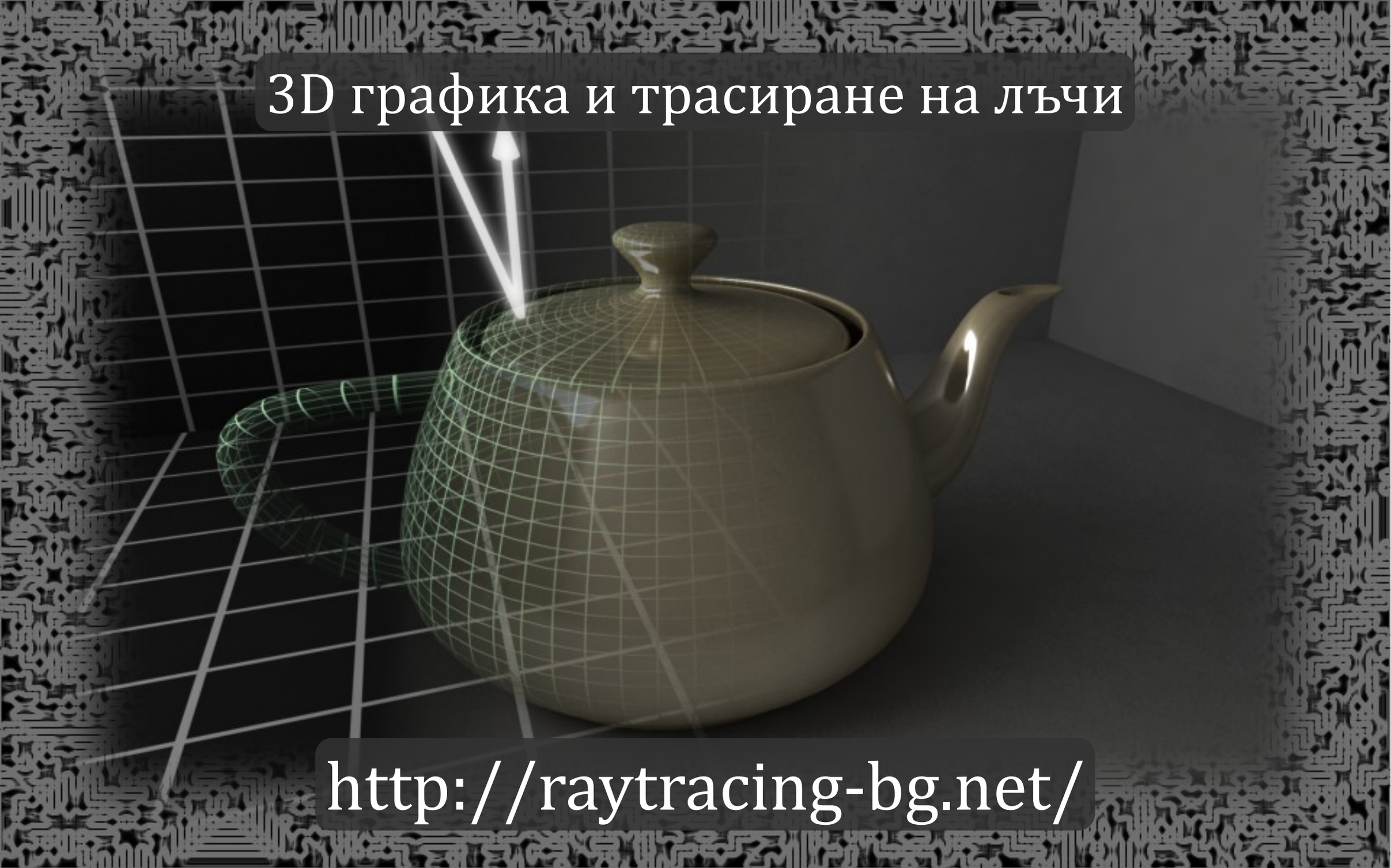
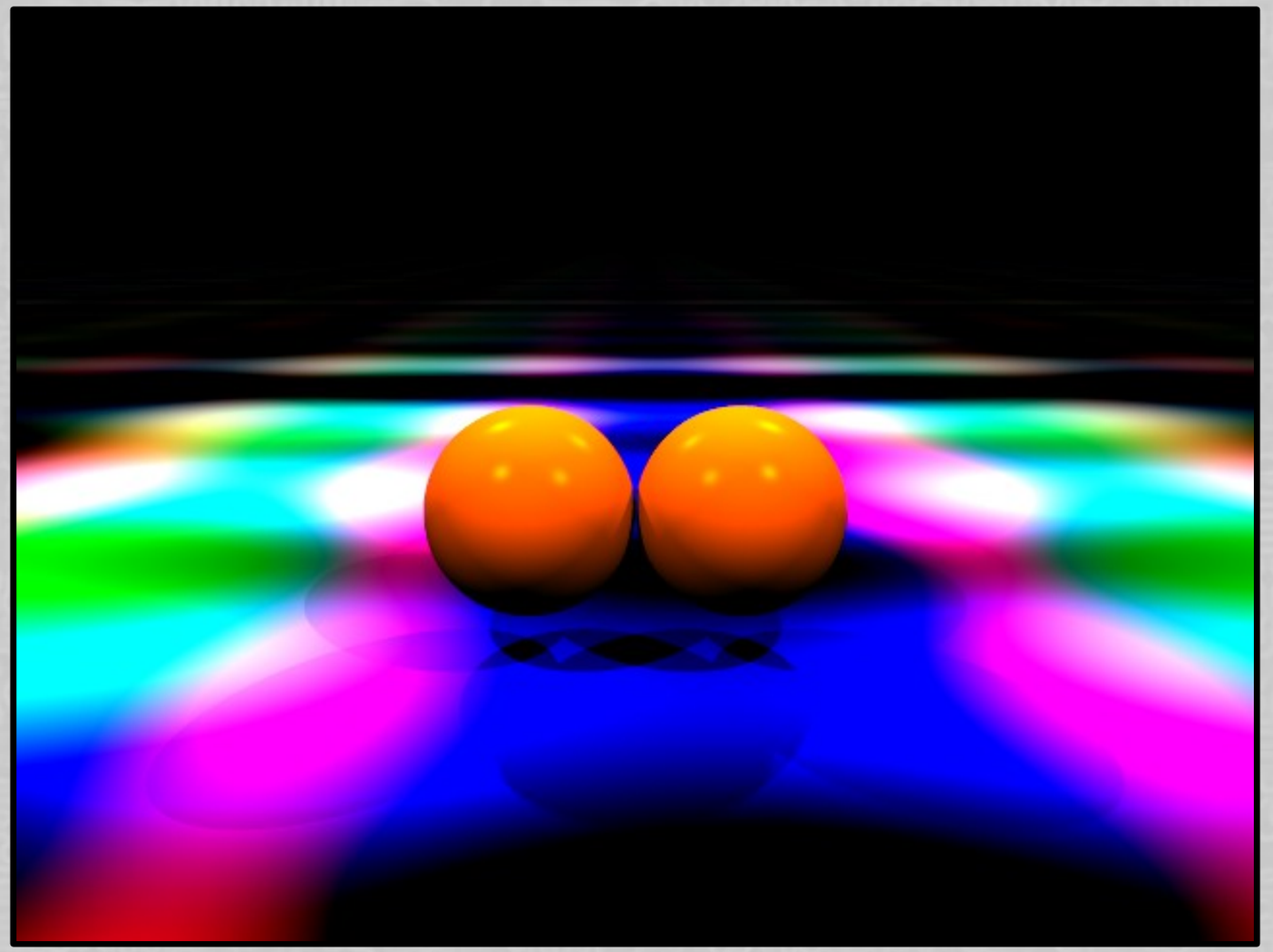
3D графика и трасиране на лъчи



http://raytracing-bg.net/

Best of homeworks :)



Тема 5

Пресичане с по-сложни примитиви
Булеви операции
Ambient цвят, текстури
Antialiasing
Model трансформация

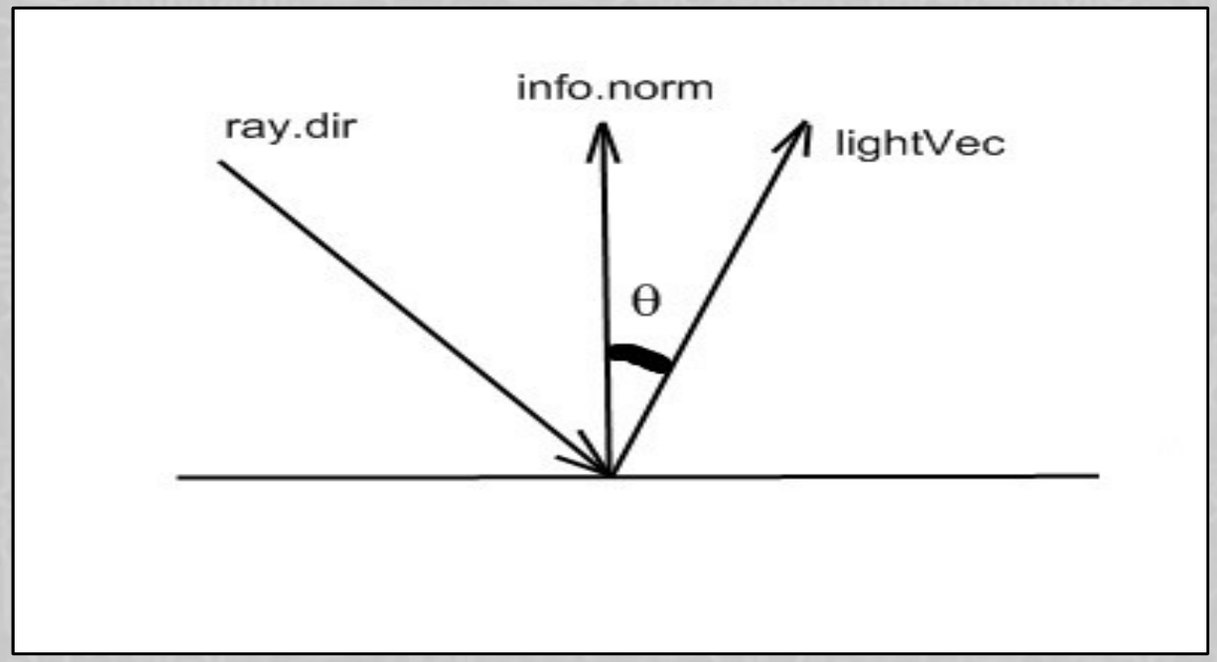
Съдържание

- Bugfixing (от предната лекция)
- Пресичане с куб
- Булеви операции
  - Обединение
  - Сечение
  - Разлика
- Ambient цвят

Съдържание (2)

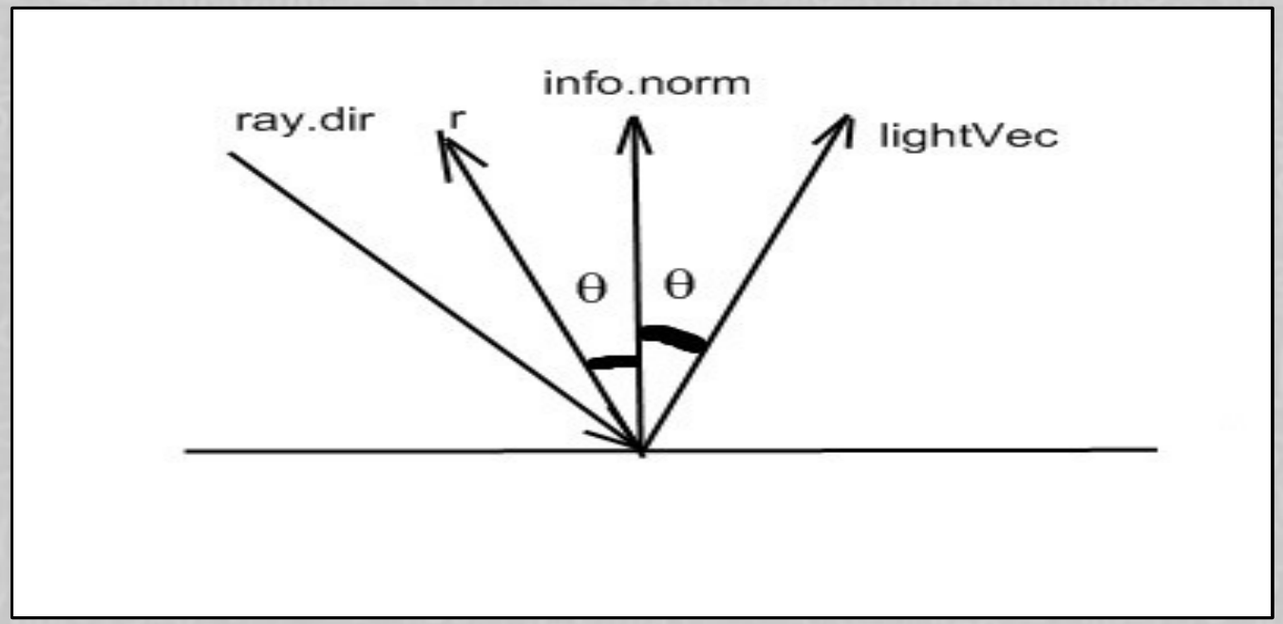
- Bitmap текстури
- Заглаждане на ръбовете (anti-aliasing)
- Model трансформация

Ламберт



- OutColor = materialColor * max(0, cos(theta)) * lightColor
- cos(theta) = dot(lightVec, info.norm)

Фонг (Phong)



- R е отражението на lightVec спрямо нормалата info.norm
- OutColor = pow(dot(-ray.dir, r), exponent) * lightColor
- Exponent контролира „колко лъскав е материала“

Пресичане с куб

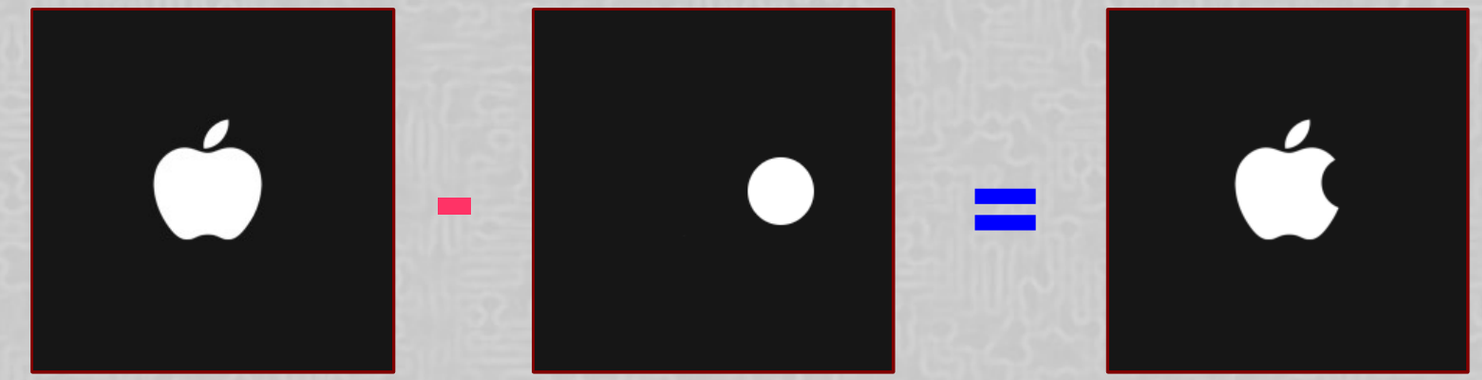
- За всяка страна приемаме, че пресичаме с равнина...
  - ... и проверяваме дали пресечната точка е в граници по другите две измерения
  - Подобно на пресичането със сферата, трябва да внимаваме за стени, намиращи се зад камерата
    - т.е. търсим най-близката положителна пресечна точка.
- Нормалите 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

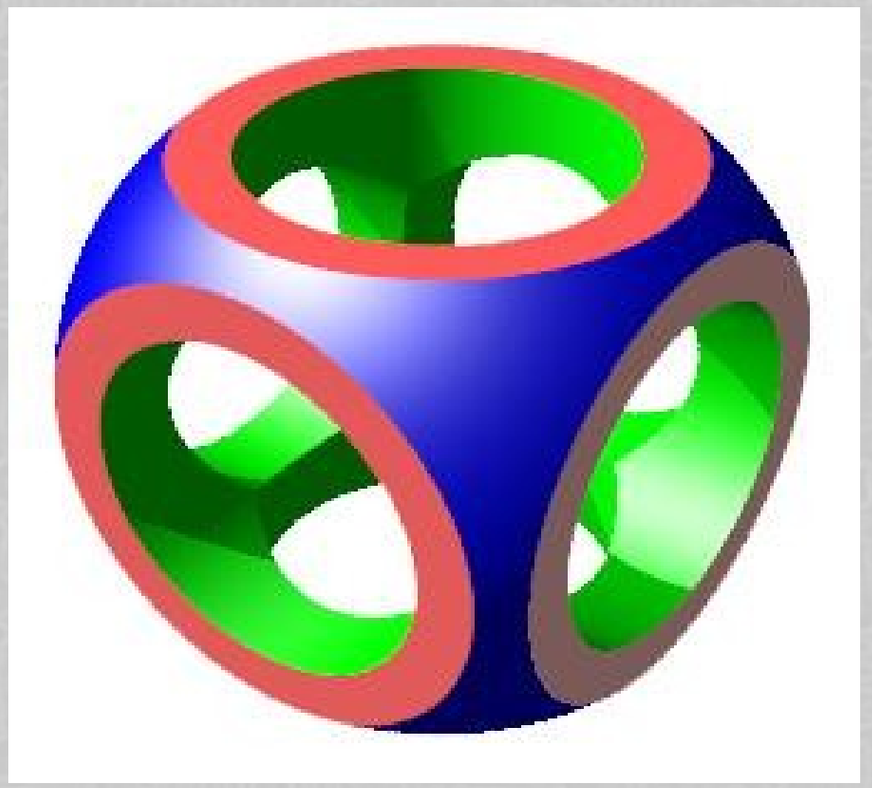
- UV:
  - (u, v) := (x + y, z) – съвпада по ръбовете, но разтегля текстурата по две от стените
  - (u, v) := 2*(p.x – cube.x, p.y – cube.y) / cube.side (за стена +/-Z)
    - Разкъсва текстурата по ръбовете, но изглежда добре
  - Сферични координати:
    - (u, v) := (atan2(p.z – cube.z, p.x – cube.x),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

Булеви операции

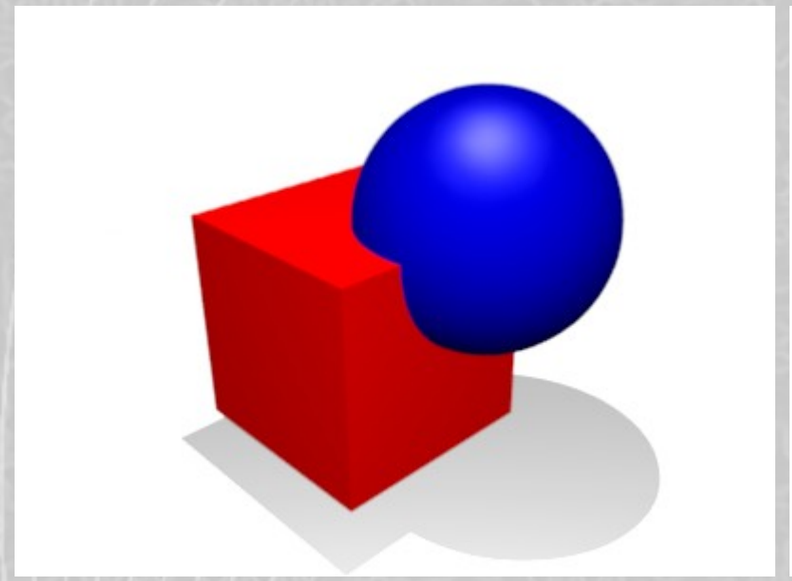
- Често е полезно да създадем нов 3D обект, като комбинираме два съществуващи, и се интересуваме от някоя част на двата обекта, която да изпълнява някакво логическо (булево) условие
- На английски: CSG = Constructive Solid Geometry

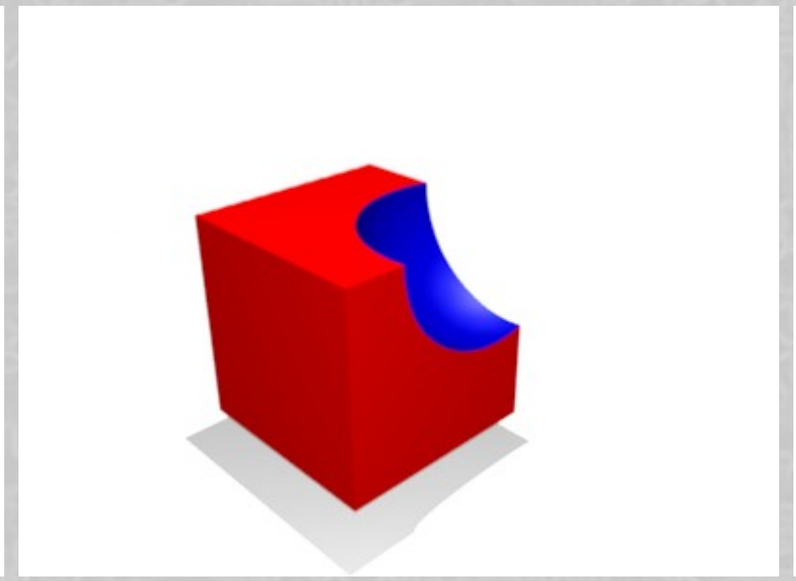


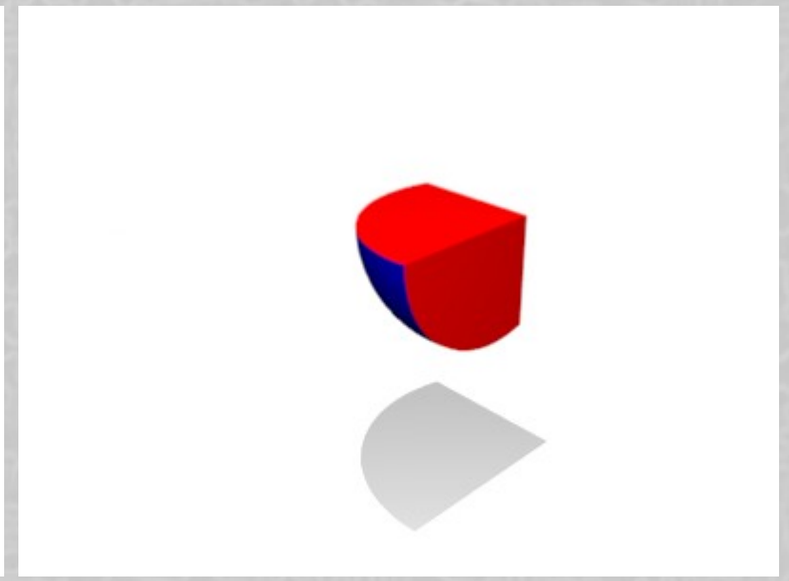
Булеви операции



Булеви операции





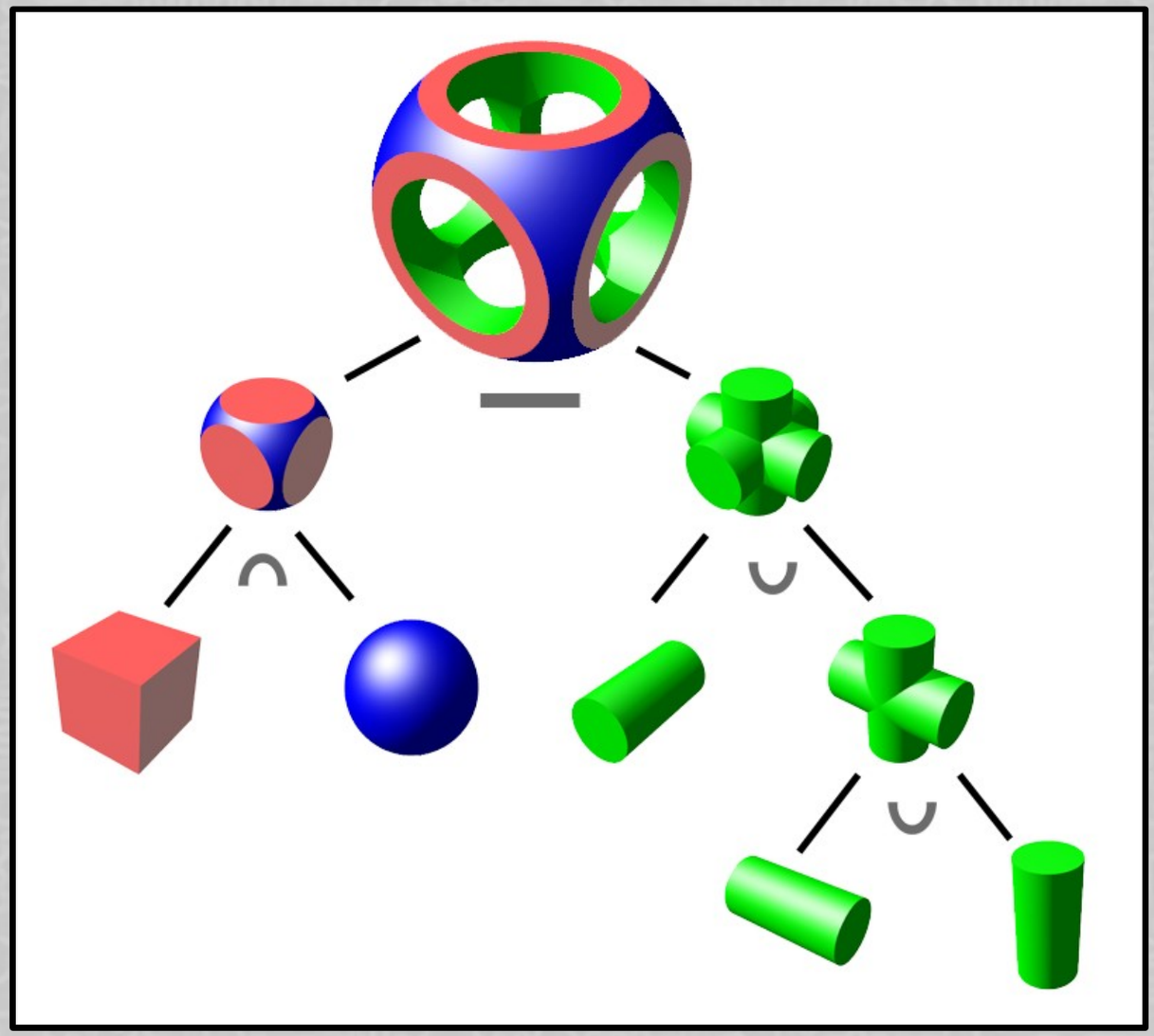


Apple's logo

Булева операция между два обекта

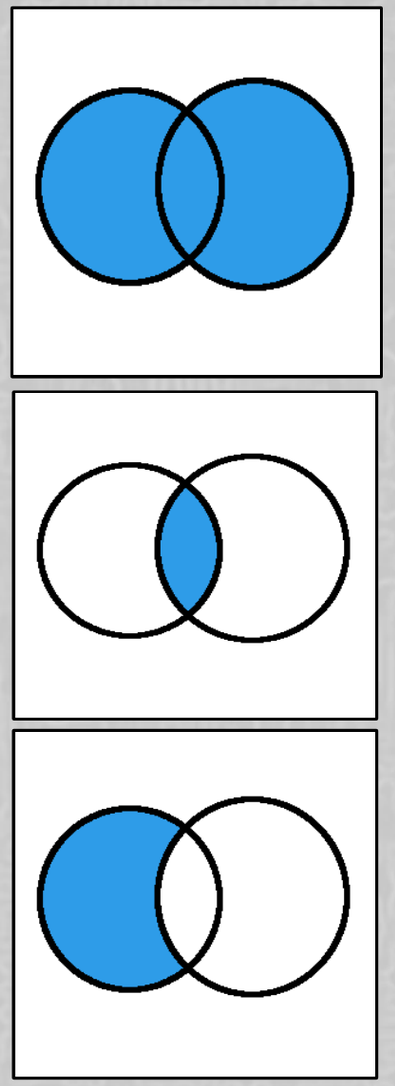
- Алгоритъмът за булеви операции е специфичен за Raytracing-а. При Z-buffer, например, аналогичен алгоритъм няма; булевите операции там се реализират много по-трудно
- Булевата операция между два обекта е фундаментална; при повече от един обект, може да реализираме дървовидна структура (двоично дърво), 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 вътре_сме_в(A) && не_сме_вътре_в(B)



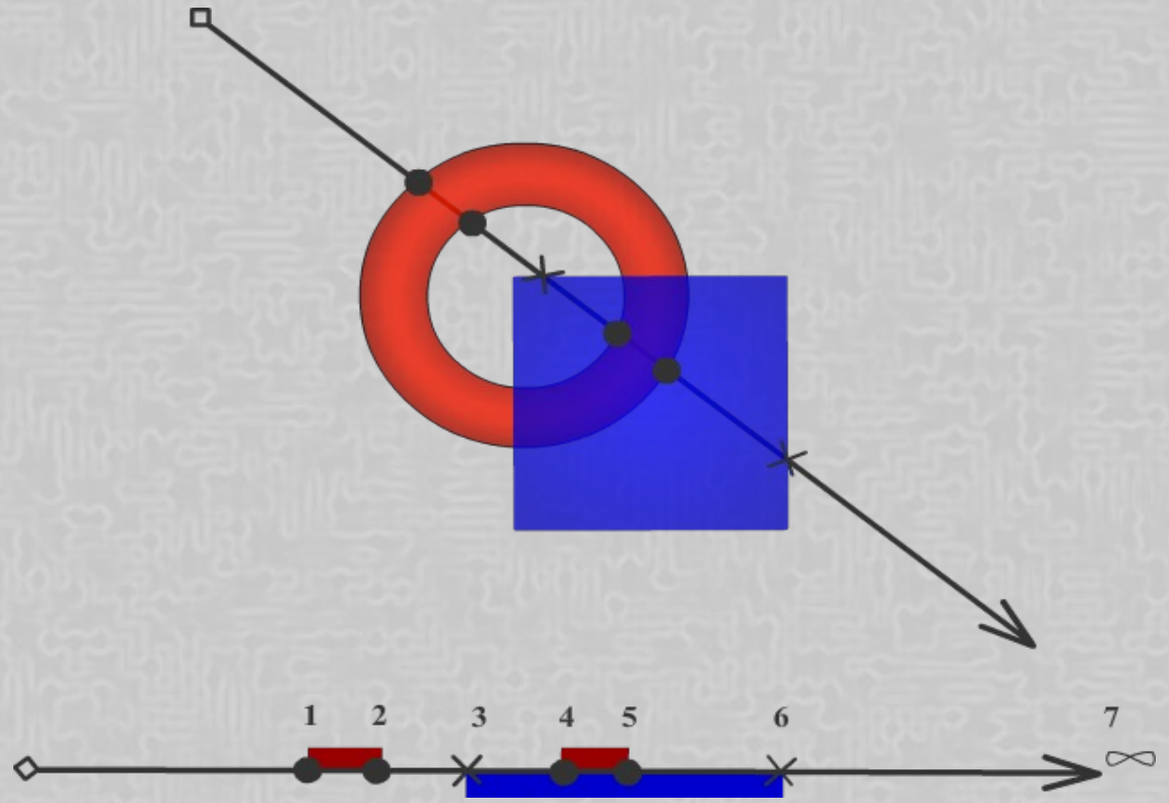
Алгоритъм за CSG с две геометрии

- 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 сме 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 := normalize(intersection.pos – 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*R > 0
- Backface culling: това е оптимизация, при която пропускаме (части от) обекти, за които N*R < 0

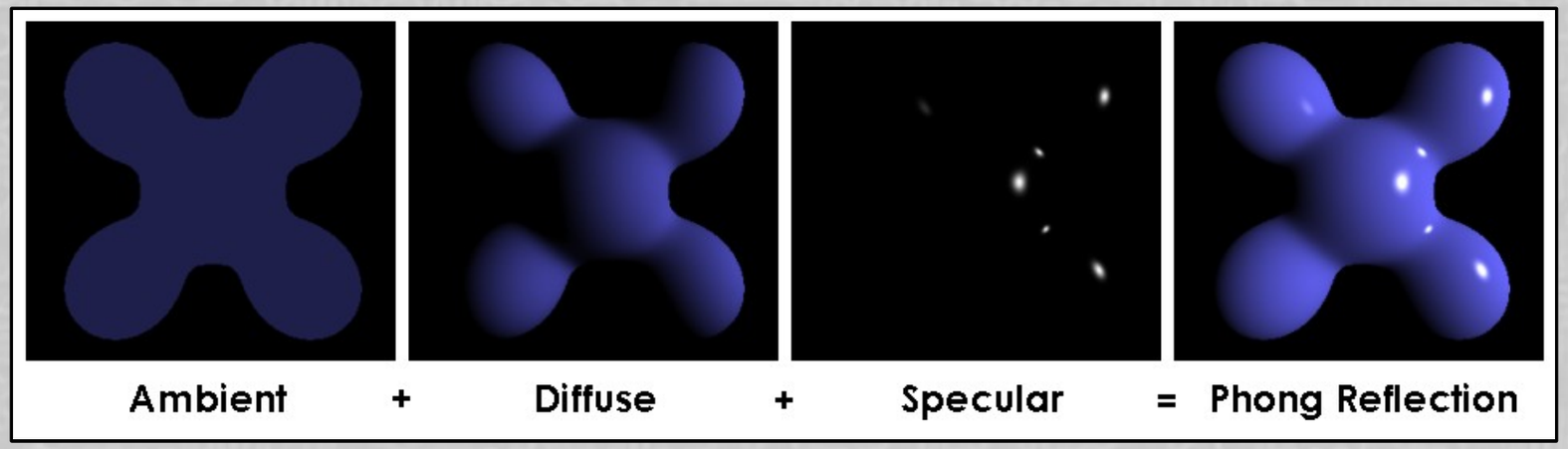
Нормали

- 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 ($R * N_c >= 0$)
    - 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.

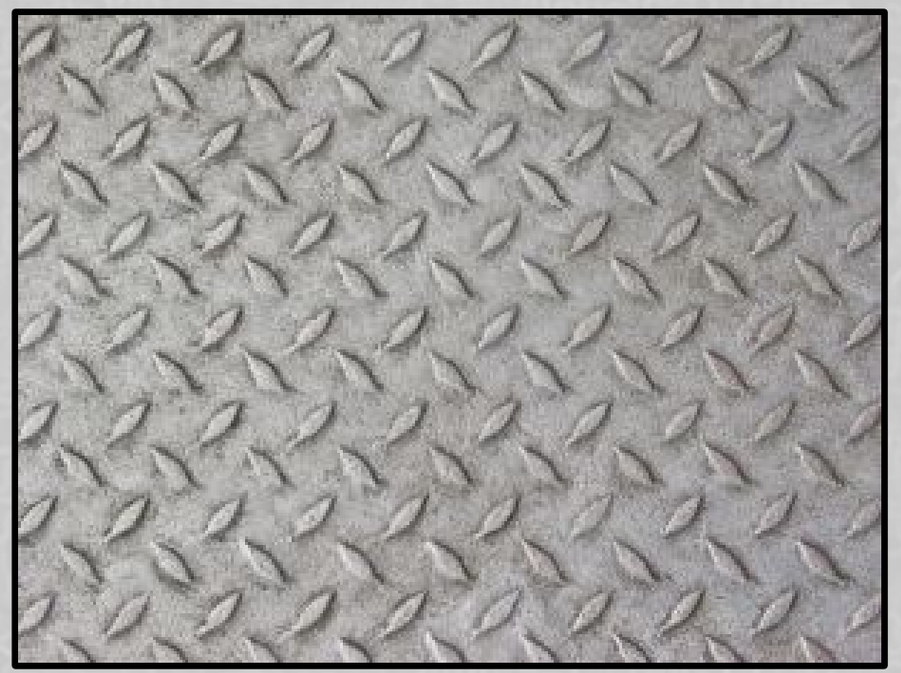
Ambient ≠ Global Illumination

- 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!

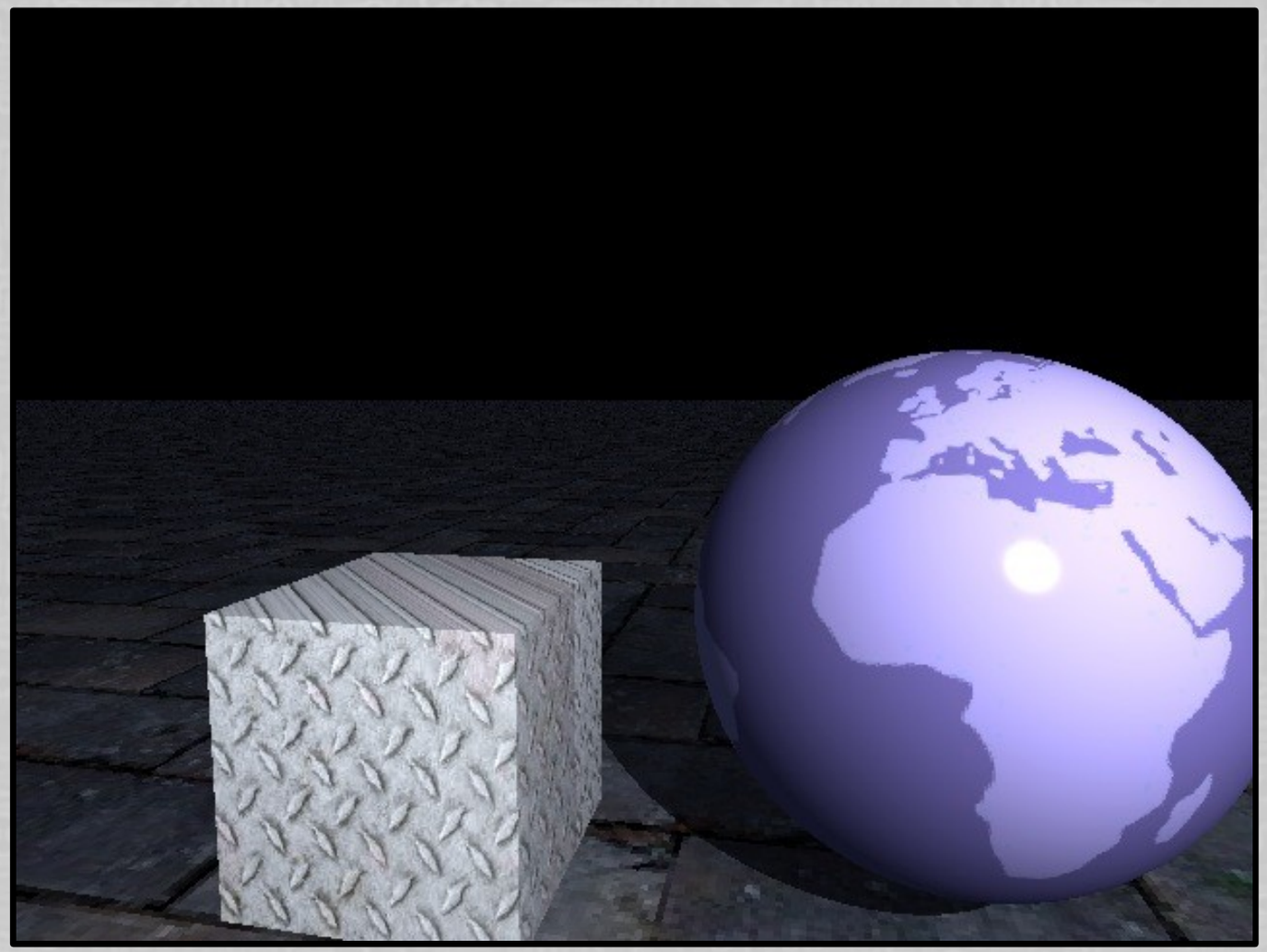
Color bleeding

Bitmap 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



Bitmap 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
  - $v = 1.0 - (\pi/2 + \text{asin}((\text{info.p.y} - \text{sphere.y}) / R))/\pi$

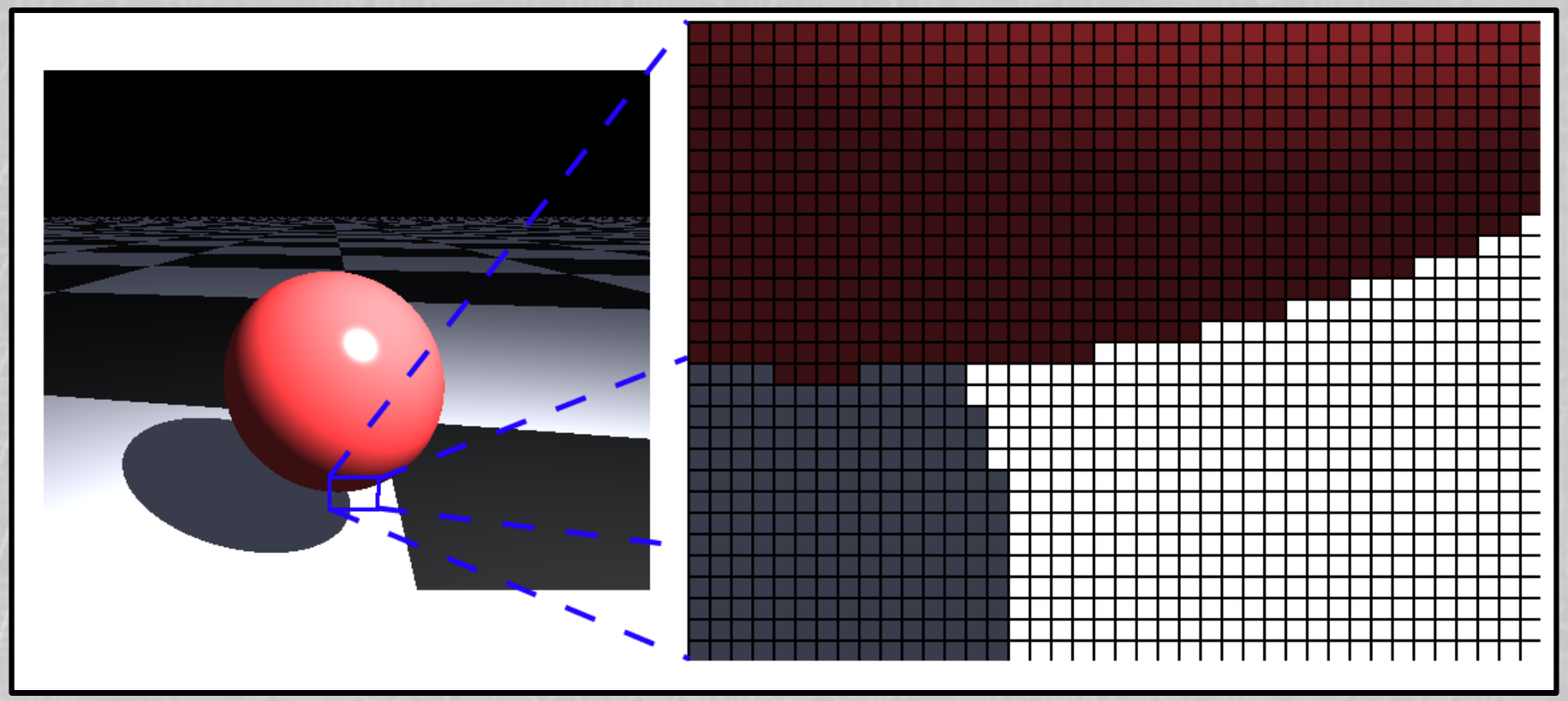
Anti-aliasing

- Под „aliasing“ ще разбираме 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

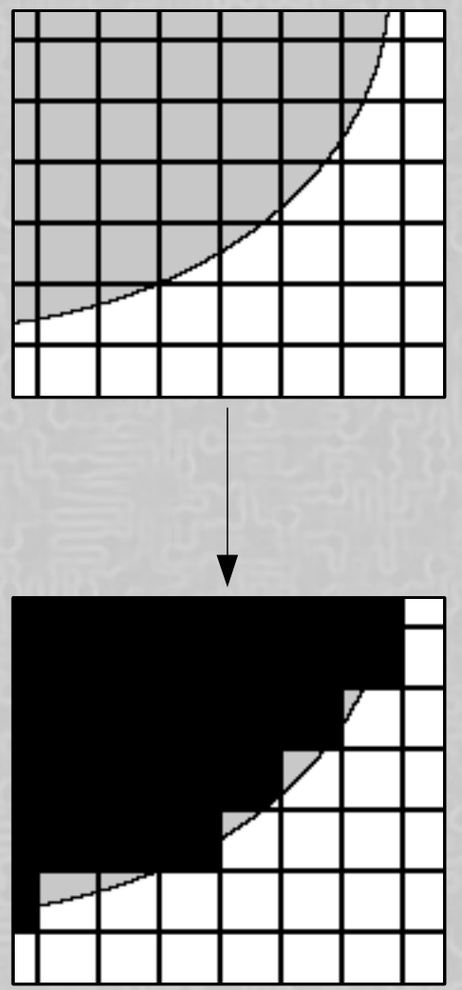
- Лъчите в raytracing-а нямат обем, те са безкрайно тънки, и винаги „удрят“ точно един обект. Междинно положение няма
  - Има и разработки, при които лъчите се представят като пирамиди, и при пресичане, цвета се смята аналитично. Този подход, обаче, е много сложен и рядко ползван на практика
- Често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

Anti-aliasing



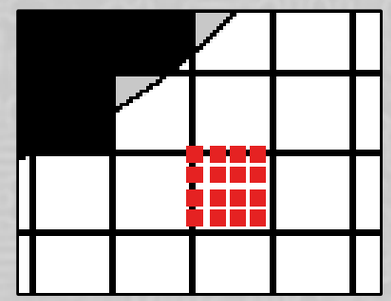
Anti-aliasing

- 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-а!)

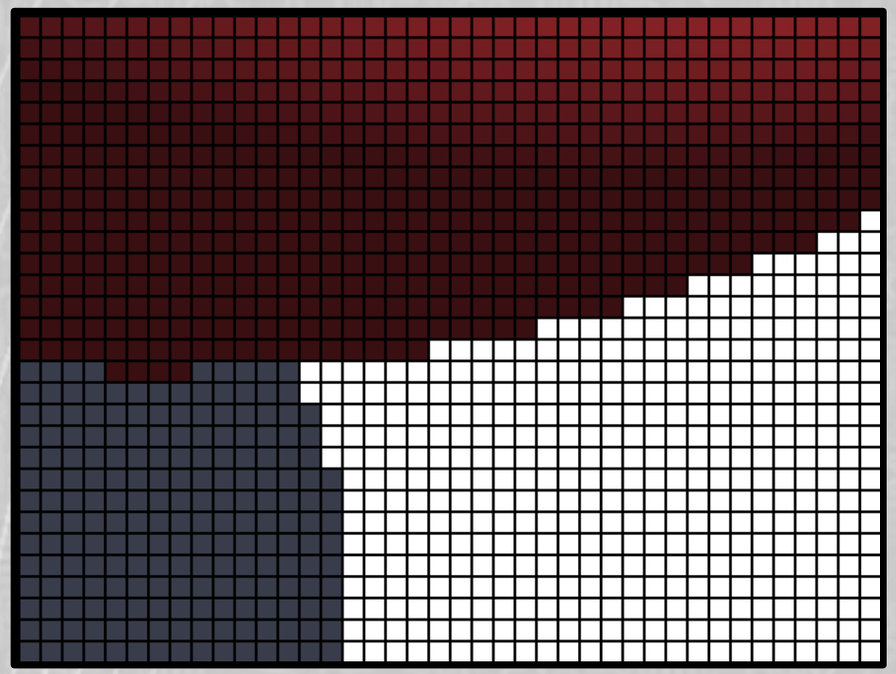


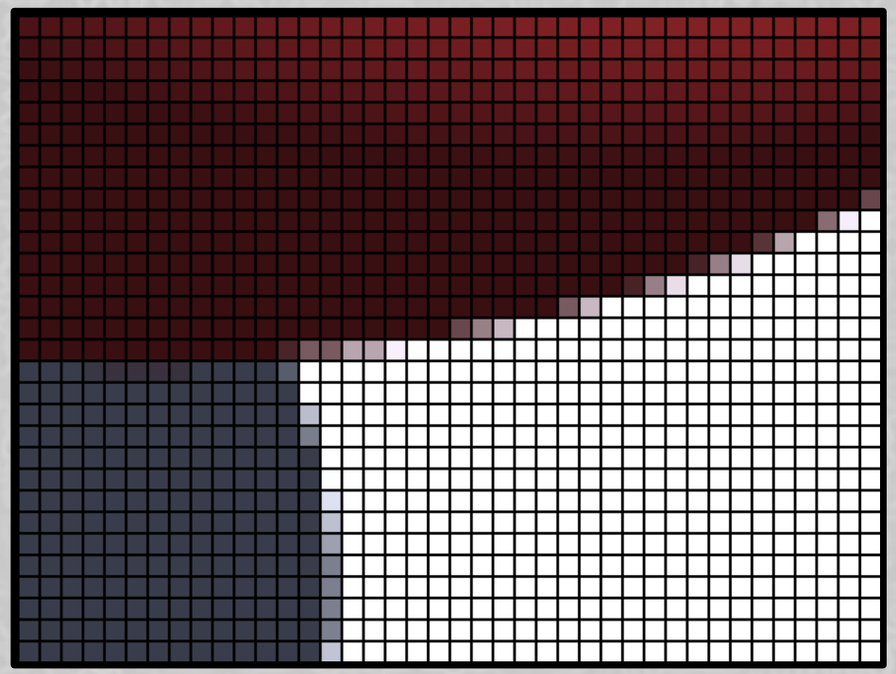
Anti-aliasting

- 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

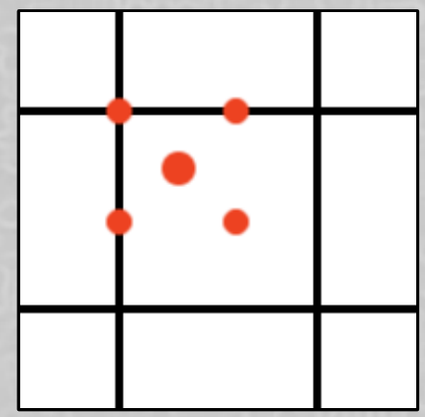


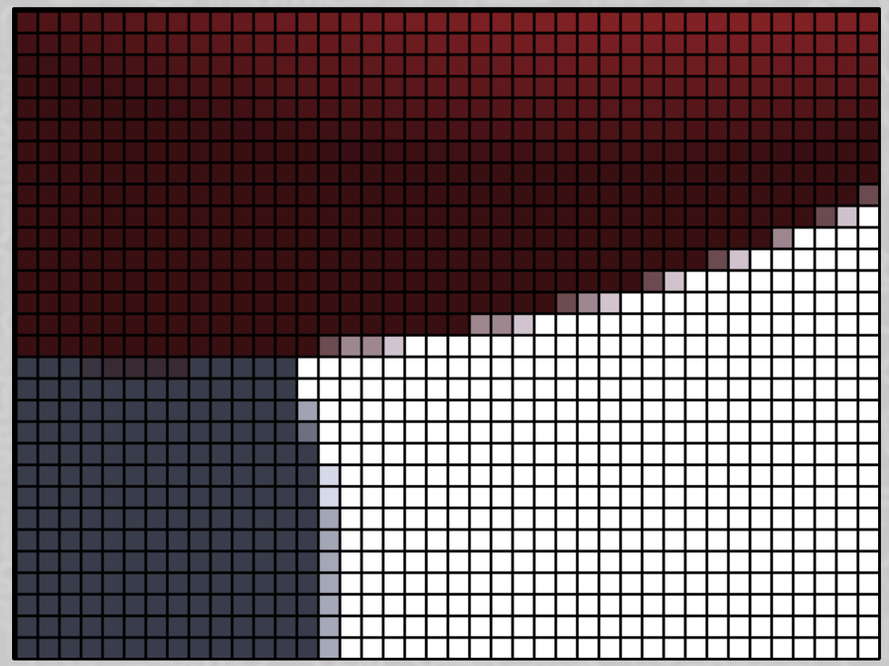


- Ляво: No-AA, rendertime = 0.2s
- Дясно: 4x4 AA, rendertime = 3.1s

По-бърз anti-aliasing

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

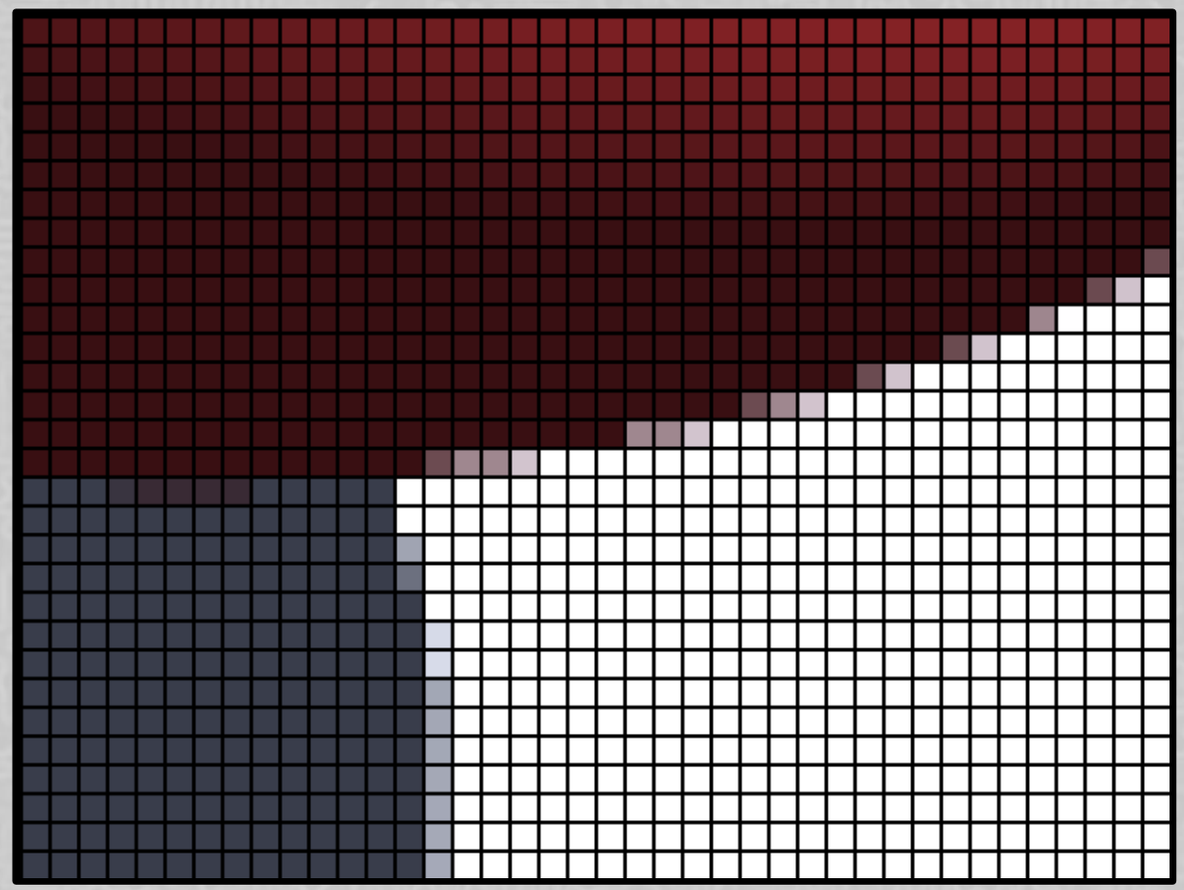




Още по-бърз Anti-aliasing

- 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

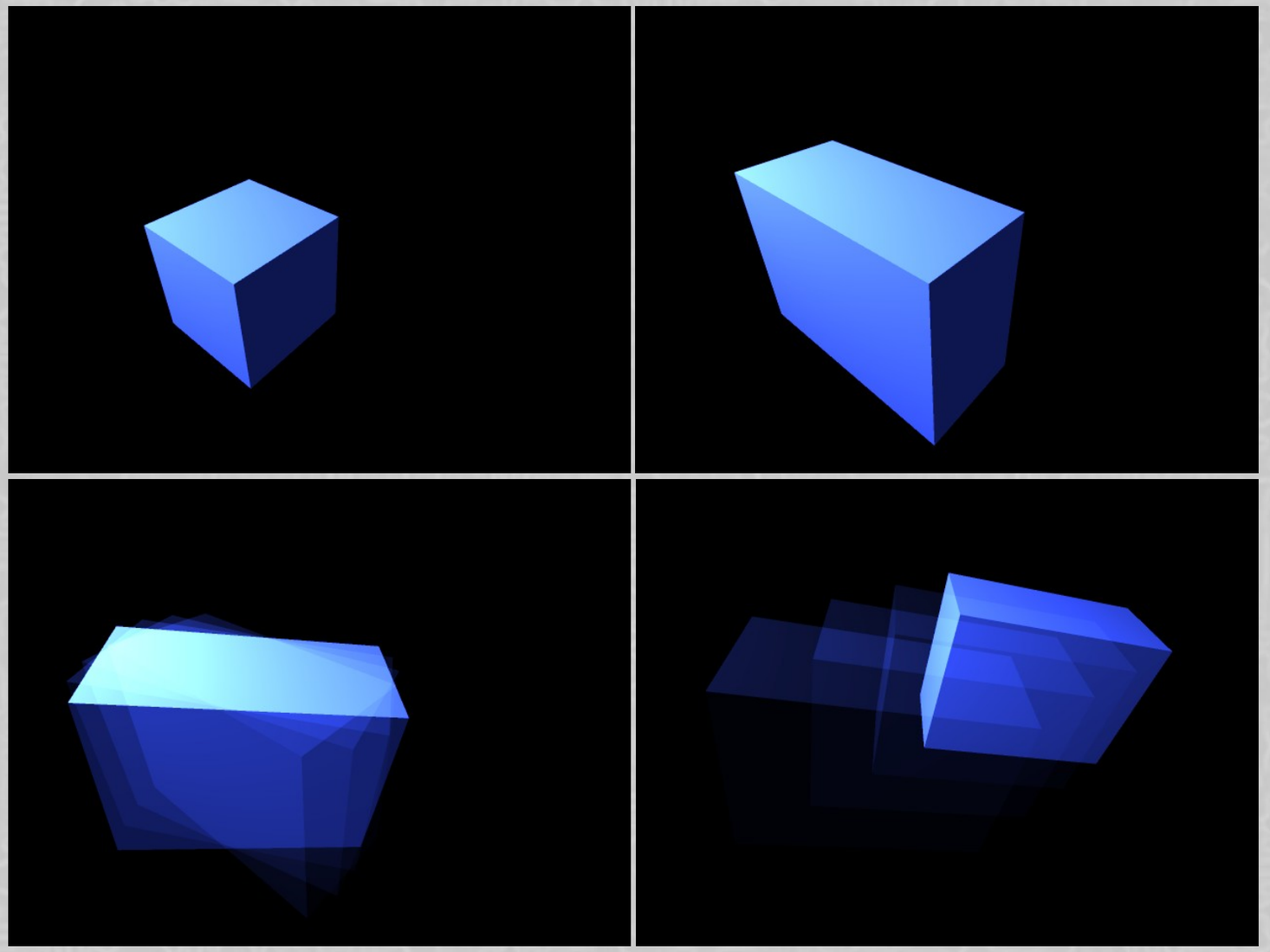
- При пресичането на лъч с геометрия, често е много по-удобно геометрията да е в „каноничен“ вид; например:
  - Куб със страни, успоредни на координатната система
  - Цилиндър с основи, успоредни на XZ
  - Чайник, лежащ на основата си, с чучурче в посока +X
- На практика, обикновено искаме да поставим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

Model трансформация

- Пресичането с „неканоничните“ обекти е много по-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 транслация, 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}
- Т.е., $\texttt{intersect}(T \cdot o, r) \approx \texttt{intersect}(o, T^{-1} \cdot r)$
- 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:

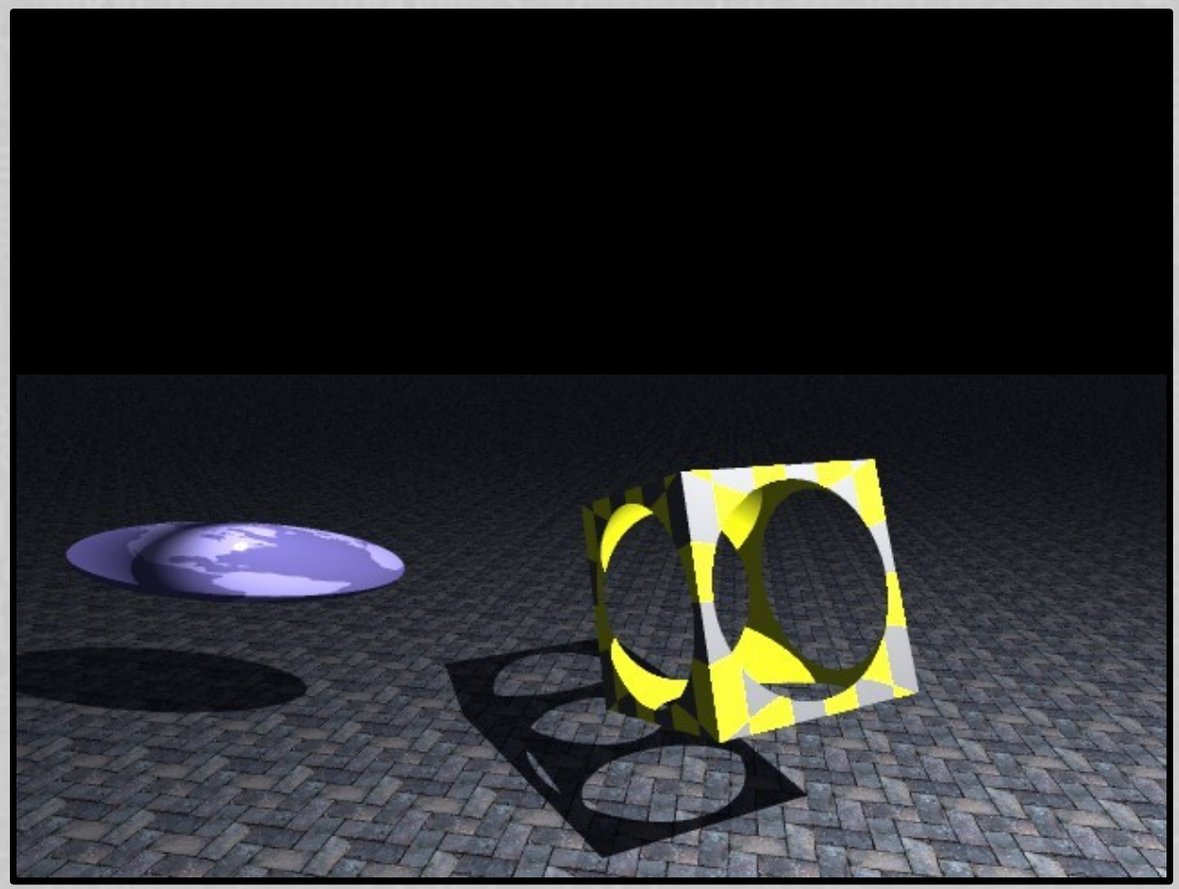
```
function Node::intersect(ray, info)
  t_ray = T^-1 * ray
  geometry->intersect(t_ray, info)
  info = T * info
```

- Не всички членове на info трябва да бъдат трансформирани: само позицията, нормалата, и дължината до пресичането

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