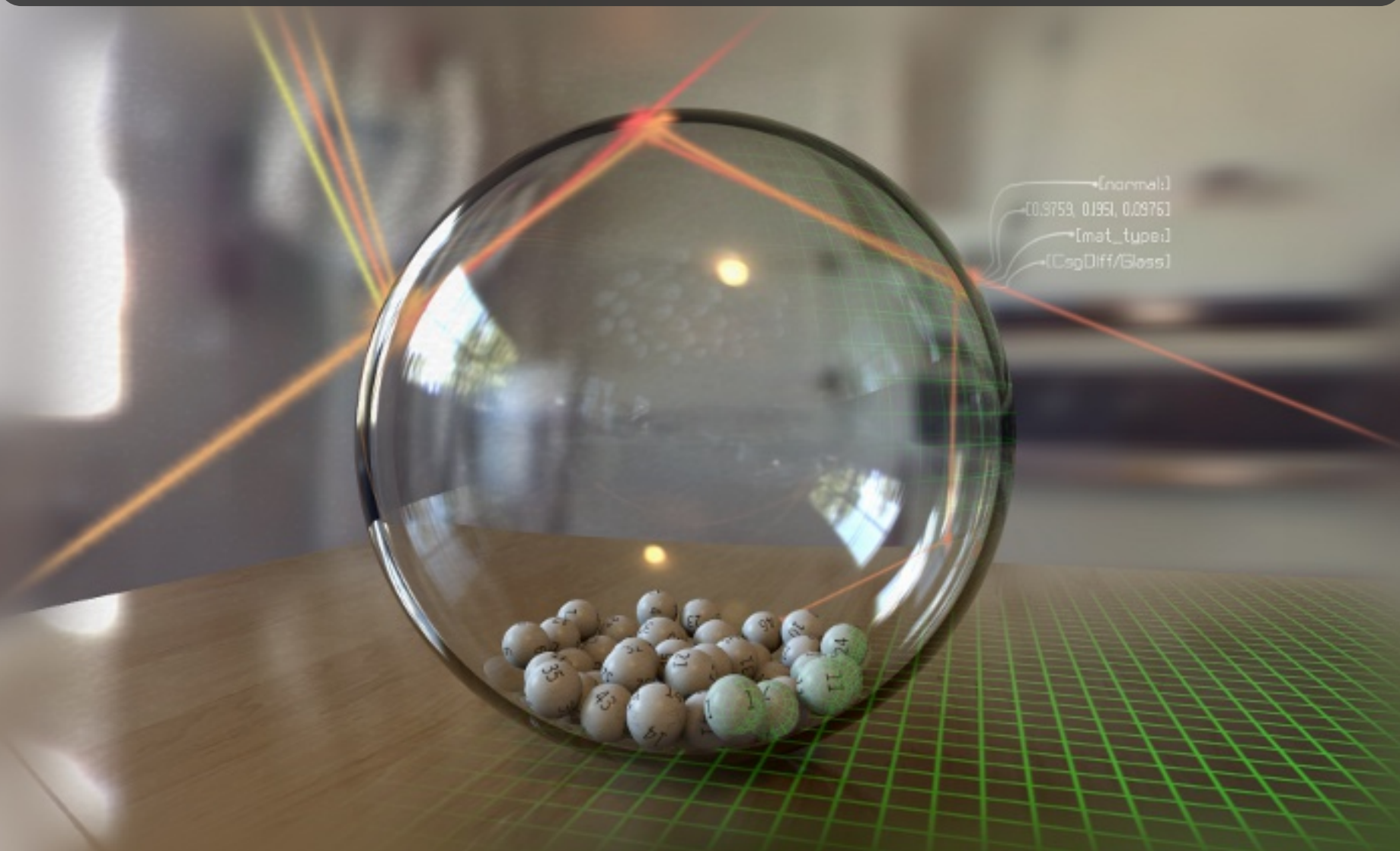


3D графика и трасиране на лъчи v.3.0



<http://raytracing-bg.net/>

Тема 5

Пресичане с по-сложни примитиви

Булеви операции

Ambient цвят, текстури

Antialiasing

Model трансформация

Съдържание

- Bugfixing (от предната лекция)
 - Пресичане със сфера
 - Lambert, Phong
- Сенки
- Пресичане с куб
- Булеви операции
 - Обединение
 - Сечение
 - Разлика

Съдържание (2)

- Ambient цвят
- Bitmap текстури
- Заглаждане на ръбовете (anti-aliasing)
- Model трансформация

Новини

- Ново лого
- Домашни
 - ДР4 – до следващата Неделя
 - ДР5 – ще се публикува утре
- Записи

Raytracing алгоритъм

- **Function** raytrace(ray) :
closestDist = $+\infty$
closestNode = None
for each node **in** sceneNodes:
 intersectionInfo = intersect(ray, node)
 if intersectionInfo.distance < closestDist:
 closestDist = intersectionInfo.distance
 closestNode = intersectionInfo.node
if closestDist == $+\infty$:
 return backgroundColor
return closestNode.computeColor()

Пресичане със сфера

- Разстоянието от центъра на сферата O до произволна точка T е: $\text{sqrt}((T_x - O_x)^2 + (T_y - O_y)^2 + (T_z - O_z)^2)$
- Позицията на всяка точка по лъча е: $\text{start} + p * \text{dir}$
 - Където $p > 0$ е параметър
 - Разстоянието от всяка точка на лъча до сферата се получава по горната формула, като заместим $T = \text{start} + p * \text{dir}$, за някое конкретно p
 - Търсим такова p , че разстоянието да е точно R (радиуса на сферата)

Пресичане със сфера

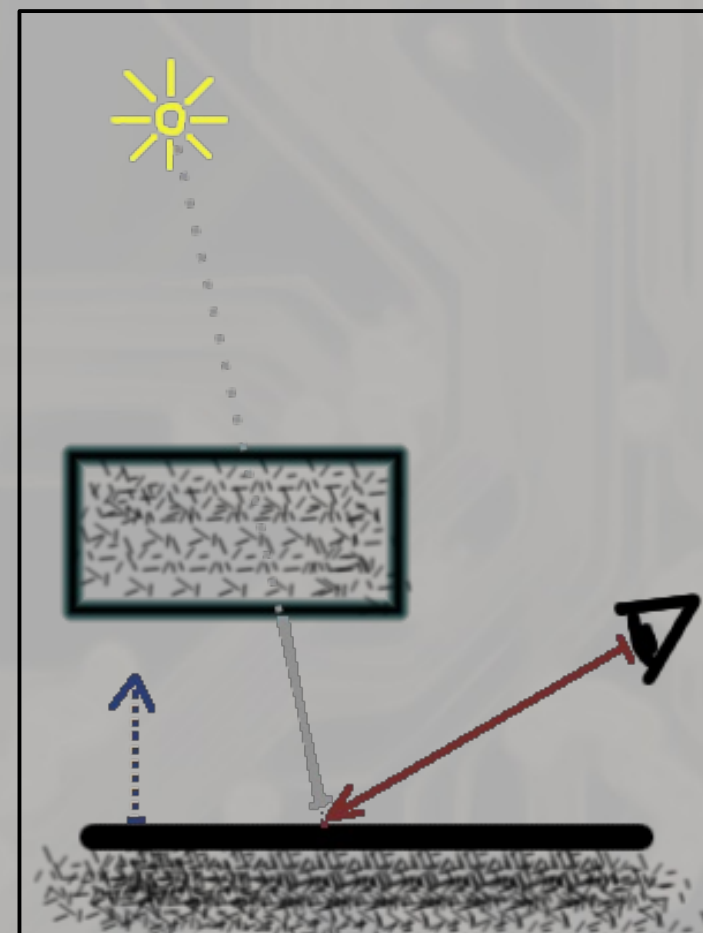
- $\text{sqrt}(((\text{start}_x + p \cdot \text{dir}_x) - O_x)^2 + ((\text{start}_y + p \cdot \text{dir}_y) - O_y)^2 + ((\text{start}_z + p \cdot \text{dir}_z) - O_z)^2) = R \Leftrightarrow$
 - (нека $H = \text{start} - O$):
- $\text{sqrt}((H_x + p \cdot \text{dir}_x)^2 + (H_y + p \cdot \text{dir}_y)^2 + (H_z + p \cdot \text{dir}_z)^2) = R \Leftrightarrow$
- $(H_x + p \cdot \text{dir}_x)^2 + (H_y + p \cdot \text{dir}_y)^2 + (H_z + p \cdot \text{dir}_z)^2 = R^2 \Leftrightarrow$
- $p^2(\text{dir}_x^2 + \text{dir}_y^2 + \text{dir}_z^2) + 2 \cdot p(H_x \text{dir}_x + H_y \text{dir}_y + H_z \text{dir}_z) + (H_x^2 + H_y^2 + H_z^2 - R^2) = 0 \Leftrightarrow$
- $p^2 \cdot \text{dir.length}^2 + p \cdot (2 \cdot \text{dir} \cdot H) + (H.length^2 - R^2) = 0$
- Квадратно уравнение!
 - 0 решения – лъчът не пресича сферата
 - 1 или 2 решения – лъчът допира или пресича сферата

Осветление

- Ще използваме най-простия модел за лампа – точкова
- Точковата лампа има две характеристики
 - Позиция
 - Интензитет
- Точковата лампа разпръсква светлина във всички посоки
- Осветеността на дадена точка намалява с квадрата на отдалечеността ѝ от лампата ($1/r^2$)
 - Защо?

Сенки

- Проверка дали точка е в сянка
 - Пресечната точка се отмества малко по посока нормалата на повърхината
 - От там се пуска нов лъч в посока лампата
 - Ако удари обект, на разстояние по-малко от това до лампата → имаме засенчване



Материали

- Ще изнесем генерирането на дифузен цвят в отделен клас – Texture.
- Ще напишем 2 прости материала
 - Ламберт (или дифузен) модел
 - За матови повърхности, които не хвърлят отблясъци
 - Фонг (Phong) модел
 - За „лъскави“ повърхности

Ламберт vs Фонг



Ламберт

- Осветлението зависи единствено от ъгъла, под който пада светлината спрямо повърхността
 - Нормалният вектор (`info.norm`) е перпендикулярен на геометрията в пресечната точка
 - т.е., ако имаме допирателна равнина в тази точка, това е нейният нормален вектор
- Яркост = $\cos(\Theta)$
 - Може да се изведе геометрично



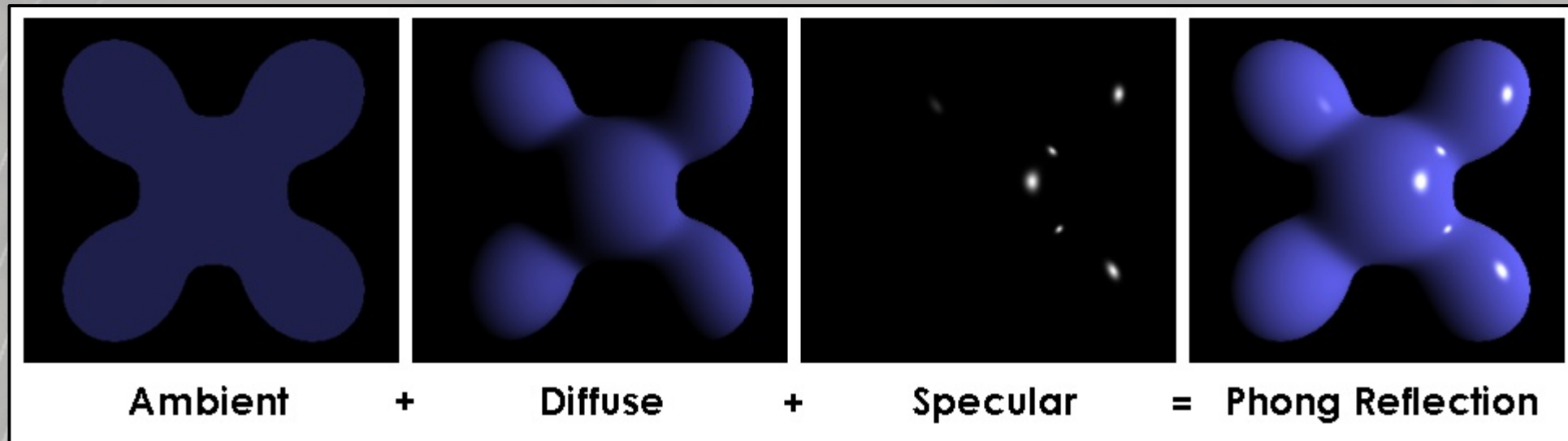
Ламберт



- $\text{OutColor} = \text{materialColor} * \max(0, \cos(\Theta)) * \text{lightColor}$
- $\cos(\Theta) = \text{dot}(\text{lightVec}, \text{info.norm})$

ФОНГ

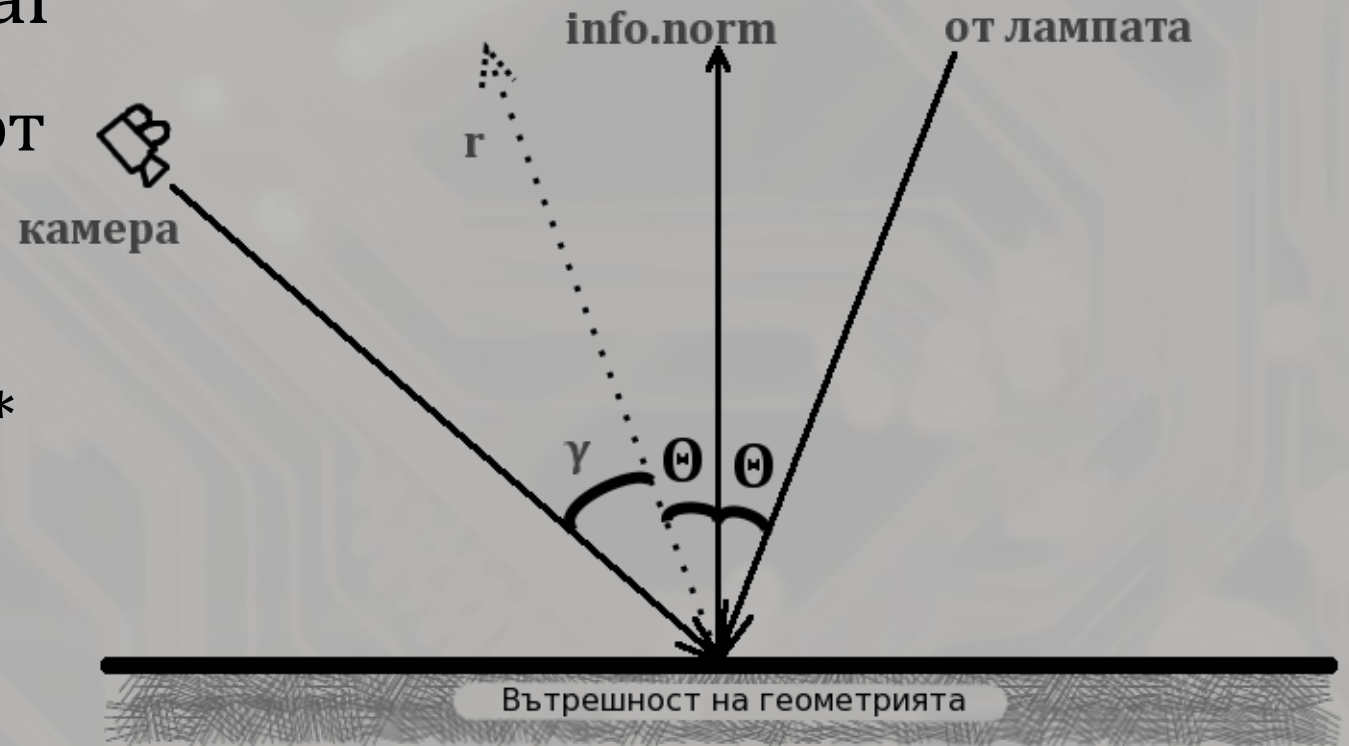
- ФОНГ е същото като Ламберт, само че добавяме и „лъскав“ (specular) компонент.



- Specular компонента се определя от това дали отражението на идващата светлина е близко до посоката на камерата

Фонг

- Фонг = Ламберт + specular
 - r е отражението на лъча от лампата, спрямо нормалата `info.norm`
 - $\text{specular} = \max(0, \cos(\gamma))^{\eta} * \text{lightColor}$
 - Параметърът η (обикновено наричан „exponent“) контролира „колко лъскав е материала“



$$r = \text{reflect}(\text{lightVec}, \text{info.norm})$$
$$\cos(\gamma) = \text{dot}(r, -\text{cameraRay.dir})$$

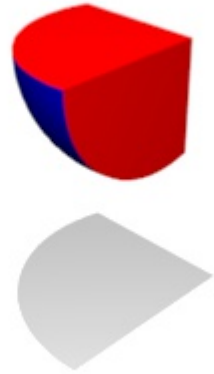
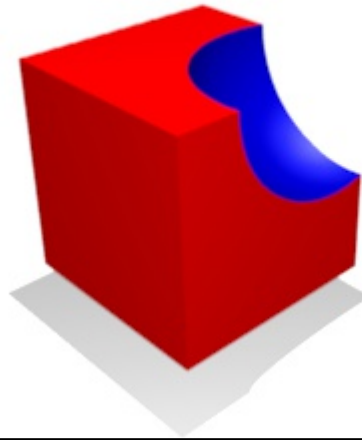
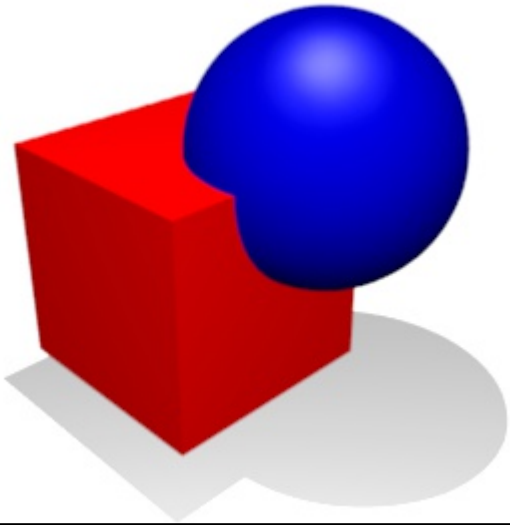
Пресичане с куб

- За всяка страна приемаме, че пресичаме с равнина...
 - ... и проверяваме дали пресечната точка е в граници по другите две измерения
 - Подобно на пресичането със сферата, трябва да внимаваме за стени, намиращи се зад камерата
 - т.е. търсим най-близката положителна пресечна точка.
- Нормалите зависят единствено от стената
 - Например, за стената +X, нормалният вектор е $(1, 0, 0)$
- UV координатите могат да се генерират по много начини

Пресичане с куб

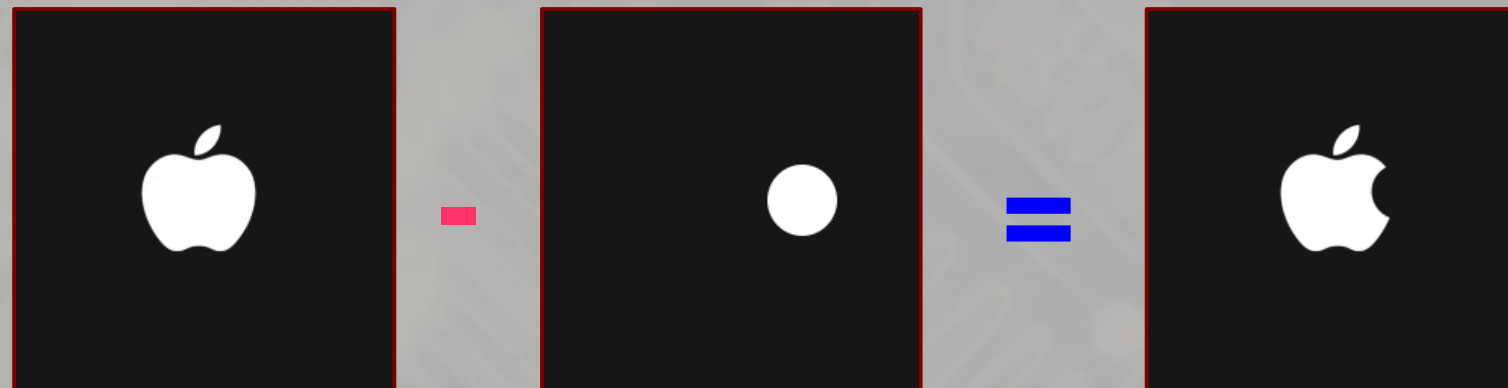
- UV:
 - $(u, v) := (x + y, z)$ – съвпада по ръбовете, но разтегля текстурата по две от стените
 - $(u, v) := 2 * (p.x - \text{cube.x}, p.y - \text{cube.y}) / \text{cube.side}$ (за стена +/-Z)
 - Разкъсва текстурата по ръбовете, но изглежда добре
 - Сферични координати:
 - $(u, v) := (\text{atan2}(p.z - \text{cube.z}, p.x - \text{cube.x}), \text{asin}(2 * (p.y - \text{cube.y}) / \text{cube.side}))$
 - Изглежда доста странно

Булеви операции



Булеви операции

- Често е полезно да създадем нов 3D обект, като комбинираме два съществуващи, и се интересуваме от някоя част на двата обекта, която да изпълнява някакво логическо (булево) условие
- На английски: CSG = Constructive Solid Geometry



Apple's logo

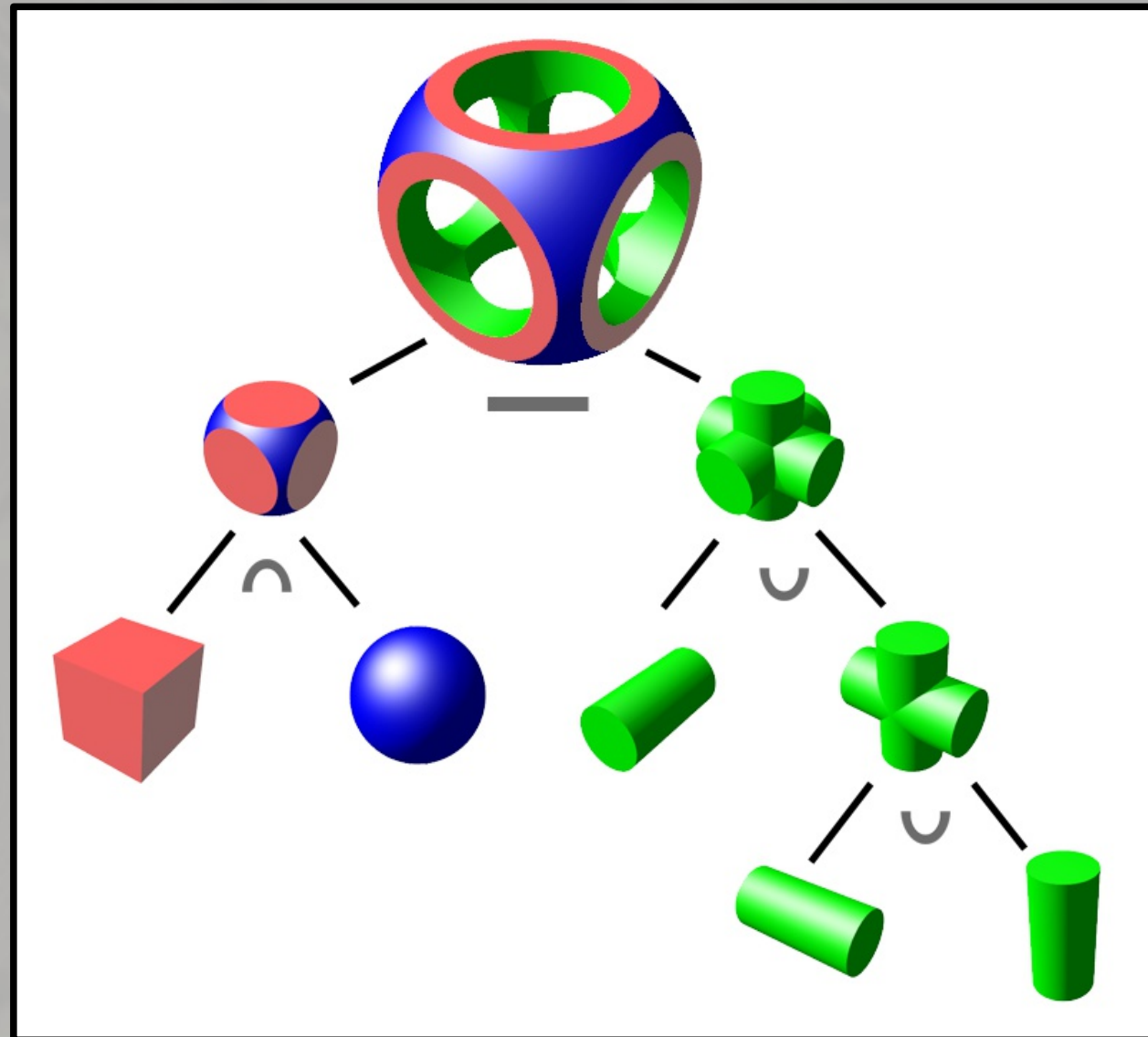


NOW WE KNOW WHO TOOK THE BITE !

Булева операция между два обекта

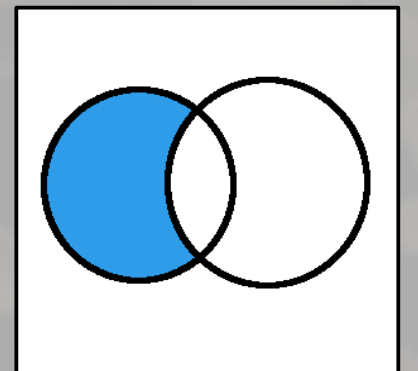
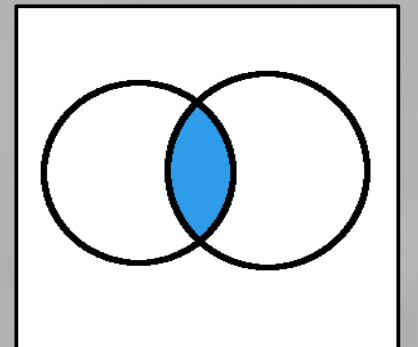
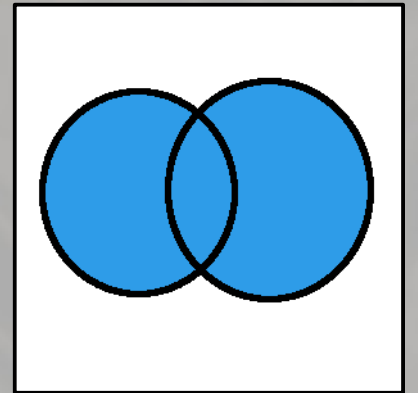
- Алгоритъмът за булеви операции е специфичен за Raytracing-а. При Z-buffer, например, аналогичен алгоритъм няма; булевите операции там се реализират много по-трудно
- Булевата операция между два обекта е фундаментална; при повече от един обект, може да реализираме дървовидна структура (двоично дърво), при което резултатът от някаква булева операция да се ползва за вход на друга булева операция
 - Ще дадем пример

CSG trees



Видове булеви операции

- Обединение
- Сечение
- Разлика
 - Несиметрична операция!
- ...и всъщност, произволна булева функция на два аргумента – алгоритъмът е един и същ, променя се само самата булева функция
 - Булевата функция е израз от вида $\text{в\`тре_сме_в}(A) \ \&\& \ \text{не_сме_в\`тре_в}(B)$



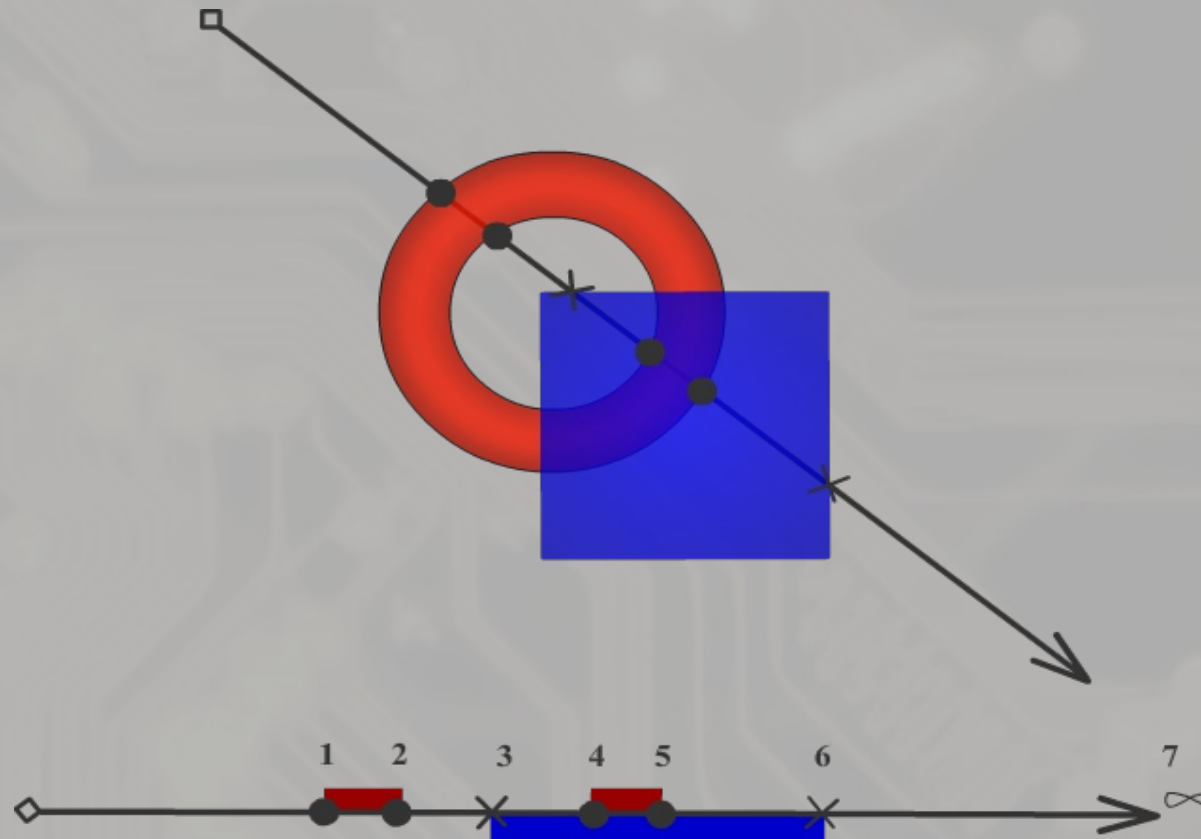
Алгоритъм за CSG с две геометрии

- Намираме всички пресечни точки на лъча с всяка от двете геометрии
 - Това включва всички пресечни точки на геометрия по лъча (например, за сфера, те са 1 или 2; за по-сложни примитиви, може да са повече)
- Обединяваме двата списъка с пресечни точки. Обединението сортираме по разстояние до пресечната точка
- Вървим по сортирания списък, като за всяка точка
 - Изчисляваме дали сме вътре във всяка от двете геометрии...

Алгоритъм за CSG с две геометрии

- ... и ако изпълняваме условието на булевата функция, значи сме намерили пресичане – обявяваме пресечната точка за пресечна на лъча с булевия обект
- Проверката дали сме вътре в някоя геометрия е лесна
 - Може да определим дали началото на лъч е вътре или извън дадена геометрия, като преброим пресечните точки на лъча с геометрията
 - Нечетен, ако сме вътре; четен, ако сме извън нея
- При подминаването на пресечна точка, сменяме флага, показващ дали сме „вътре в геометрията“ (за тази геометрия, с която е пресечната точка)

Алгоритъм за CSG



- Операцията е сечение. Алгоритъмът ще намери точка 4 (това е първата точка, в която условието „вътре_сме_в_(A) && вътре_сме_в_(B)“ е изпълнено)

Нормали

- Принципно, всяка точка от една повърхнина има нормален вектор; най-общо, той сочи „навън“ от „тялото“ на обекта
 - Пример за сфера: $N := \text{normalize}(\text{intersection.pos} - \text{sphere.pos})$
- Обикновено, нормалите са ориентирани така, че лъчът R от пресечната точка към камерата да е в същото полупространство като N , т.е. $N \cdot R > 0$
- Backface culling: това е оптимизация, при която пропускаме (части от) обекти, за които $N \cdot R < 0$

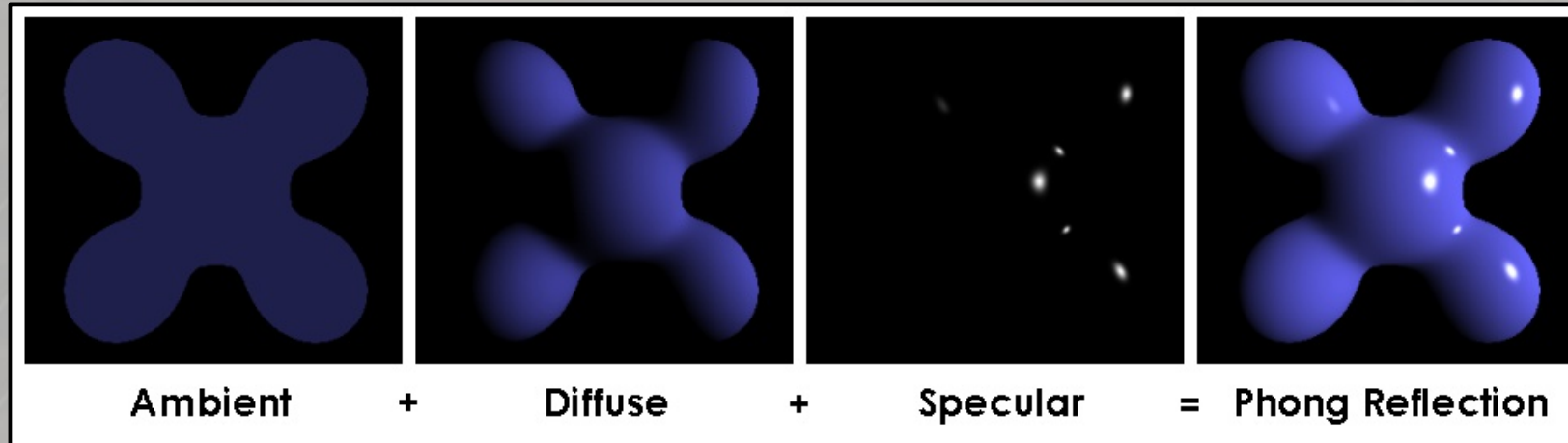
Нормали

- Когато лъчът тръгва от вътрешността на даден обект, обаче, нещата са по-различни
 - Например, ако тръгваме от средата на куб, backface culling-ът ще игнорира всички страни на куба
 - Geometric normal (gnormal) – истинският нормален вектор на повърхнината
 - Camera normal (normal) – нормален вектор, който е ориентиран в посока на камерата ($R * N_c \geq 0$)
 - Т.е., или $normal = gnormal$, или $normal = -gnormal$
 - Реализирано в нашия код от `faceforward()` функцията

Ambient light

- Светлината от обкръжението (ambient light) изразява факта, че една (малка) част от светлината не идва директно от лампите, а чрез отражения от други (неизлъчващи) предмети.
 - В природата, наситени сенки (Color = (0, 0, 0)) са рядкост
 - Досега ние смятахме само директната светлина; ambient light цели да симулира индиректната светлина
 - Ambient е просто някакъв константен цвят, който се добавя към цялото осветление
 - Груб „hack“, целящ да свърши работата на Global Illumination алгоритмите

Ambient light



- Ambient участва във Phong модела, като компонент, който не се влияе от косинуса между светлината и нормалния вектор или от наличието на сянка; просто константен цвят, умножен по текстурата или базовия цвят на обекта.

Ambient \neq Global Illumination

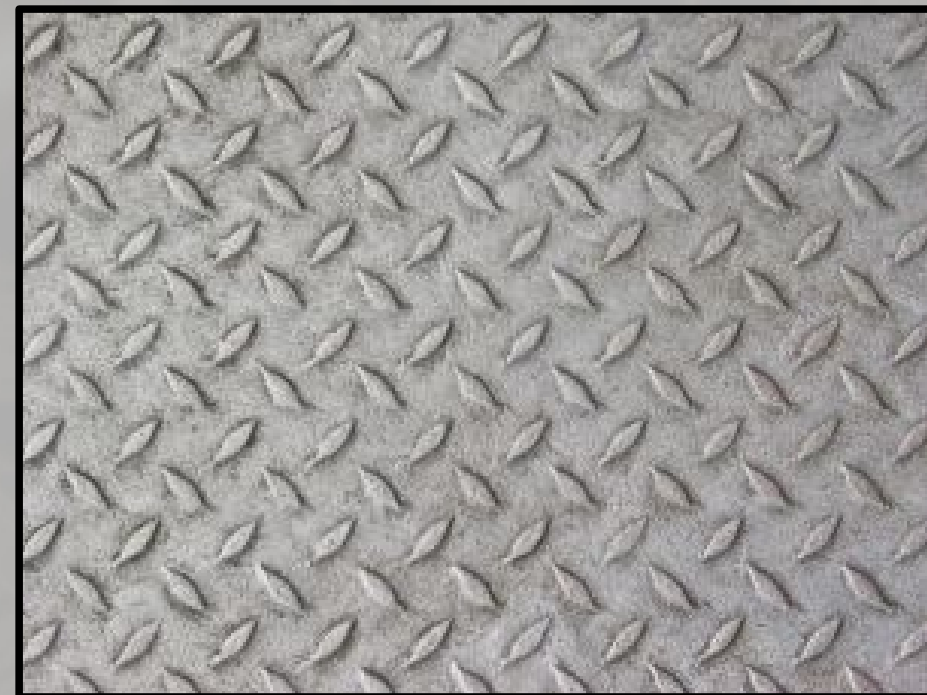
- Ambient е еднакъв за всички точки от сцената
- При Global Illumination алгоритмите, именно това е трудният момент – да се сметне колко трябва да е ambient цвета, в зависимост от това къде по сцената се намираме
 - Например, в стая с една зелена и една червена стени, близо до зелената стена ще имаме по-зеленикав ambient. Този ефект се нарича Color bleeding.
 - Ambient **не** може да симулира този ефект!

Color bleeding

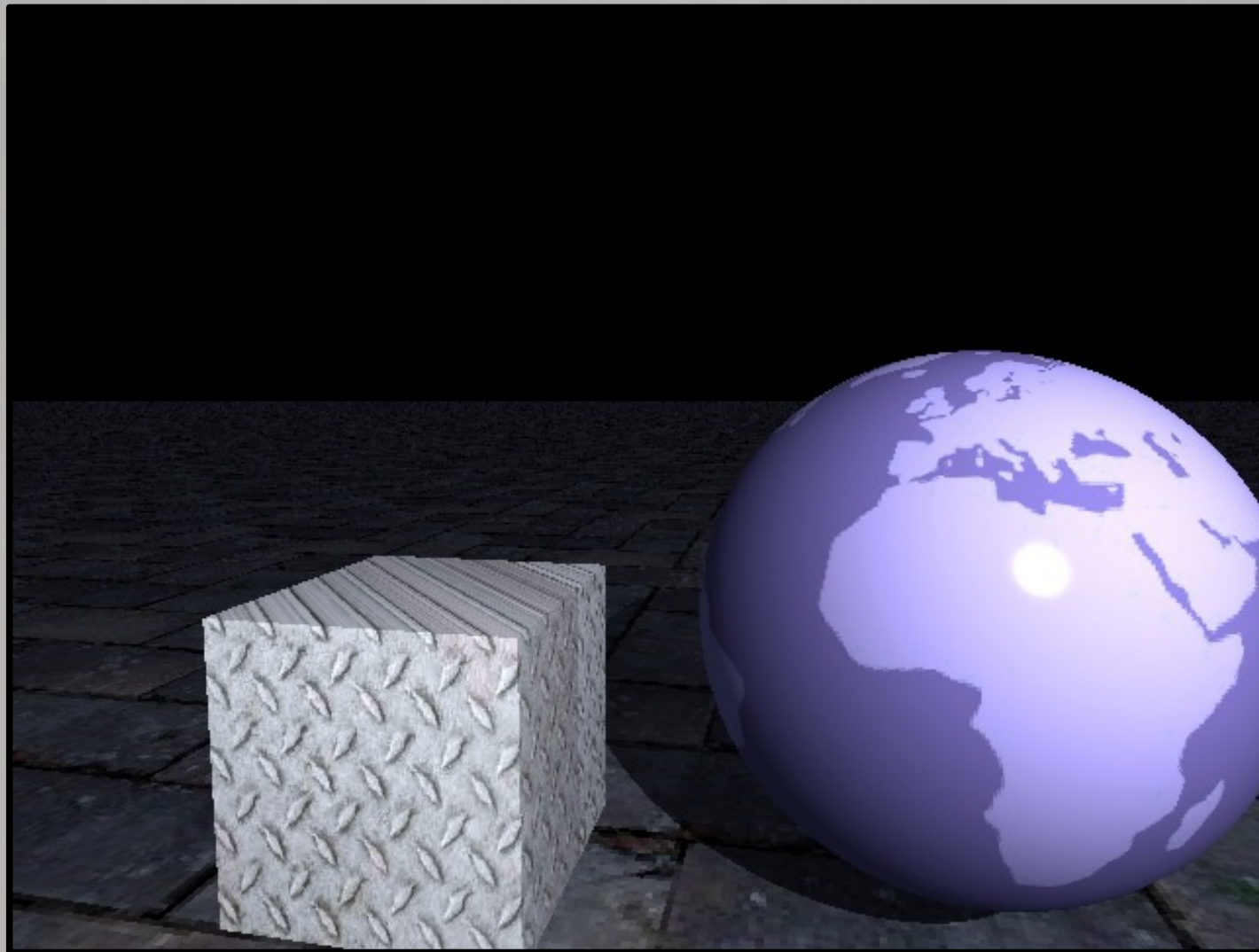


Витмар текстури

- Като цяло, текстурите се ползват за да създадат усещане за допълнителен детайл, какъвто обектът няма геометрически
- Тук ще говорим за „дифузни“ текстури – такива, които променят цвета на обекта
 - По-нататък ще разгледаме и други видове текстури



Вітмар текстури



Bitmap текстури

- При bitmap текстурите, цветът не се генерира процедурно, ами се взема от картинка
- Обикновено се абстрахираме от размера на картинката и приемаме, че нейните координати са в квадрата $(0,0)$ - $(1,1)$, като оставяме скалирането скрито в `getColor()` на текстурата
- Геометрията трябва да се грижи да създава удобни u,v координати, които да могат правилно да се облепят с текстури.

UV координати - пример

- В примера със сферата и картата на света, нямаме повторение (tiling) на текстурата, така че uv координатите трябва да са в $[0, 1]$
- u пробягва по ширината на картинката, т.е. u отговаря на географска дължина
 - $u = (\pi + \text{atan2}(\text{info.p.z} - \text{sphere.z}, \text{info.p.x} - \text{sphere.x})) / (2\pi)$
- v пробягва по дължината на картинката, т.е. v отговаря на географска ширина
 - $v = 1.0 - (\pi/2 + \text{asin}((\text{info.p.y} - \text{sphere.y}) / R)) / \pi$

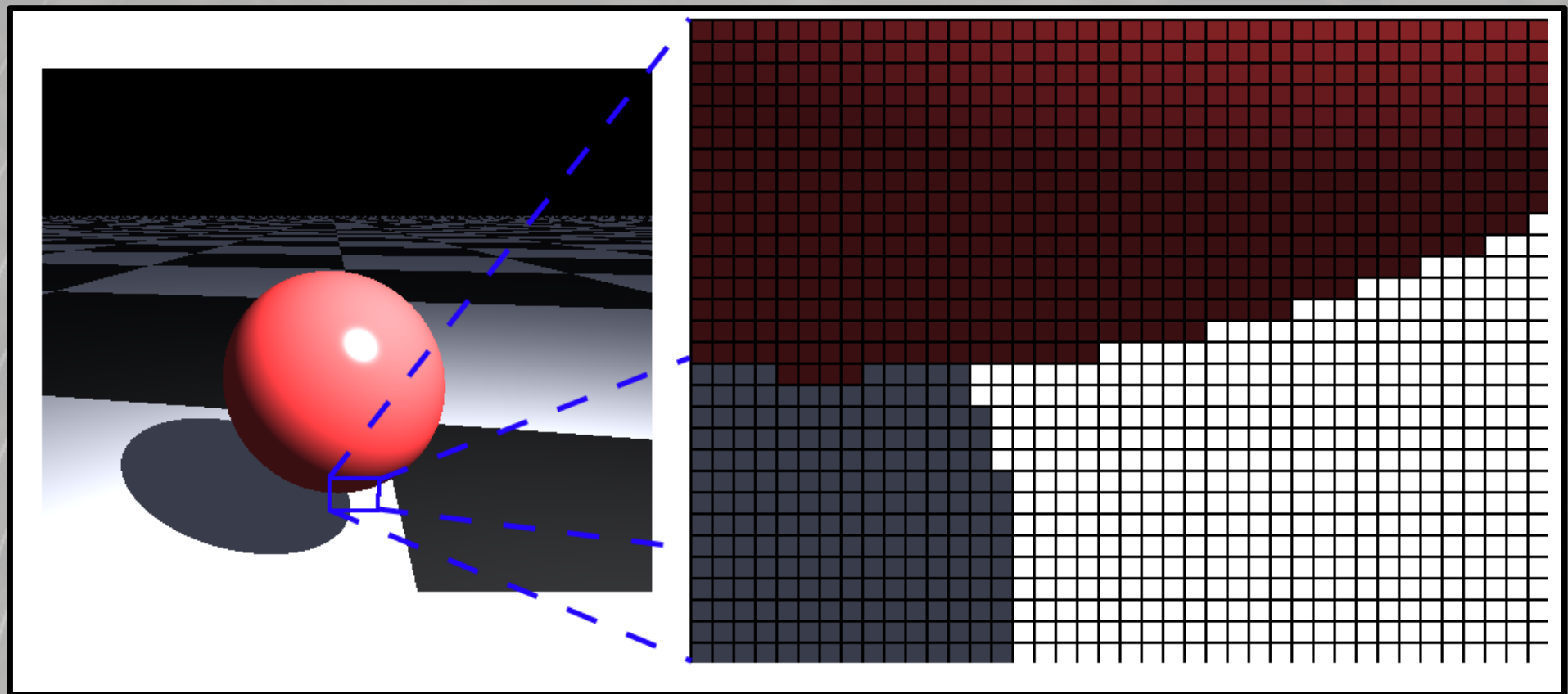
Anti-aliasing

- Под „aliasing“ ще разбирате неприятните ефекти, които се получават, когато рендерираме сцена с много фин детайл, с големина под тази на единичен пиксел
 - Например, ако имаме чаша, пълна със червени, зелени или сини топчета, и я отдалечим толкова далече, че да стане голяма колкото един пиксел от кадъра
 - Принципно, трябва да оцветим този пиксел с тъмно сиво $(R + G + B)/3$. На практика, при трасирането на лъча, ще попаднем на някое конкретно топче – червено, зелено или синьо.
 - Сгрешения цвят на пиксела е пример за aliasing

Anti-aliasing

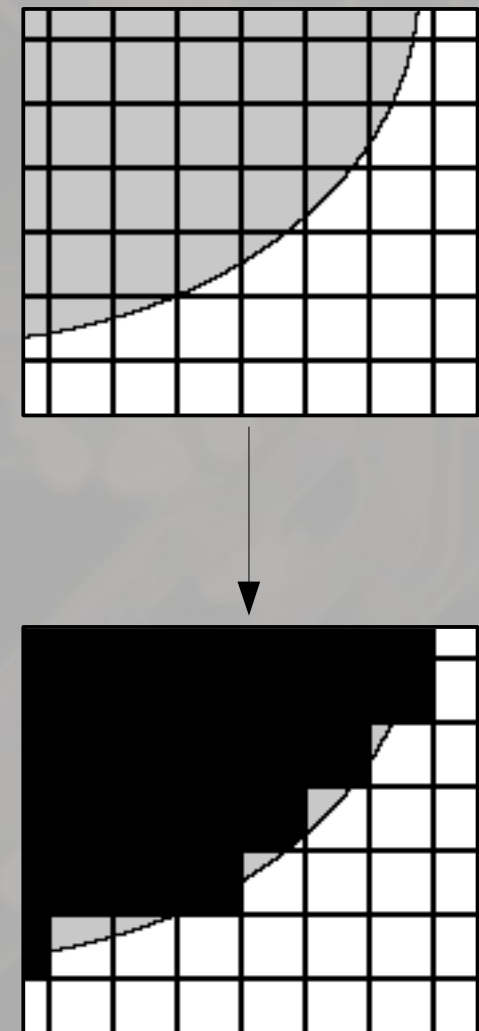
- Лъчите в raytracing-а нямат обем, те са безкрайно тънки, и винаги „удрят“ точно един обект. Междинно положение няма
 - Има и разработки, при които лъчите се представят като пирамиди, и при пресичане, цвета се смята аналитично. Този подход, обаче, е много сложен и рядко ползван на практика
- Често срещан пример за aliasing: резките ръбове на обектите

Anti-aliasing



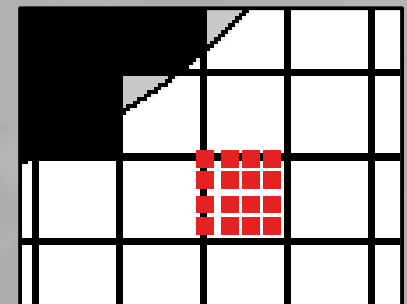
Anti-aliasing

- Причината за твърдите ръбове е, че даден лъч удря или сферата, или пода. Но в един пиксел може да имаме и от двете
- В нашия рейтрейсър, лъчите се изстрелват през горния ляв ъгъл на всеки пиксел
- Но ние може да ги изстреляме от където си искаме, със sub-pixel точност
 - (Camera::getScreenRay() приема 2 double-a!)

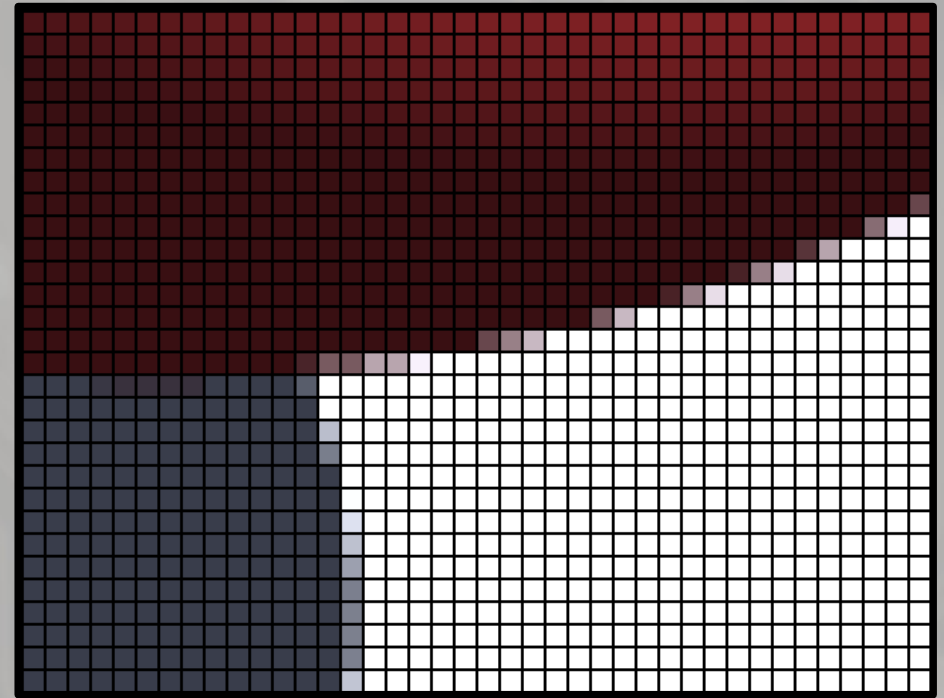
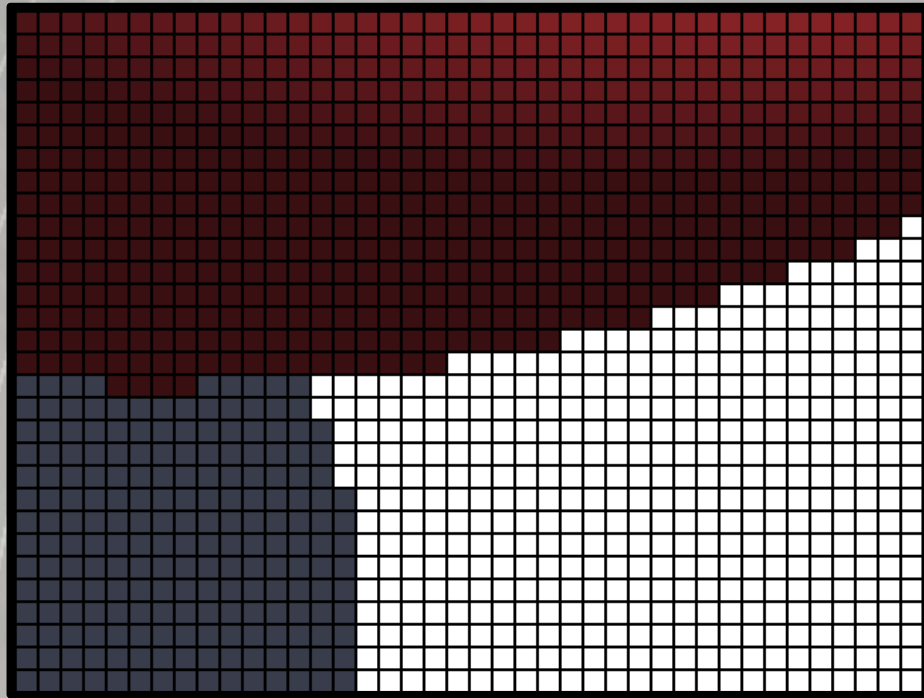


Anti-aliasing

- Идея №1: Supersampling
 - Ще изстреляме повече от един лъч на пиксел и ще усредним резултатите
 - Отделните резултати се наричат семпли (sample)
 - Бройката и разположението на семплите в един пиксел се нарича Anti-aliasing kernel. Най-простия kernel е равномерно разпределение, например 4x4 семпъла в един пиксел:



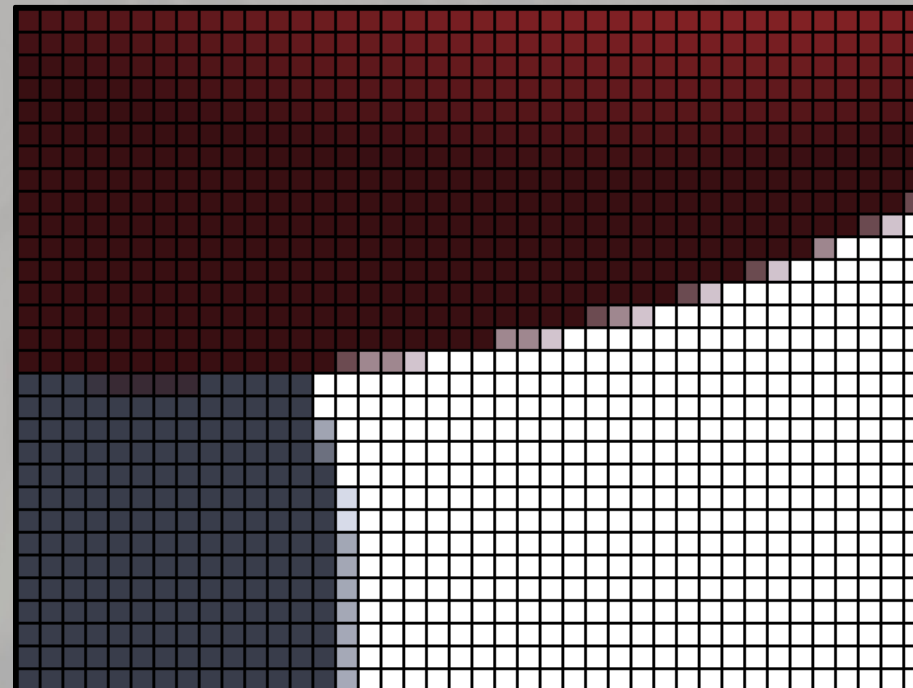
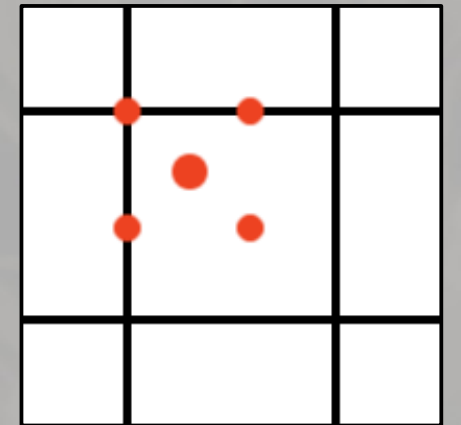
Anti-aliasing



- Ляво: No-AA, rendertime = 0.2s
- Дясно: 4x4 AA, rendertime = 3.1s

По-бърз anti-aliasing

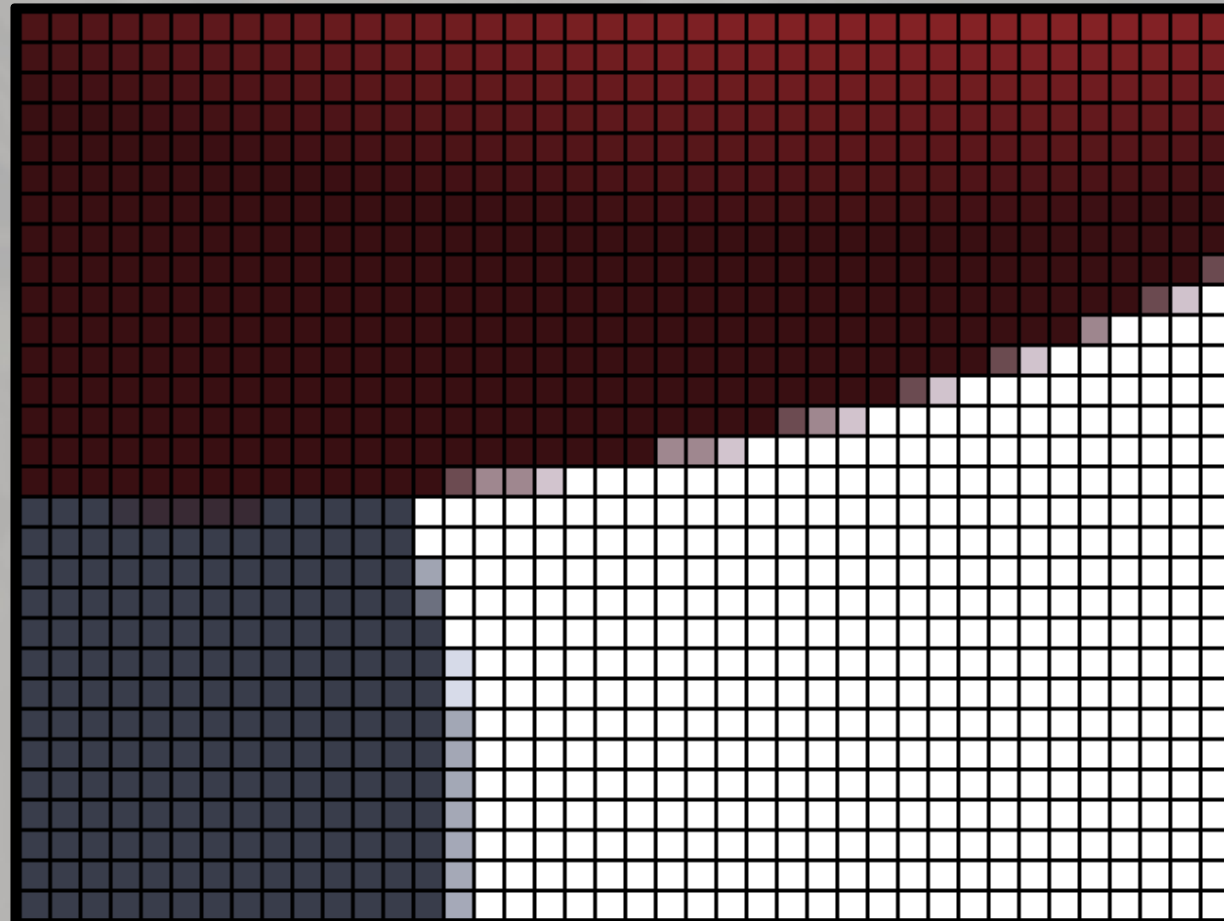
- Идея №2: може да ползваме не толкова тежки kernel-и. Например, следният kernel (quincunx) е доста добър
 - rendertime 0.9s



Още по-бърз Anti-aliasing

- Anti-aliasing-ът е нужен основно по ръбовете
- Идея №3: можем да направим един кадър без никакъв anti-aliasing, след което да минем по него и да засечем ръбовете
 - Чрез цветовата разлика на 4 съседни пиксела – ако е над даден праг, приемаме, че е ръб, и се нуждаем от anti-aliasing.
- Т.е., имаме адаптивен anti-aliasing, като числото, задаващо този праг, определя „качеството“

Адаптивен Anti-aliasing



- Rendertime: 0.3s, праг: 0.1

Model трансформация

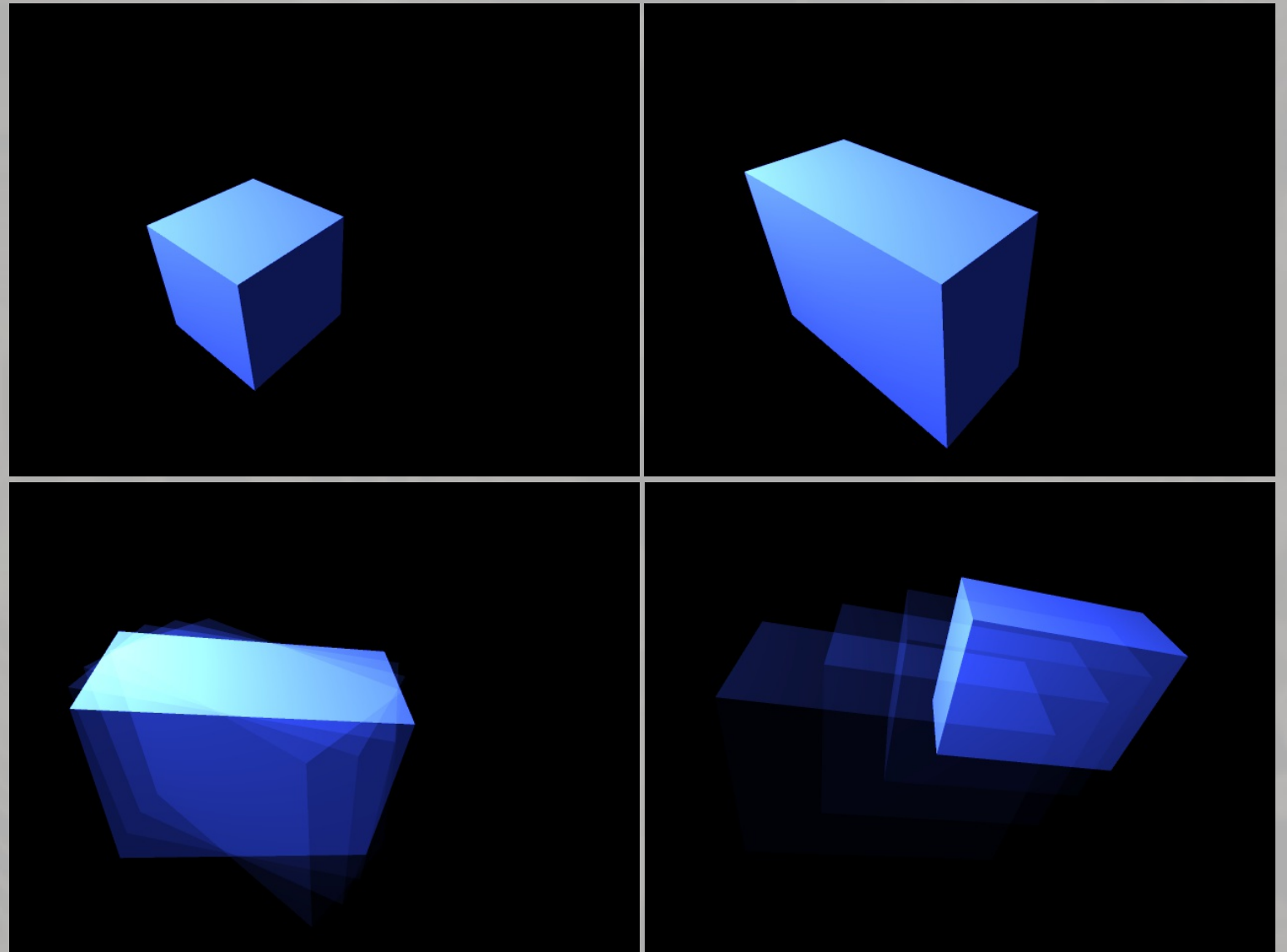
- При пресичането на лъч с геометрия, често е много по-удобно геометрията да е в „каноничен“ вид; например:
 - Куб със страни, успоредни на координатната система
 - Цилиндър с основи, успоредни на XZ
 - Чайник, лежащ на основата си, с чучурче в посока +X
- На практика, обикновено искаме да поставим предметите в не-канонично положение и ориентация
 - Например, паралелепипед, завъртян около Z с 31°
 - Разтеглена сфера (елипсоид)

Model трансформация

- Пресичането с „неканоничните“ обекти е много трудно, дори невъзможно
- Тук се намесва моделната трансформация
- Ще си представим, че имаме едно канонично пространство, в което геометрията е в каноничен вид. Имаме и трансформация T , която прехвърля геометрията в истинското пространство
 - T е комбинация от мащабиране, ротиране и трансляция, в този ред

Пример за Model трансформация

- Каноничен обект
- Мащабиране
(2.0, 1.5, 0.9)
- Ротация
(0.8, 0.1, 0.2)
- Транслация
(3.3, 0.0, 18.1)



Model трансформация

- Вместо да пресичаме трансформираният обект с истинския лъч, може да пресечем не-трансформираният обект с лъч, подложен на обратната трансформация, T^{-1}
- T.e., $intersect(o \cdot T, r) \approx intersect(o, r \cdot T^{-1})$
- Пресечната точка, която ще намерим, е „правилната“, но в каноничното пространство; необходимо е после да я прехвърлим (заедно с нормалата и прочее) обратно в реалното пространство

Model трансформация

- Алгоритъм:

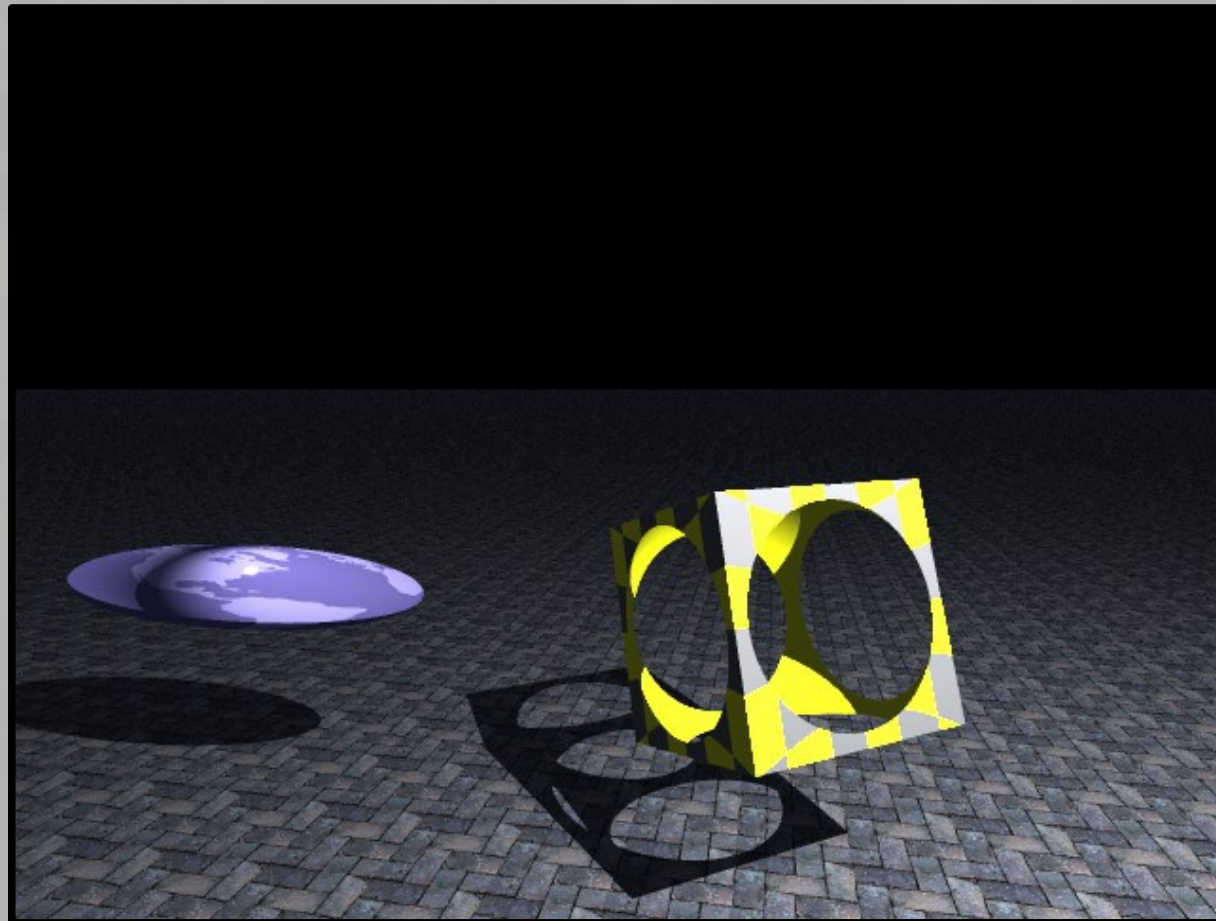
```
function Node::intersect(ray, info)
    t_ray = ray * T-1
    geometry->intersect(t_ray, info)
    info = info * T
```

- Не всички членове на info трябва да бъдат трансформирани: само позицията, нормалата¹, и дължината до пресичането
 - ¹ ако в трансформацията участва скалиране, за нормалата трябва да вземем специални мерки

Model трансформация

- Използването на model трансформации ни дава много **ВЪЗМОЖНОСТИ**
 - Чрез скалиране, от сферата може да получим елипсоид, от куба – паралелепипед
 - Произволна ориентация на всички примитиви (например равнината) – със запазване на коректните uv-та (те не се трансформират)
 - Два различни Node-а могат да ползват една и съща геометрия, но различни трансформации (instancing)

Model трансформация



Model трансформация

- Трансформацията (Transform) е свойство на Node-а
- Вече пресичанията не са с node->geometry, а със самото node. То взима в предвид трансформацията си
- Така може лесно да се реализира гореспоменатия instancing

