стъпки (стъпката е голяма колкото един тексел от релефната карта). На всяка стъпка, проверяваме дали края на лъча не е под нивото на релефа в текущата позиция
  - Сложност: O(N) за релефна карта NxN



Наивен алгоритъм

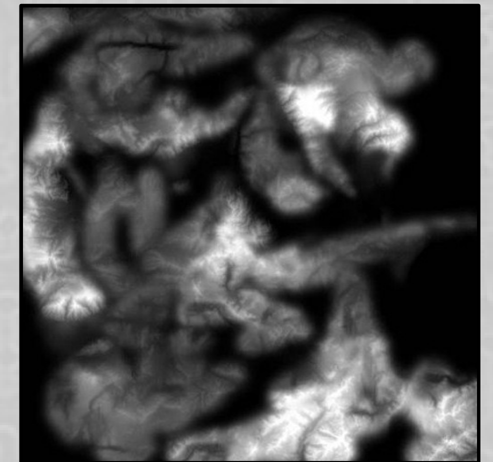
```
Function intersectHeightfield(ray):
    p = bbox.findNearestIntersectionPoint(ray)
    while bbox.inside(p):
        p += ray.dir
        if heightfield(p.x, p.z) > p.y:
            // intersection found
            return ...
    return ∞
```

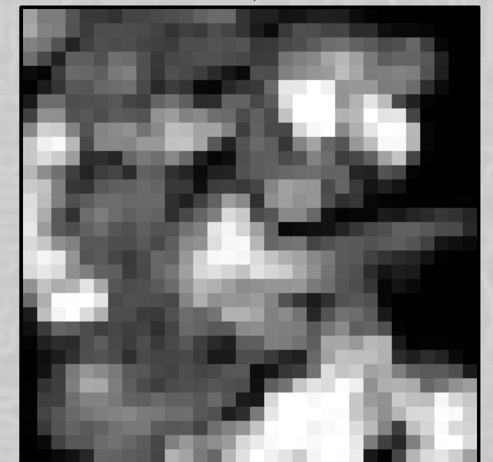
Ускорени структури за пресичане

- Триъгълна мрежа
  - Карта NxN пиксела можем да превърнем в триъгълна мрежа с 2 * N * N триъгълника, и да построим K-d дърво
  - Много неефективно като памет, не се възползва от факта, че heightfield-а има специфични ограничения

Ускорени структури за пресичане

- Частична йерархия
  - Ще разбием входната релефна карта на блокове от (например) 16x16 пиксела и за всеки блок ще пазим най-високата точка
    - Така от 512x512 входна картинка ще получим 32x32 „груба“ релефна карта
  - Трасираме лъча в грубата карта (така прескачаме цели блокове от 16x16 пиксела в истинската)
  - При пресичане в грубата карта, пускаме лъча в съответния блок в истинската



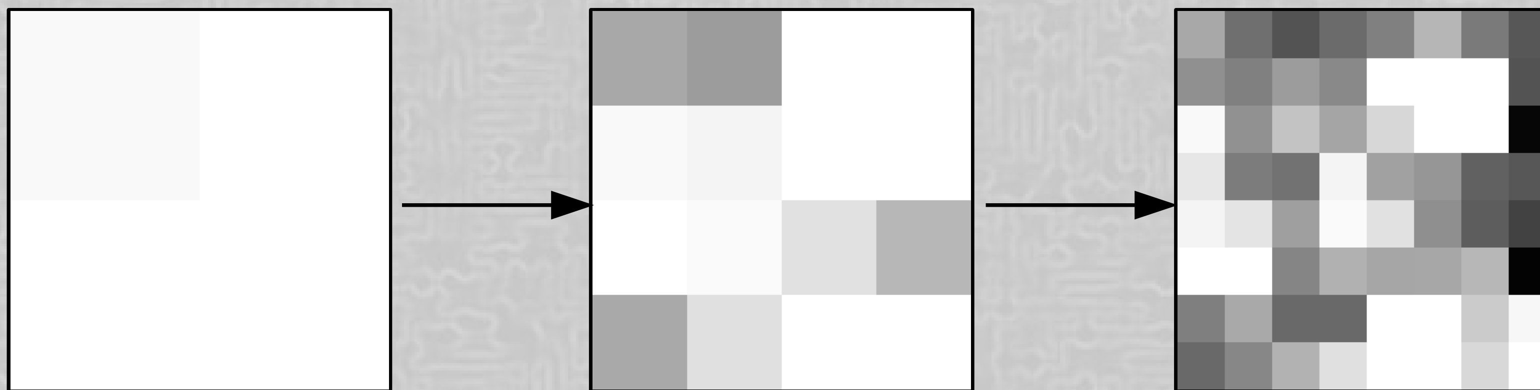


Ускорени структури за пресичане

- Сложност: O(N/K + K), където K е големината на един блок. Оптимално е K = sqrt(N), при което сложността става O(sqrt(N))

Ускорени структури за пресичане

- Пълна йерархия
  - Подобна на частичната, само че ползваме много отделни карти с различна степен на детайлност – 2x2, 4x4, 8x8, ..., NxN
  - Пресичането е подобно на търсене в quadtree
  - Недостатъци: доста тежка за реализация. Пресичащата функция е рекурсивна, което я бави допълнително
  - Сложност: O(log N) (best case – когато релефа е много слаб)

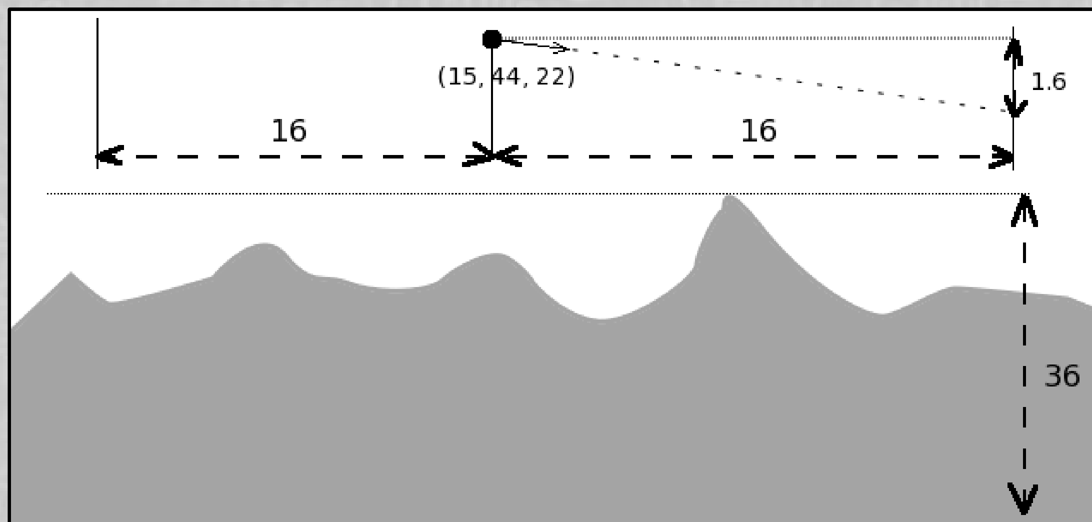


Ускорени структури за пресичане

- Най-бързо спускане
  - Идеята: все едно реализираме наивния алгоритъм, само че ще преместваме точката *p* с максималното разстояние, за което е няма опасност да се пресечем с някой от близките върхове на релефа
  - Ще реализираме структура, която да отговаря на въпросите getHighest(x, y, 2^k) - „кой е най-високия връх в кръга с център x, y и радиус 2^k?“

Най-бързо спускане

- Например: p = (15, 44, 22), dir = (1, -0.1, 0), getHighest(15, 22, 2^4) = 36
  - Намираме се в (15, 22) спрямо релефната карта, и в радиус 16 от нас няма връх, по-висок от 36. Същевременно, ако продължим 16 единици по посоката на лъча, ще се снижим само с 16*-0.1 = -1.6, т.е. до височина 42.4. Няма опасност да пресечем нищо, може да прескочим поне 16 стъпки, и има смисъл да проверим с k+1, т.е. за радиус 32.



Най-бързо спускане

- Изобщо, искаме да намерим най-голямото k, за което p.y + dir.y * 2^k > getHighest(p.x, p.z, 2^k), след което пропускаме 2^k стъпки

Най-бързо спускане

```
Function intersectHeightfield(ray):
    p = bbox.intersectNearest(ray)
    while bbox.inside(p):
        k = 1
        while p.y + dir.y * 2^k > getHighest(p.x, p.z, 2^k):
            k += 1
        p += dir * 2^(k-1)
        if heightfield(p.x, p.z) > p.y:
            // intersection found
            return ...
    return ∞
```

Най-бързо спускане

- Ако getHighest() е константна операция (напр., реализира се с масив), то сложността на пресичането е $O(\log^2 N)$ за представената реализация, и $O(\log N)$, ако се направи по-умно
- Но строенето на структурата е тежко; трябват ни да пресметнем предварително getHighest() за всеки *x*, всеки *y*, и всяко 2^k за $k = 0, 1, 2, ..., \log_2(N) - 1$
- Намирането на най-високият връх в даден кръг изисква да го търсим из цялата картинка (за $k = \log_2(N) – 1$ например), т.е., търсенето е N^2

Строене на ускоряващата структура

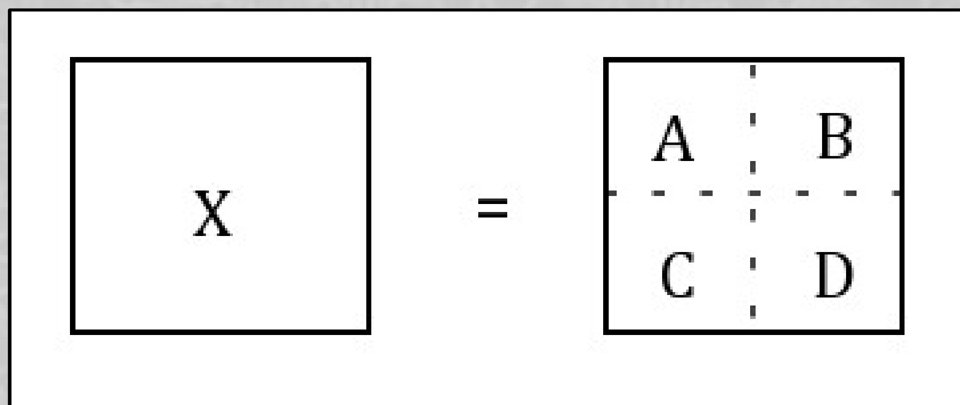
- Тоест, $N^2 * \log_2(N) * N^2$, или $O(N^4 \log N)$!
  - Това е прекалено много; дори за heightfield 512x512 ще отнеме много време, много повече от рендерирането на един кадър например

Строене на ускоряващата структура

- Вместо кръг, може да ползваме квадрат (със страна 2*r, т.е., описан квадрат около кръга). Това ще направи getHighest() малко по-консервативно (заради допълнителните пиксели, които квадрата обхваща), но не е фатално
- Пресмятането на най-високия връх на квадрата решаваме чрез техниката динамично програмиране, т.е., ползваме вече решените случаи, за да сметнем следващите (в случая, ще почнем от по-малките квадрати и ще вървим към по-големите)

Строене на ускоряващата структура

- max(X) = max(max(A), max(B), max(C), max(D))
- A, B, C и D ги имаме вече сметнати – техните размери са двойно по-малки.

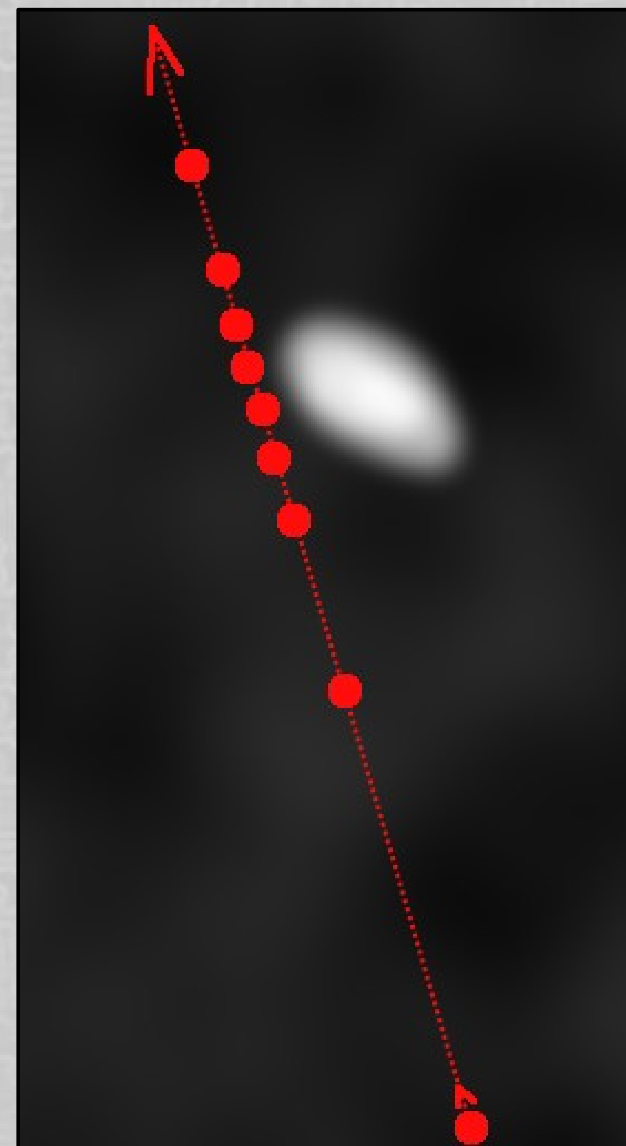


Оптимизации

- Стъпката (dir) скалираме така, че XZ дължината й да е 1
- Ако в някой момент k = 0 (близо сме до релефа), можем да приложим директно няколко итерации на наивния алгоритъм, без да проверяваме в getHighest() на всяка стъпка – по-бързо е
- Може допълнително да скалираме стъпката, ако е насочена към ръбовете на квадрата, с цел да се компенсира ползването на квадрат вместо кръг
- Или може да ползваме наистина кръгчето, преизчислението може да се свали до $O(N^3 * \log^2 N)$

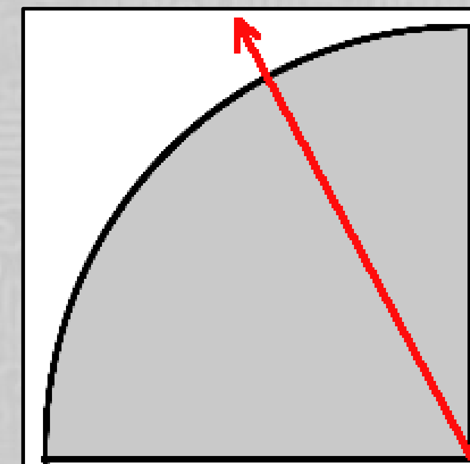
Още оптимизации

- Да разгледаме как се държи нашия алгоритъм, ако лъча пропуска наблизо (отстрани) някакъв висок връх
- В началото лъчът приближава „смело“ към върха, а около него е „предпазлив“ и върви на малки стъпки. След като го подмине, обаче, предпазливостта остава, защото все още има висок връх наблизо, макар и вече отзад



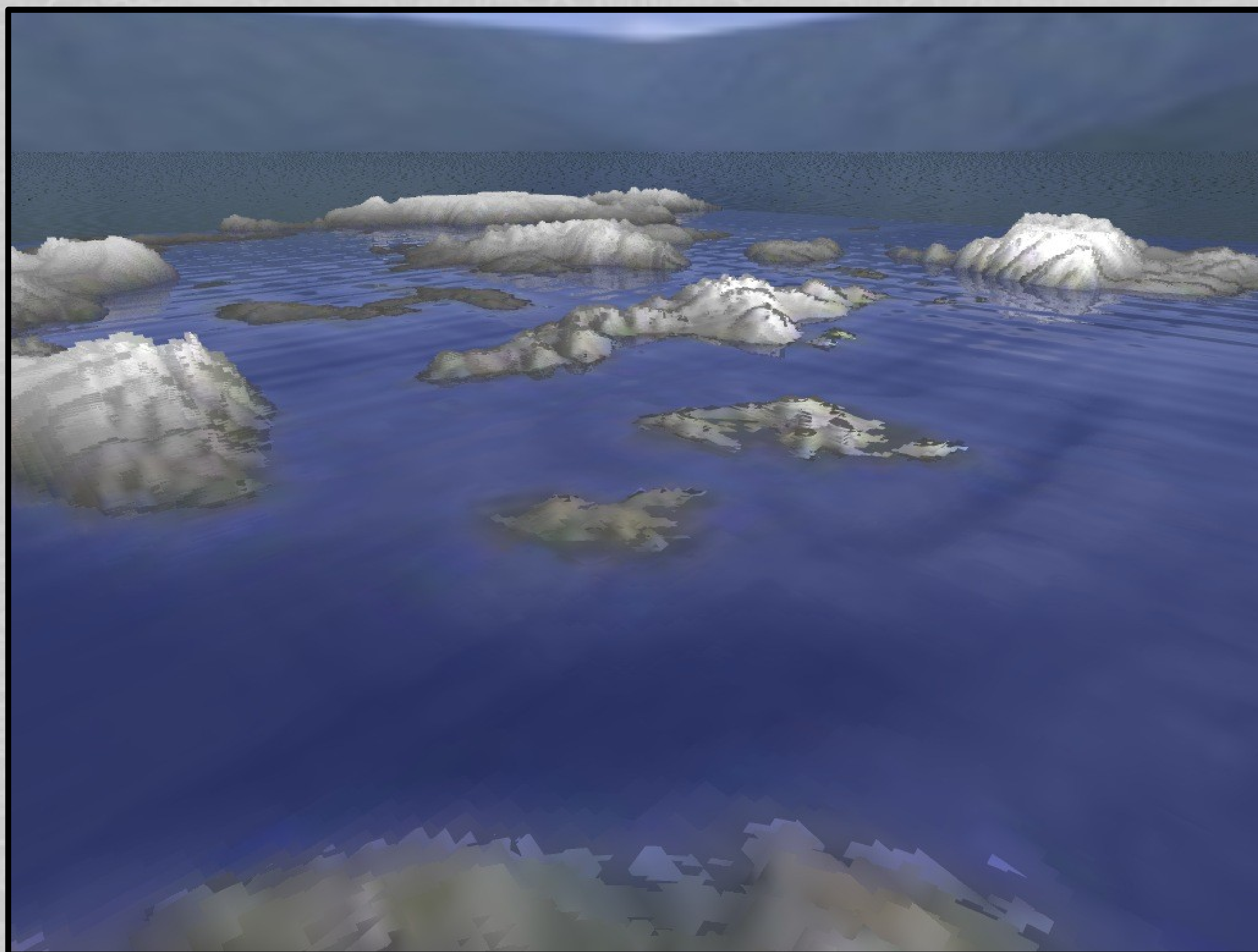
Още оптимизации

- От началната посока на лъча може да извлечем в кой квадрант се намира (посоката), и да модифицираме getHighest да има още един параметър – квадрант (1-4), т.е., да търси най-високия връх, но само в квадранта, в който сочи лъча
  - В примера с върха, след като го подмине, върха вече ще е в друг квадрант и трасирането ще продължи с пълна скорост
  - Тази оптимизация харчи още повече памет, без да печели убедително много време



Резултати

N	450px
Rendering – Naïve - O(N)	13s
Struct build - $O(N^2)$	0.4s
Rendering – Fastest descent – $O(\log^2 N)$	7.4s



Обобщение

- На практика, за по-големи релефни карти, пълната йерархия е по-добрият вариант, защото харчи значително по-малко памет
  - Пример: 2048x2048 карта, най-бързото спускане ще изразходи ~192 MiB памет. Пълната йерархия коства само ~21 MiB.