3D графика и трасиране на лъчи v.4.0

http://raytracing-bg.net/

Тема 12

Глобално осветление
Path tracing

Съдържание

- Анонси
- Path Tracing
  - Дискусия
  - Реализация

Анонси

- Честита 2016та година!
- Тест 2 по време на лекция №14, 20ти Януари, 18:20
  - 20 минути, 15 въпроса, върху лекции 8 до 13 включително
  - За тези, които не могат да дойдат, ще има дата и през сесията
- Домашни
- Нови проекти

Нови проекти

- Лесни
  - Ще бъде модифициран cross-eyed viewing
- Средни:
  - Симулация на фотографски филм (нелинеен output mapping)
  - Point-cloud to triangle mesh
- Трудни:
  - Light tracing, BVH

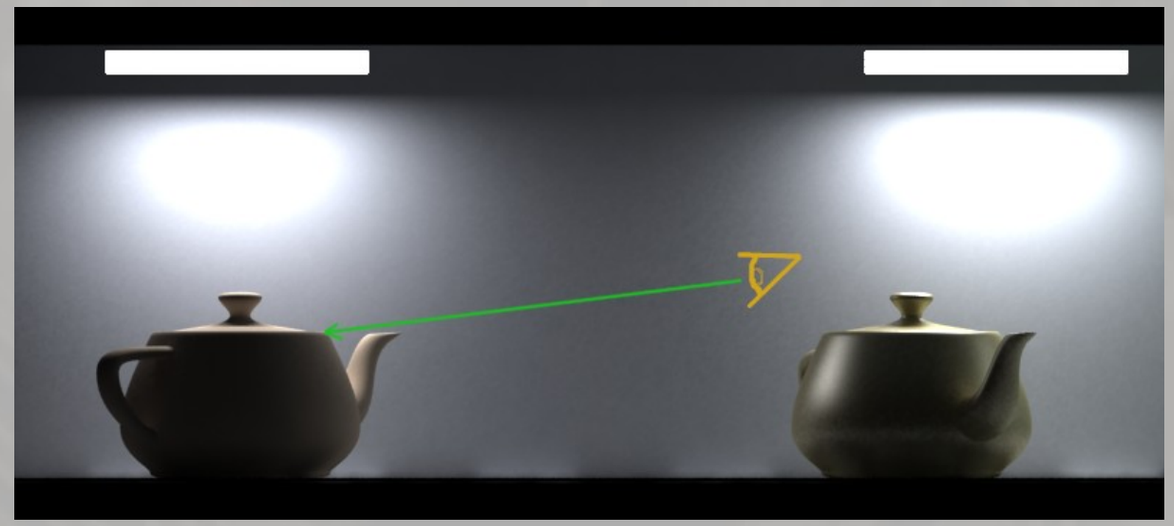
Path tracing, revisited

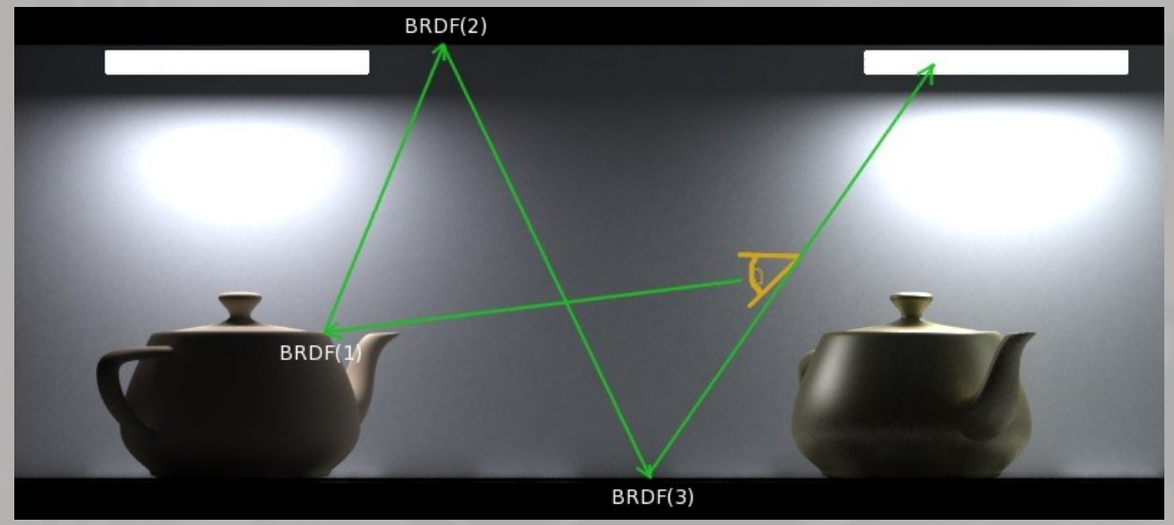
$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_i, \omega_o) L_i(x, \omega_i) (-\omega_i \cdot n) d\omega_i$$

- Path tracing алгоритъма се основава на идеята да трасираме голямо количество *пътища* през сцената, като всеки *път* започва от камерата и завършва в някоя лампа
- Даден *път* представлява начупена линия от прави сегменти, като върховете на тази начупена линия са местата, където лъчите са пресекли геометрията

Path tracing

- Пътят започва от камерата, и при всяко пресичане с геометрия, генерираме нов случаен лъч навън от пресечната точка. Това представлява следващия сегмент от пътя

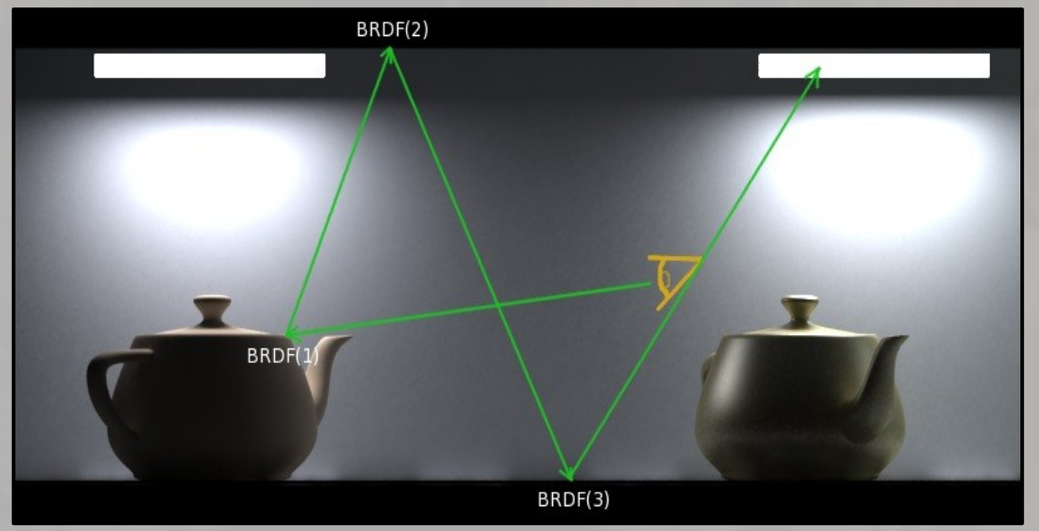




Path tracing

- След като завършим даден път, трябва да пресметнем светлината, която е дошла по този път до камерата
  - В примера, ако светлината допринася с яркост L, то резултатът от този път е L * BRDF(3) * BRDF(2) * BRDF(1)
  - Долната картинка показва пресмятането на BRDF(3):
    - BRDF(3) := $f_r(x, \omega, \omega') * (n \cdot \omega')$

      (както си е по Каджия)



BRDF класа

- Какво искаме от един BRDF?
  - Да пресмята отражателната функция за дадени лъчи (както си е по дефиниция)
  - Да може да генерира лъчи от дадена точка, с разпределението на BRDF-а
- Иначе казано:

```
class BRDF { public:
        virtual Color eval(const IntersectionInfo& x, const Ray& w_in, s Vector& w_out) = 0;
        virtual void spawnRay(const IntersectionInfo& x, const Ray& w_in, Ray& w_out,
                                        Color& color[1], float& pdf[2]) = 0;
};
```

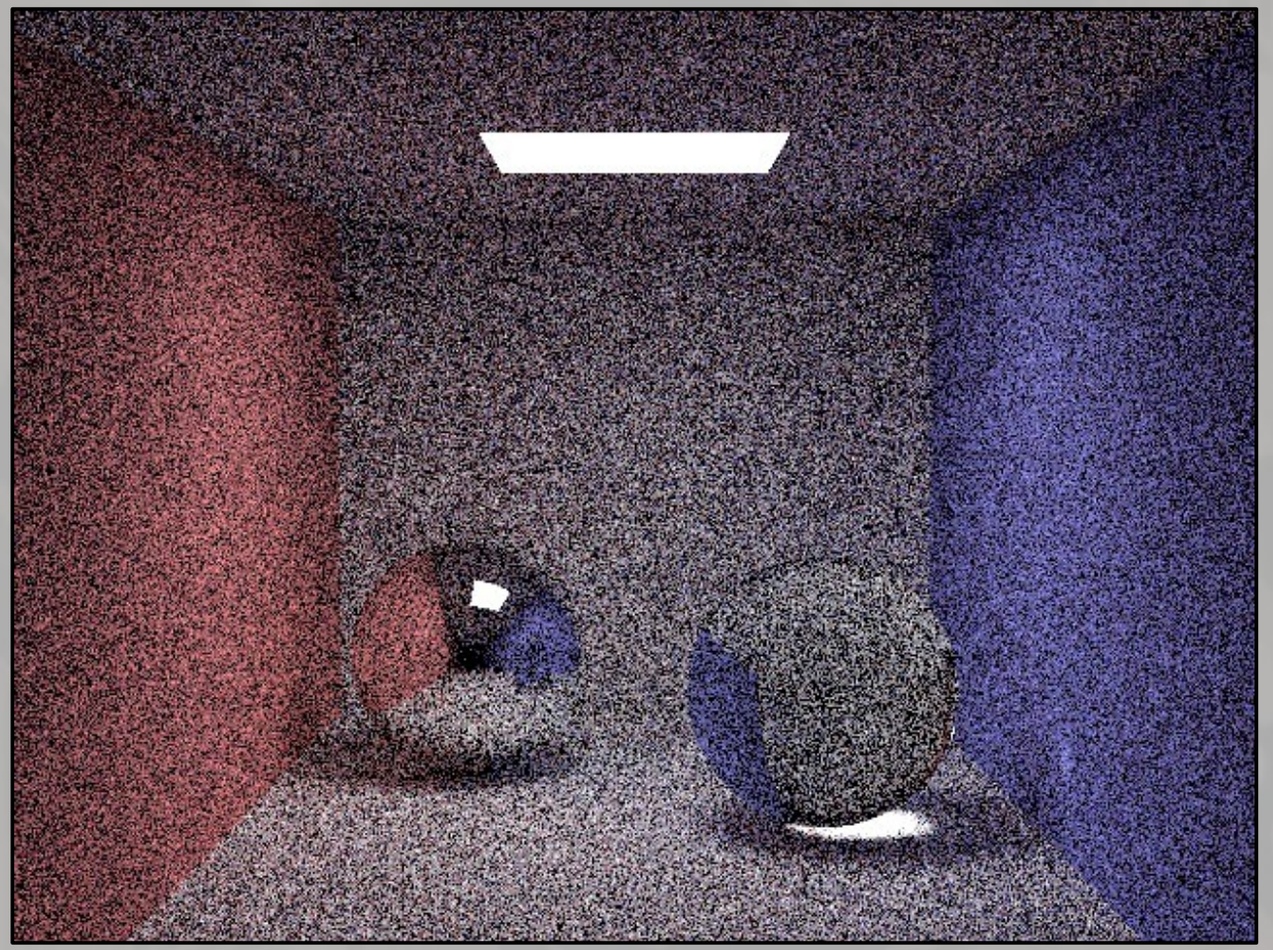
[1] Параметърът ***color*** на spawnRay връща стойността на BRDF-а в ново-генерираната посока. Т.е., това е резултатът от eval(x, w_in, w_out), където w_out е резултатът от spawnRay(). Вкаран е като параметър за удобство и скорост.
[2] Параметърът ***pdf*** на spawnRay връща вероятностната плътност на генерирания лъч. Например, за дифузен BRDF, *pdf* е $1/(2\pi)$ навсякъде, понеже е равномерно разпределена в полусферата над точката, а полусферата е с площ 2π

Path tracing

- Модификации, които ще направим:
  - Няма да пресмятаме горната сметка с BRDF(i) накрая; вместо това, ще си пазим текущото произведение П(BRDF(i)), което просто ще умножим с L, когато ни потрябва
    - Това произведение (ще го именуваме pathMultiplier) може да се пресмята постъпково, заедно със строежа на пътя
    - pathMultiplier указва „важността“ на построения път; ако падне много близко до 0, значи пътят няма да допринесе много за крайния цвят на пиксела, и може да го отхвърлим, прекратявайки по-нататъшен строеж

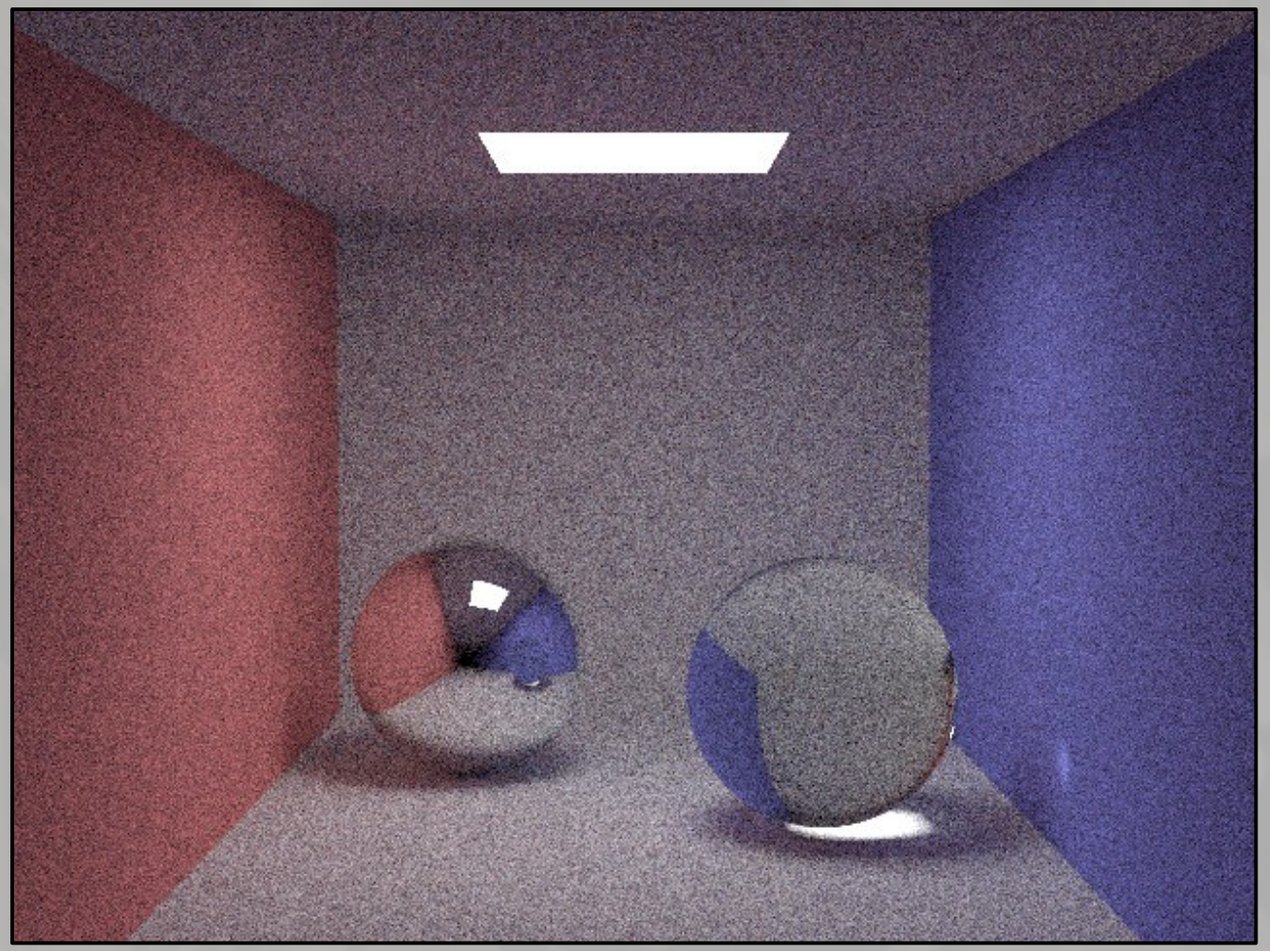
Path tracing

- Дотук описахме класическия алгоритъм на Каджия
  - Поради малката вероятност за уцелване на лампа, той е шумен и бавно сходящ
  - Отдясно: при 40 пътя за пиксел при относително голяма лампа



Path tracing

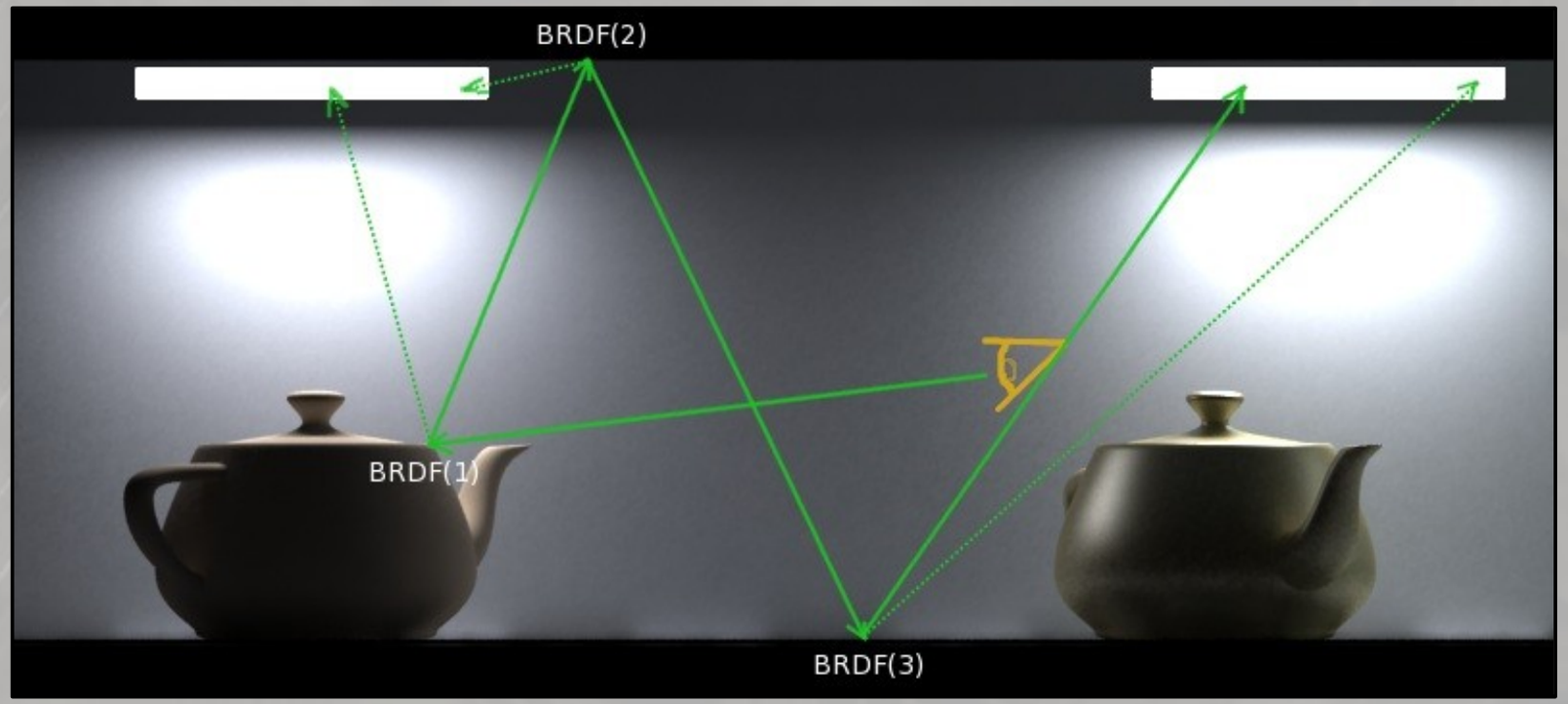
- 256 пътя за пиксел



Path tracing

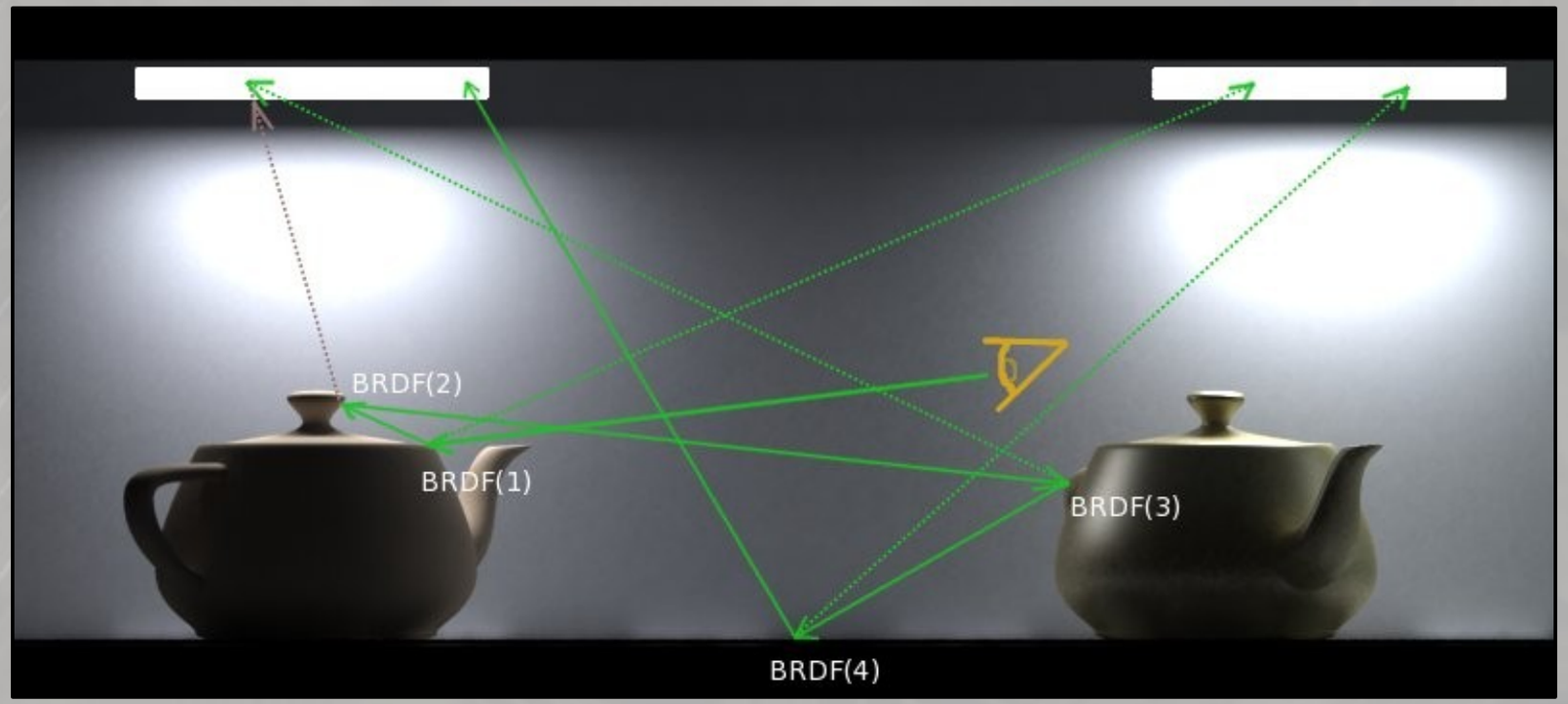
- Модификация 2:
  - При досега предложения алгоритъм за path tracing, шансът да попаднем на светлина не е голям
  - Затова, на всяка стъпка от строежа на пътя, ще направим „shortcut“, като директно свържем текущия край на пътя с лампата
  - Т.е., на всяка стъпка от строежа, пускаме два лъча – един случаен, който просто продължава пътя; и един към лампата, който „затваря“ текущия път

Path tracing (с модификации)



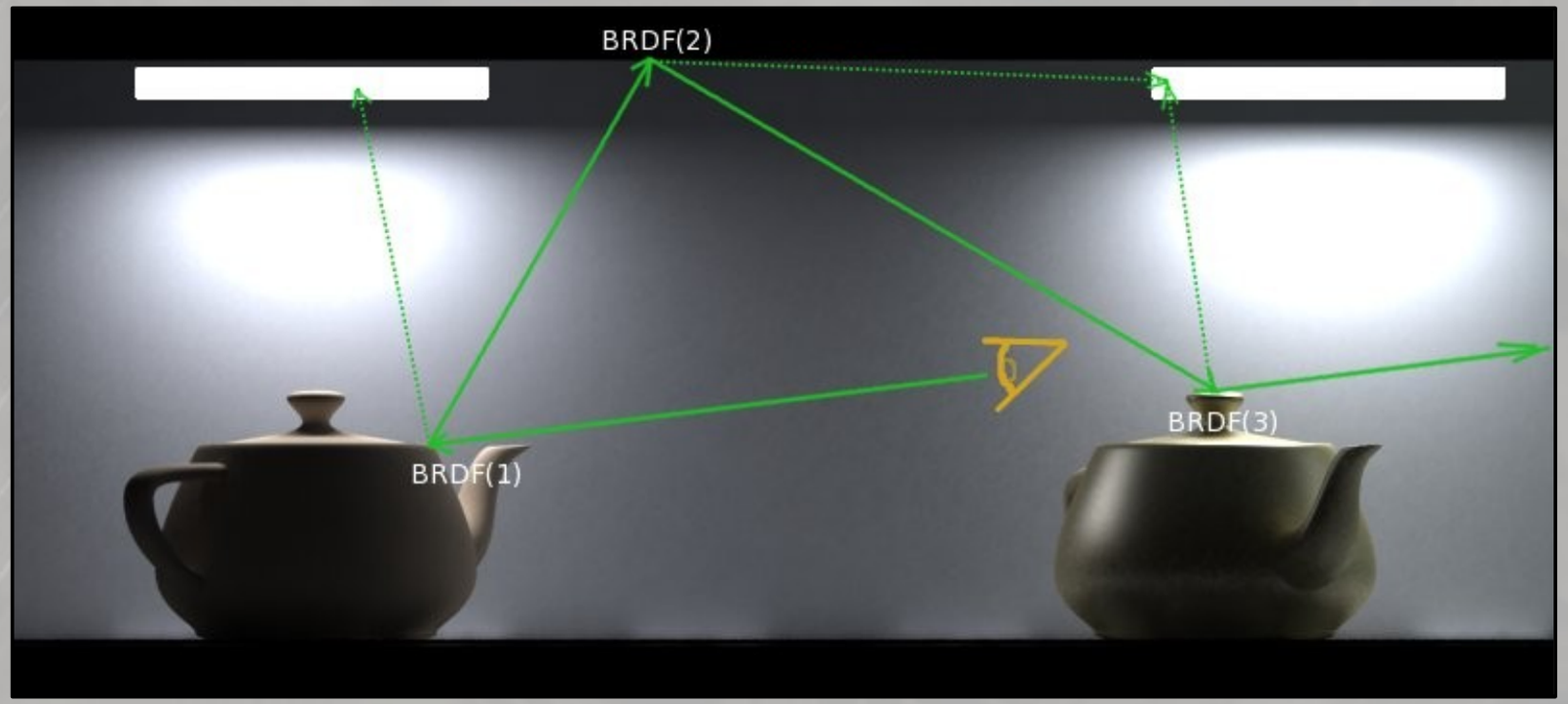
- С направените модификации, от точка 1 излизат два лъча

Path tracing (с модификации)



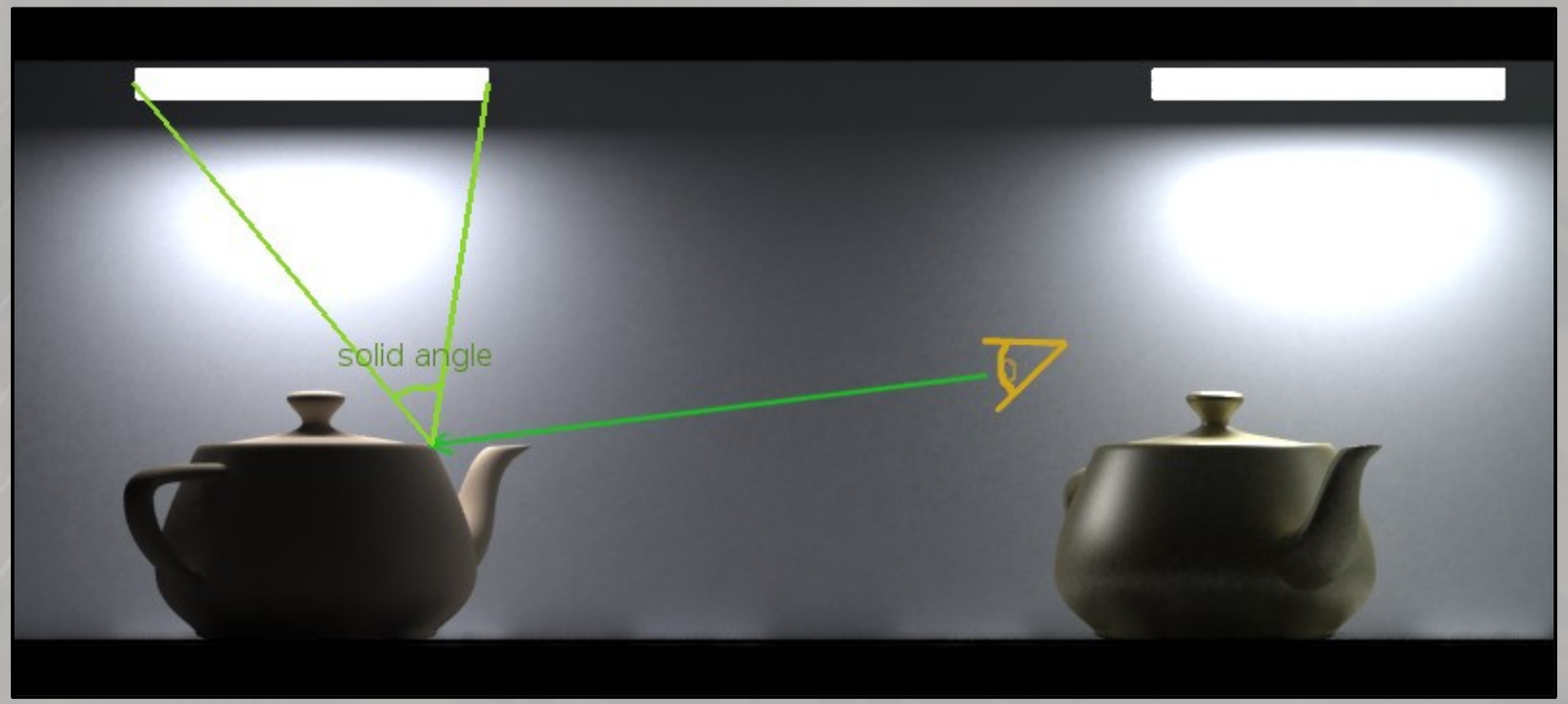
- Част от пусканите лъчи към лампите може да се окажат препречени (като примера в точка 2)

Path tracing (с модификации)



- Пътят може да завърши и когато лъчът отиде в environment-a

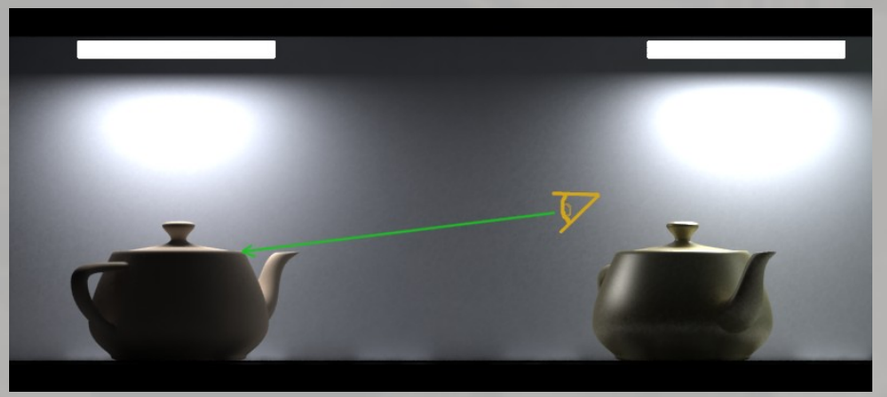
Path tracing (с модификации)



- При оценката на дадена лампа трябва да вземем в предвид солидния ъгъл, под който се вижда лампата от пресечната точка
  - По-голяма = по-ярка лампа

Anti-aliasing при Path tracing

- Тъй като първата част от пътя е винаги еднаква за всички пътища през един пиксел, може да вкараме antialiasing-а тук, като преместваме леко посоката на началния лъч вътре в пиксела (по подобие на това, което правехме при DOF)
- Понеже пускаме много лъчи, Antialiasing-ът ни идва безплатно



Обобщение

- Да обобщим:
  - За да сметнем цвета на даден пиксел, строим много пътища и усредняваме резултатите им
  - Началният лъч на всеки път тръгва от камерата, и минава през съответния пиксел (леко отместен вътре в пиксела, за AA)
  - При пресичане с геометрия, продължаваме пътя по два начина
    - Като пускаме случаен[1] лъч навън от пресечната точка
    - Като пускаме лъч към лампата[2]
- Въпросчета:
  - (1): Защо случаен? Това добър избор ли е (Reflection shader?)
  - (2): Коя лампа?

Въпрос 1

- Да се пуска случаен лъч е добра идея само за Diffuse BRDF-и, за които оценката на BRDF-а е еднаква навсякъде
- При отражателен BRDF, няма никакъв смисъл да пускаме случайни лъчи; BRDF-а е 0 навсякъде, с изключение на една единствена посока (шанса да я уцелим клони към 0)
- При грапави отражения пак не е добра идеята; шанса да да уцелим „активния“ район на отражението е малък; даже да похабим много семпли, резултатът ще е шумен

Importance sampling

- Решението: Importance sampling
  - Ще генерираме лъчи само в посоката, в която BRDF-ът е „силен“; по-прецизно казано, при даден входящ лъч ω и точка x, ще поискаме от BRDF-а да генерира изходящи лъчи ω' със същото разпределение, каквото е BRDF(x, ω, ω')
    - Например: чисто отражателния BRDF винаги ще генерира един и същи лъч (отразения на ω)
    - BRDF-а на грапавите отражения ще генерира лъчи по същия начин, както ги генерира Reflection shader-а сега (със случайно отклонение на нормалата и отразяване)
    - Diffuse BRDF-а (за Phong и Lambert шейдъри) ще генерира напълно случайни лъчи (но само в полукълбото, в което сочи нормалата)

Въпрос 2

- Коя лампа да изберем при пускането на „затварящи пътя“ лъчи? Какво да правим, ако лампата изисква семплиране (например, RectLight)?
  - Можем да направим пълно изчисление на директното осветление, както правихме и досега (сумираме всички семпли от всички лампи)
  - Но се оказва по-добре просто да си изберем една случайна лампа, да изберем случаен семпъл от нея, и да пуснем само един лъч към този семпъл
    - Т.е., да комбинираме няколко Монте-Карло стратегии в една

Въпрос 2

- Това комбиниране на Монте-Карло стратегии вече бяхме правили на няколко места:
  - DOF + Anti-aliasing и GI + Anti-aliasing
- Имаме вече доста проблеми, които решаваме с Монте-Карло
  - Меки сенки, вече и осветление (коя лампа)
  - DOF
  - Anti-aliasing
  - Грапави отражения
  - GI
  - Може да вкараме и други (например, при Layered shader избираме случаен слой, който да сметнем)

Защо е по-добре да ги комбинираме?

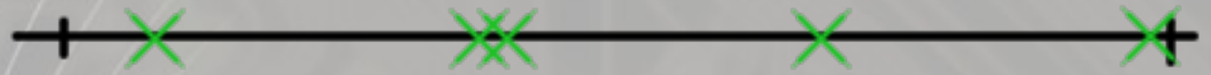
- Ако не ги комбинираме, имаме еквивалента на следното нещо:

$$\int_{\Omega} f(x)\, f(y) \left(\int_{\Psi} f(z)\, f(t)\, dt\, dz \right) dy\, dx$$

- В дадения пример, ако ползваме „некомбинирана“ Монте-Карло стратегия, ще изберем случайни стойности на *x* и *y*, след което ще пуснем вътрешно Монте-Карло за *z* и *t* (вероятно с много семпли), след което ще изберем други *x*, *y* и пак вътрешното Монте-Карло за *z*, *t* и т.н.
- Оказва се, че сходимостта е по-бърза, ако избираме наново случайни *x*, *y*, *z* и *t* за всеки семпъл

Интуитивен пример

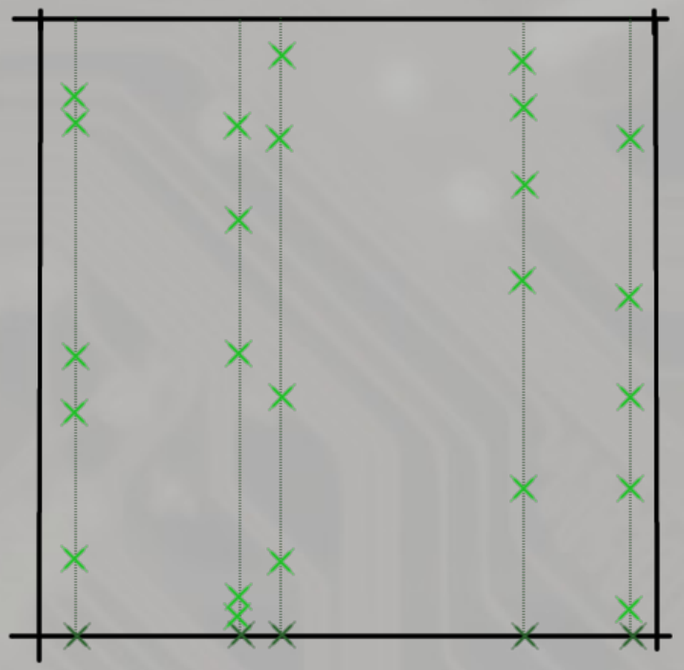
- Нека сме реализирали Монте-Карло за светлина тип „флуоресцентна лампа“ (линийка). Реализацията е просто да изберем 5 случайни точки от [0..1], по линийката:



- Да речем, искаме да разширим този метод за 2D вариант („квадратна“ лампа). Ако не комбинираме Монте-Карло схемата в двете измерения, то алгоритъмът ни ще е: избираме 5 случайни точки по X, и за всяка от тях избираме 5 случайни точки по Y.

Интуитивен пример

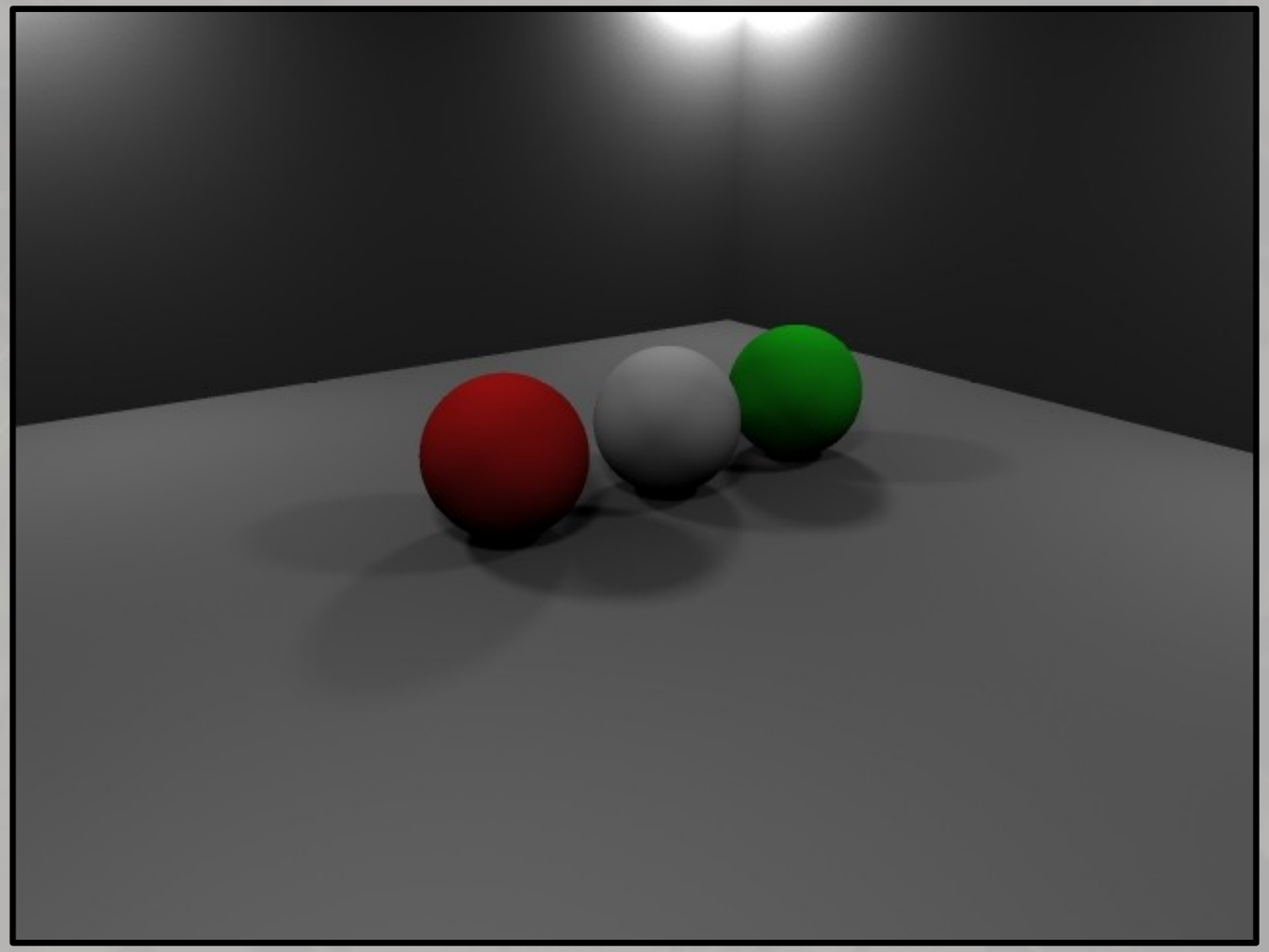
- С което поучаваме това:



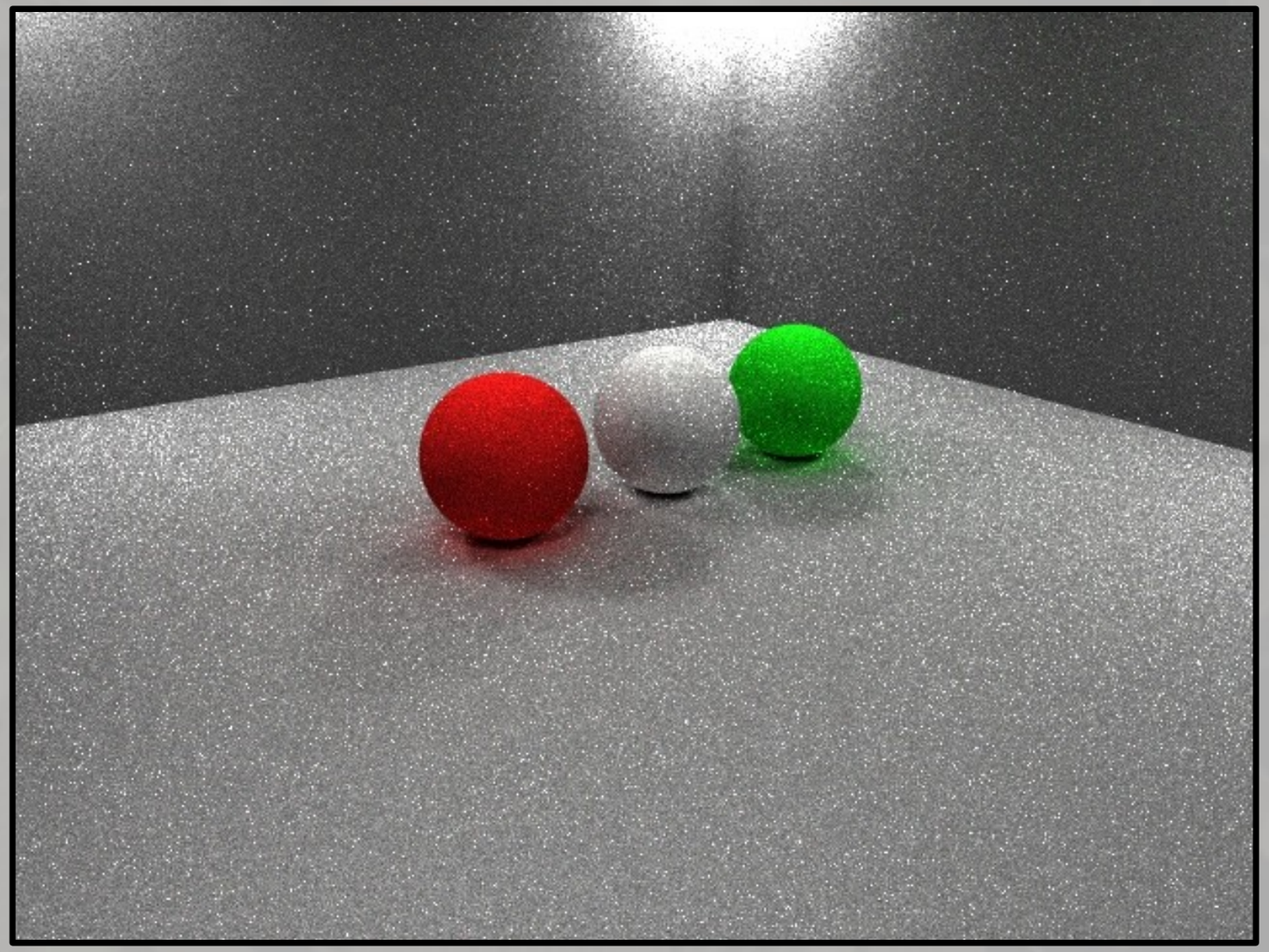
- Очевидно подобрение е да разпръснем 25-те случайни точки из квадрата, т.е. да смесим двете Монте-Карло схеми в една

Резултати

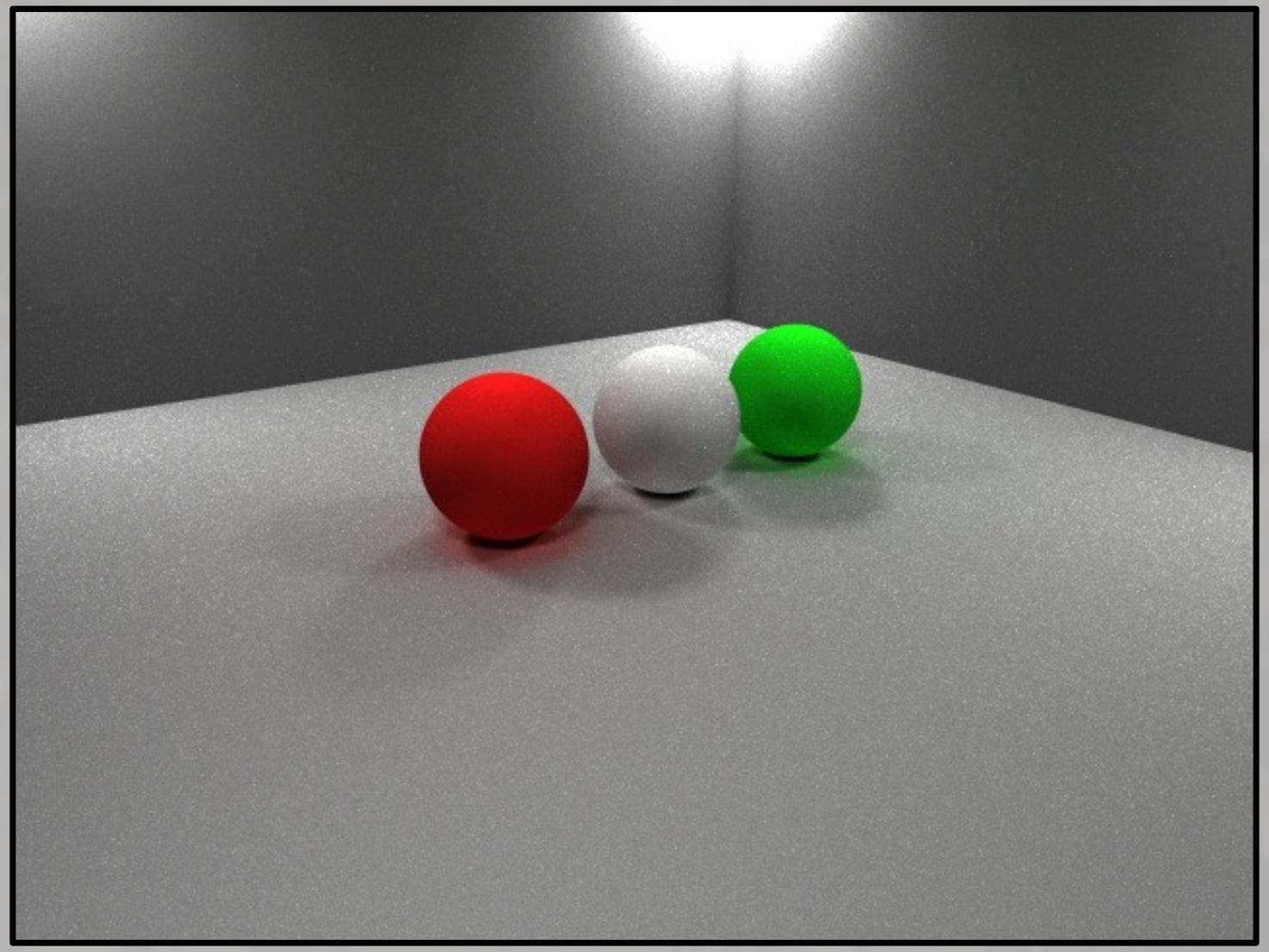
Директно осветление



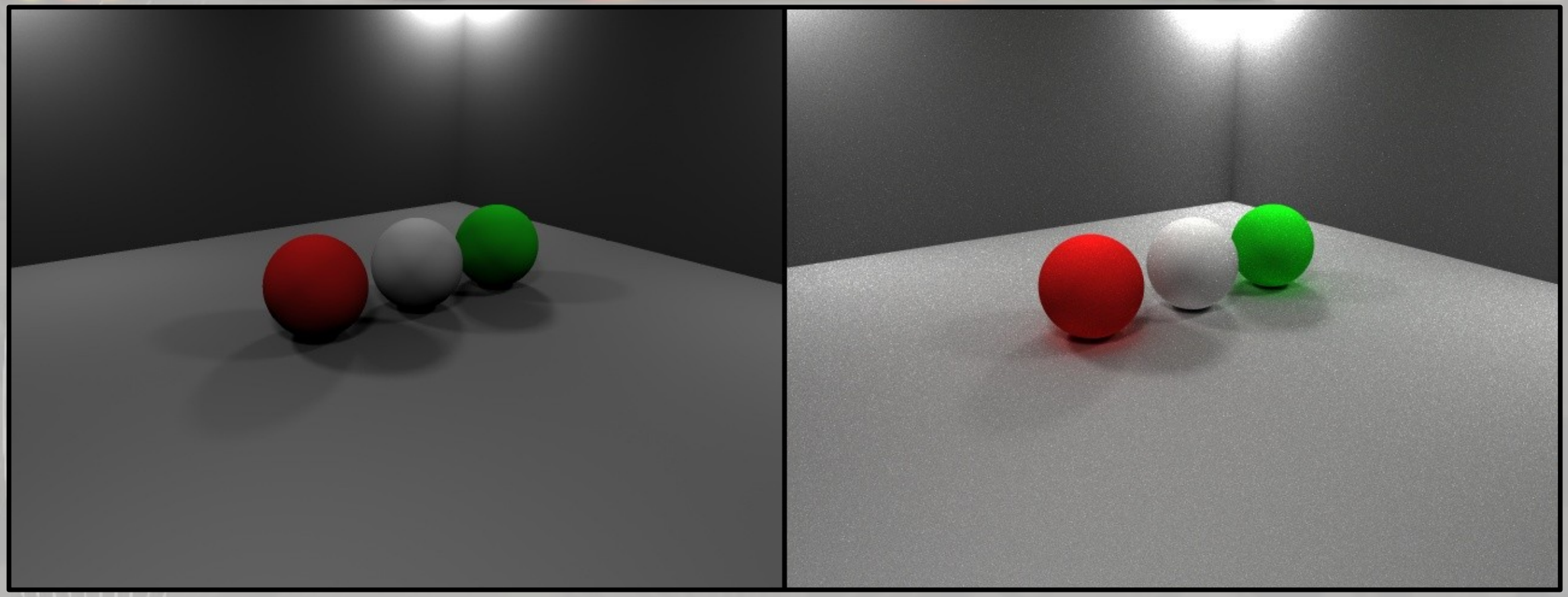
Path tracing (80 пътя на пиксел)



Path tracing (1600 пътя на пиксел)

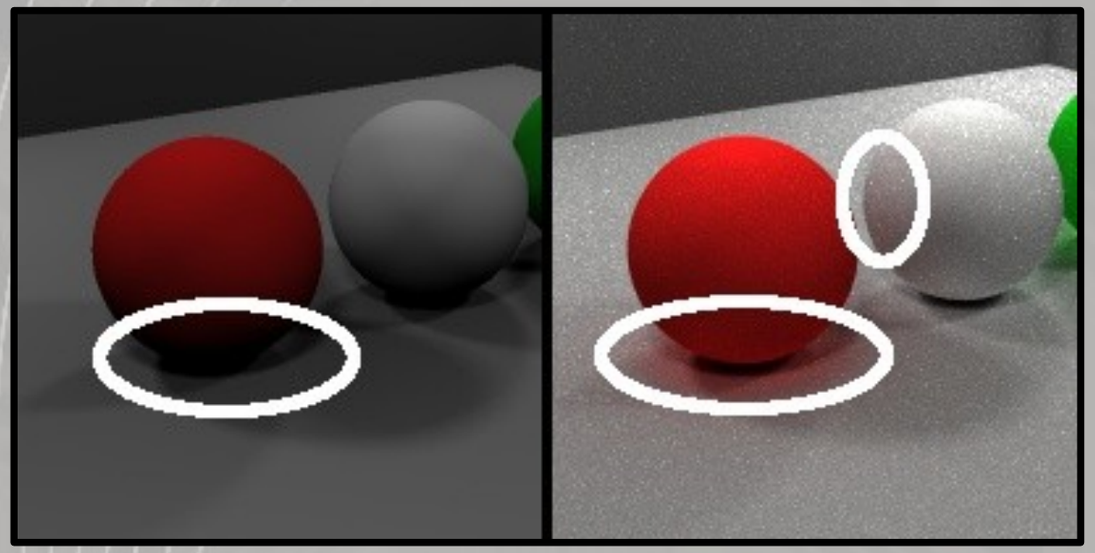


Сравнение



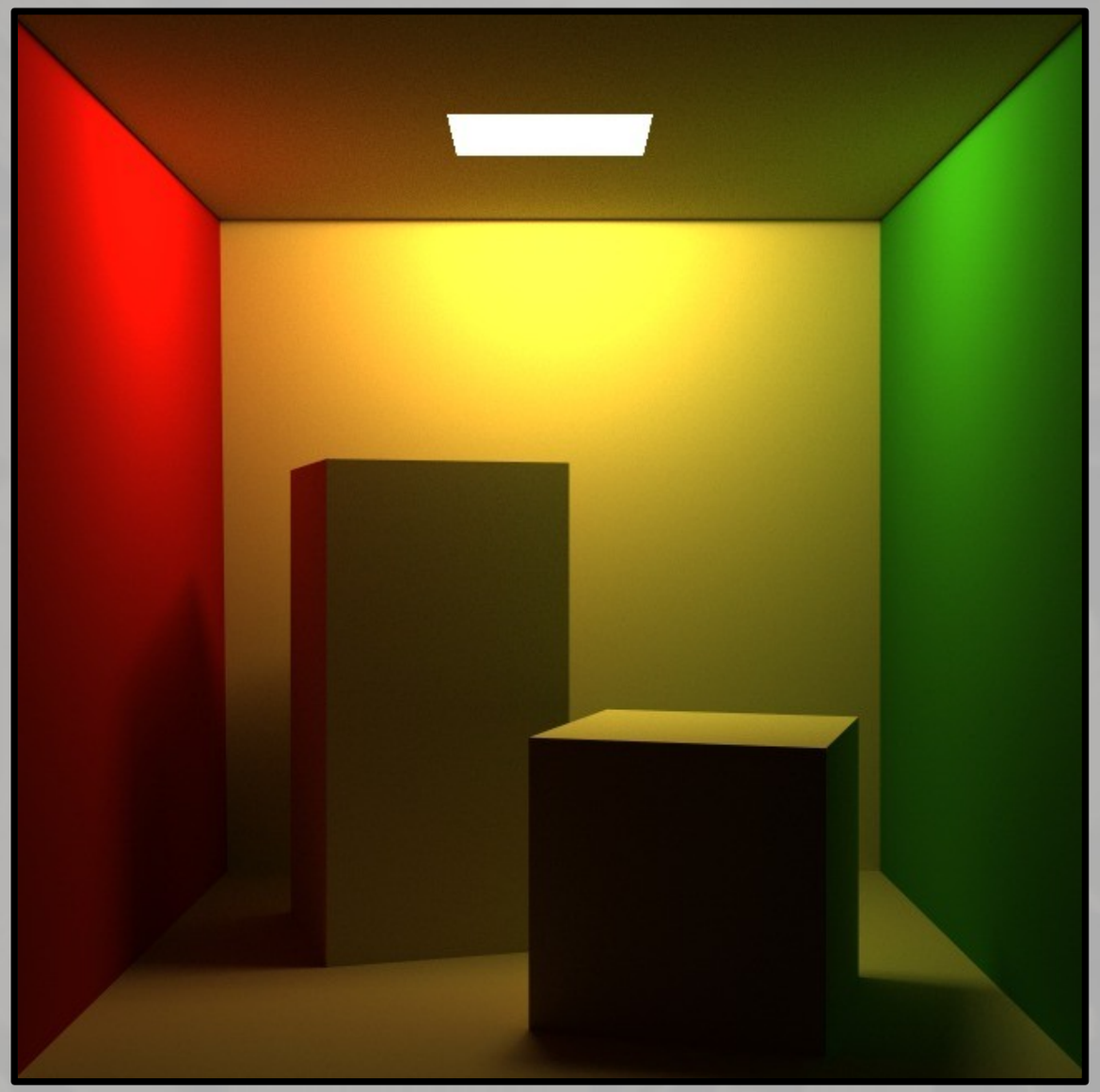
- GI варианта е значително по-светъл (заради индиректните отражения на лампите в сивите стени)
- Топките са осветени отдолу (индиректно от стените и пода)

Сравнение (отблизо)

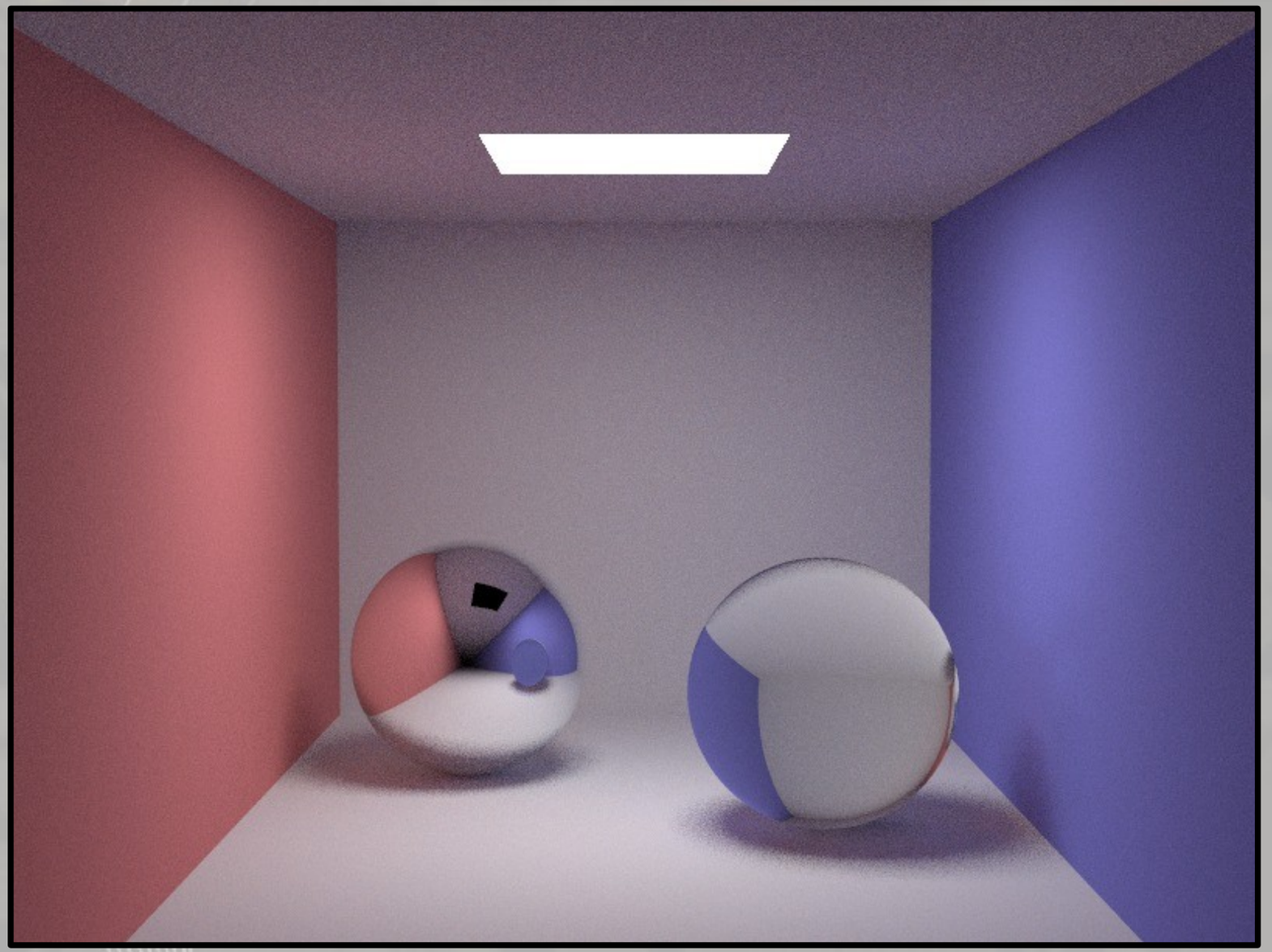


- В GI версията се забелязва ефектът на взаимо-осветяването на дифузните обекти – т. нар. „color bleeding“
- Сенките са доста по-слаби, защото има какво да ги освети. Плътни сенки има само директно под топките (ambient occlusion)

Cornell box (1000 пътя на пиксел)

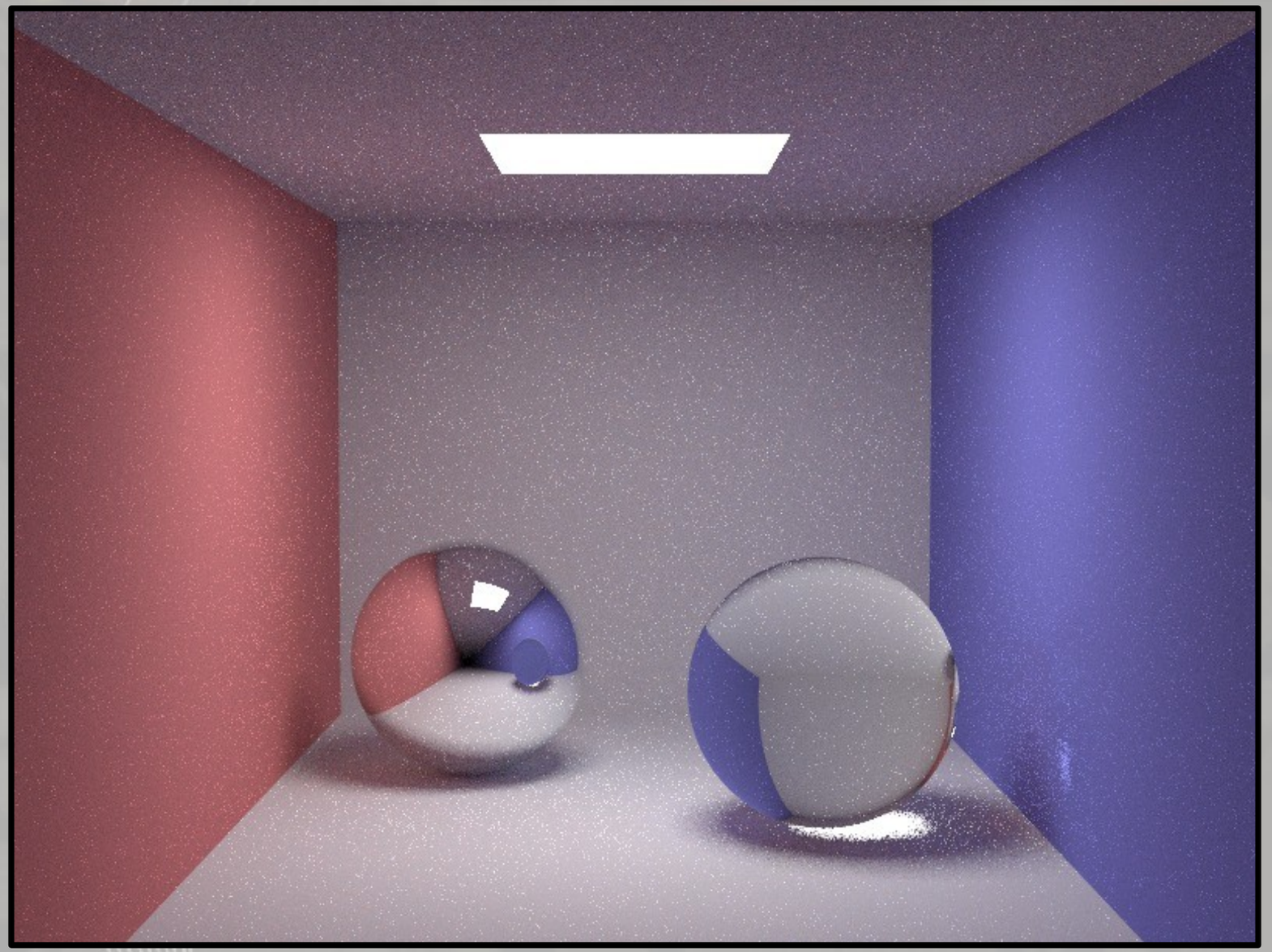


SmallPT test (40 пътя на пиксел)



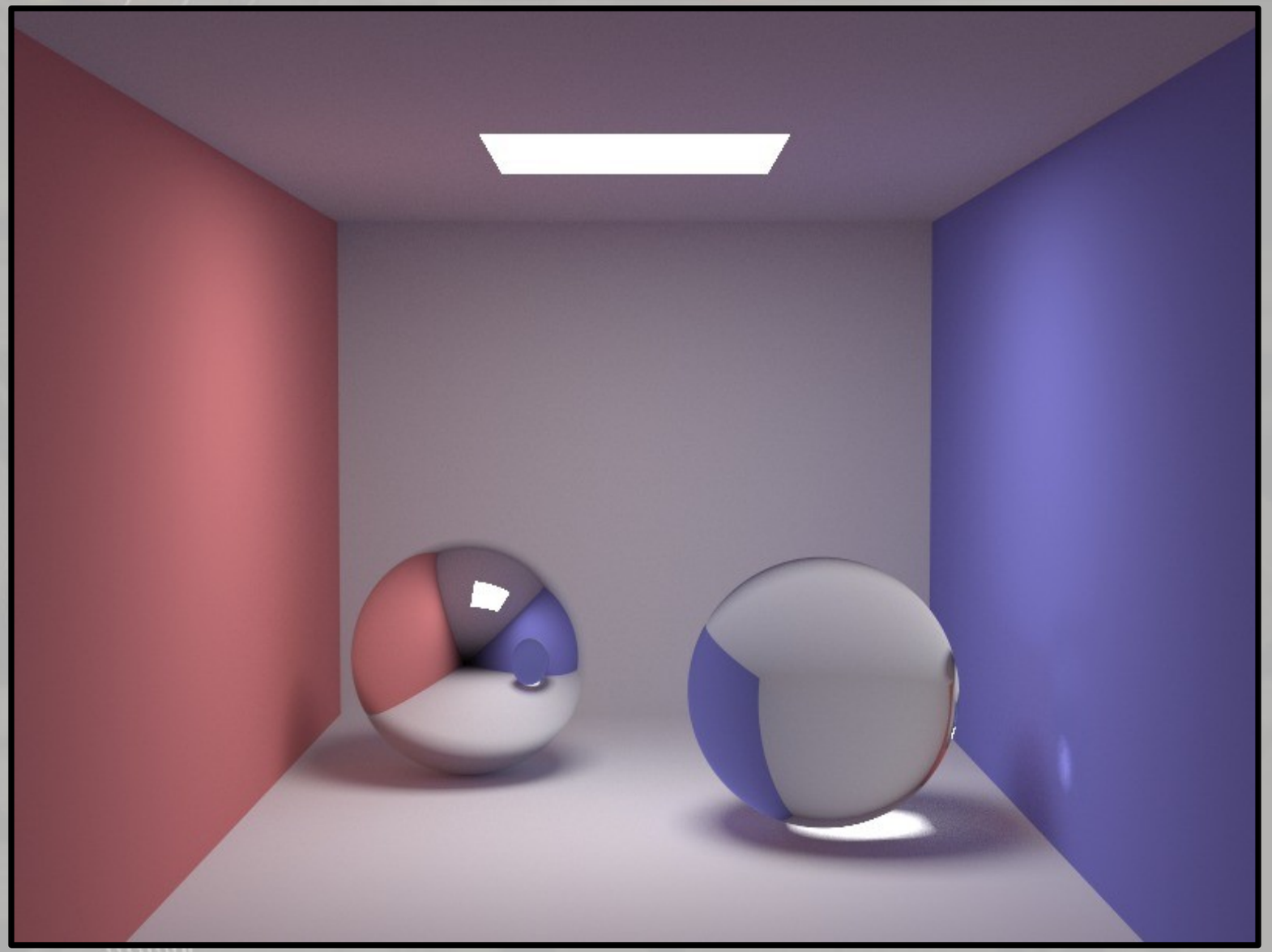
- Изрично семплиране на лампата и игнориране на директните попадения в лампата, за всички вторични лъчи

SmallPT test (40 пътя на пиксел)



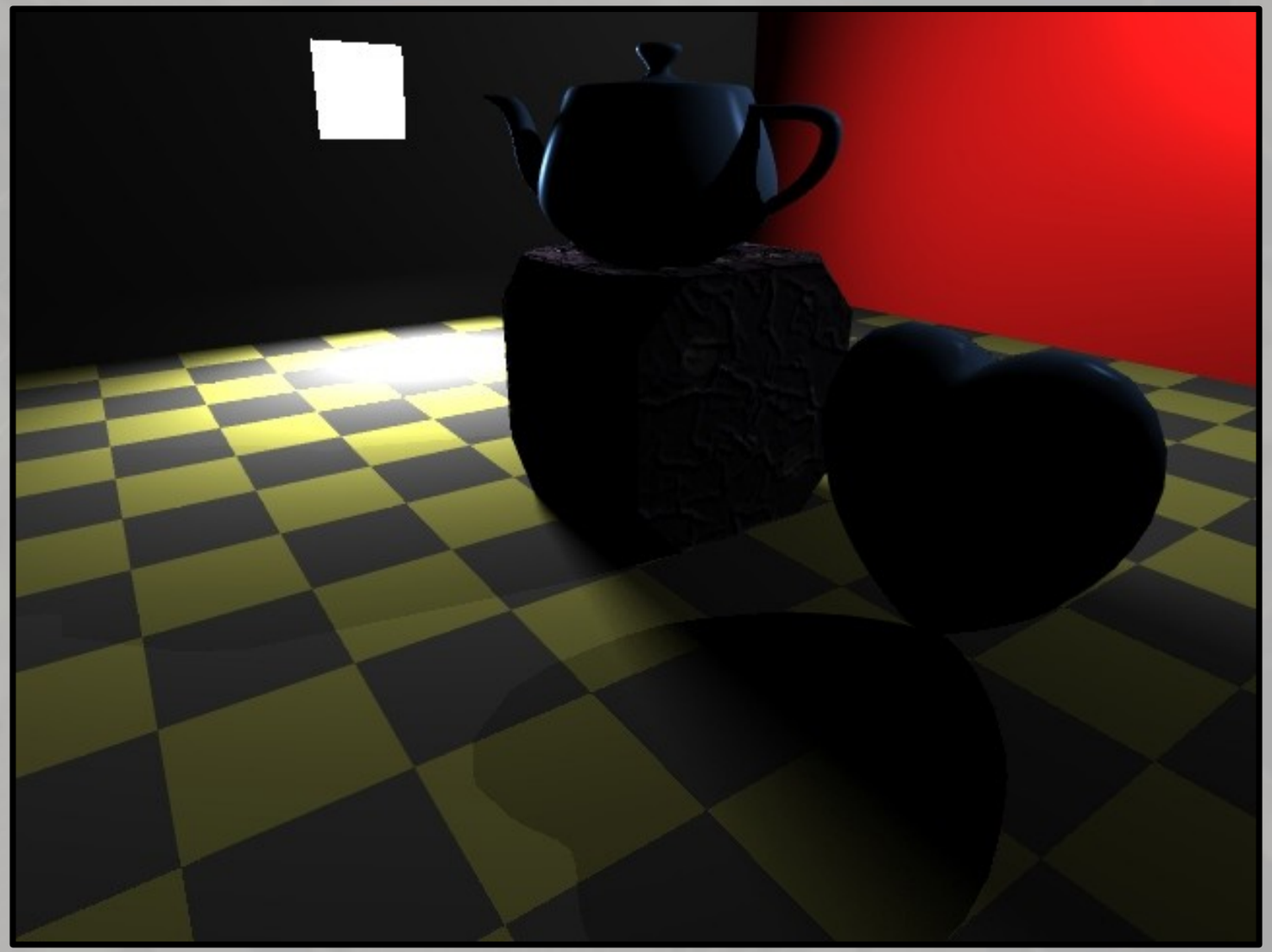
- С игнориране на директните попадения в лампата, но само след дифузно отражение

SmallPT test (4000 пътя на пиксел)

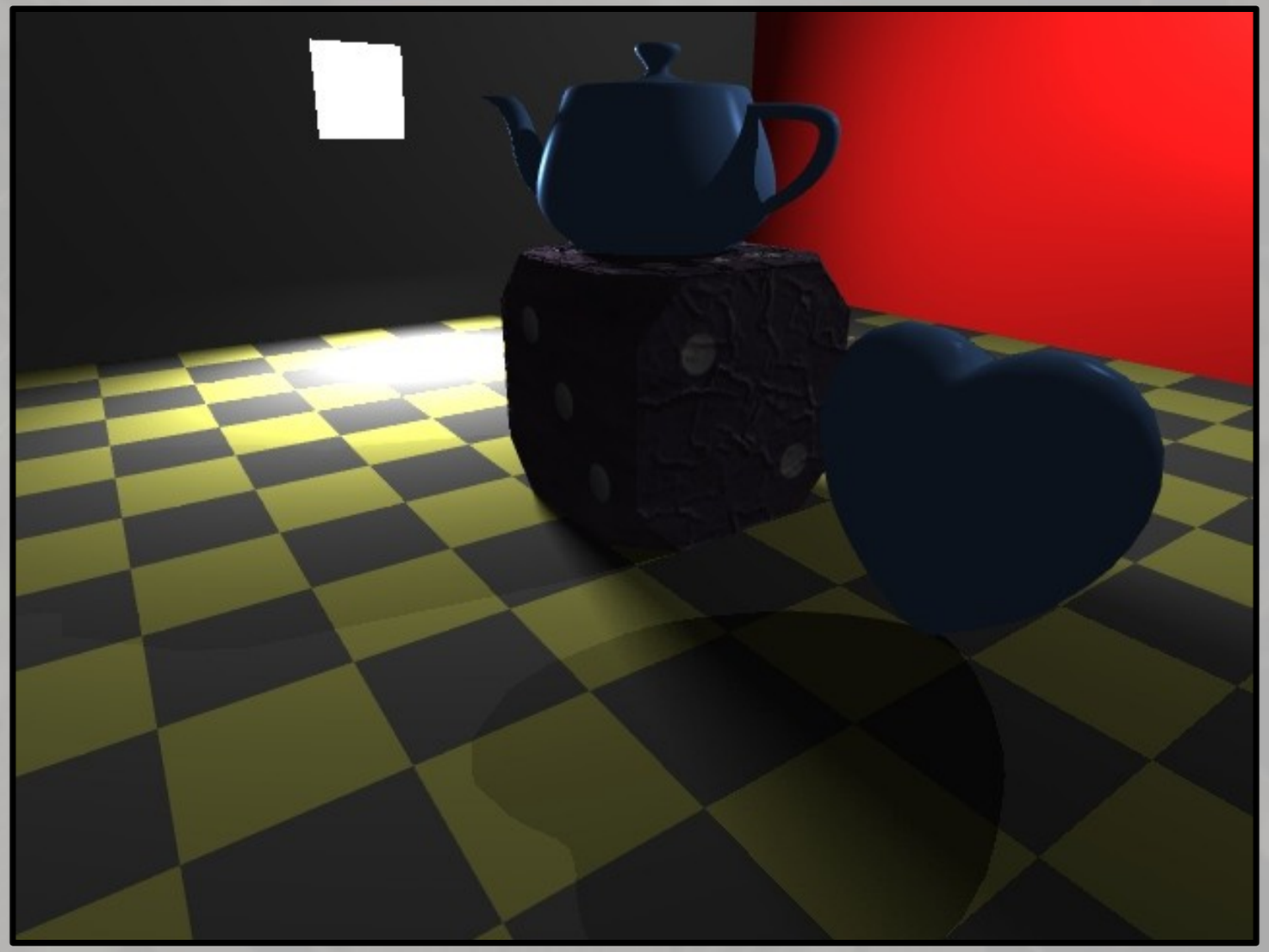


- За сравнение
  - 100 пъти повече пътища
    - (~10 пъти по-малко шум)

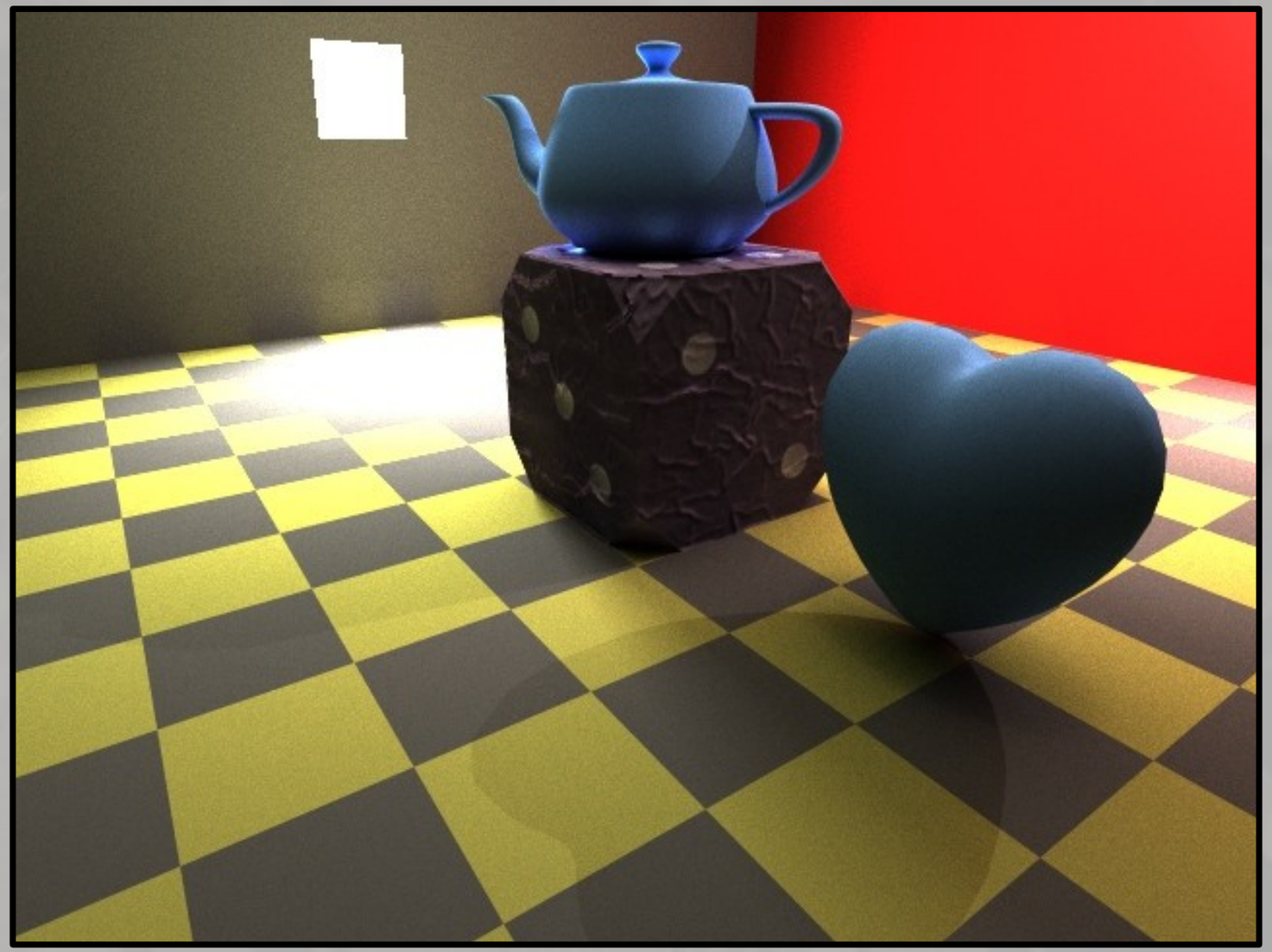
Директно осветление (без ambient light)



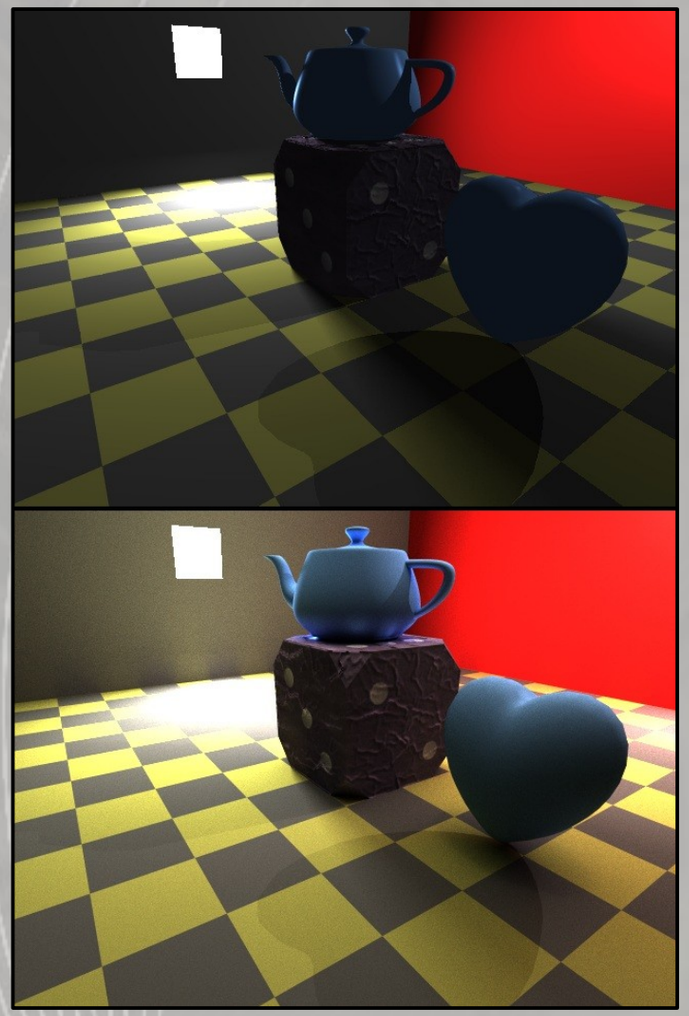
Директно осветление (ambient light = 0.15)



Path tracing (1600 пътя/пиксел)

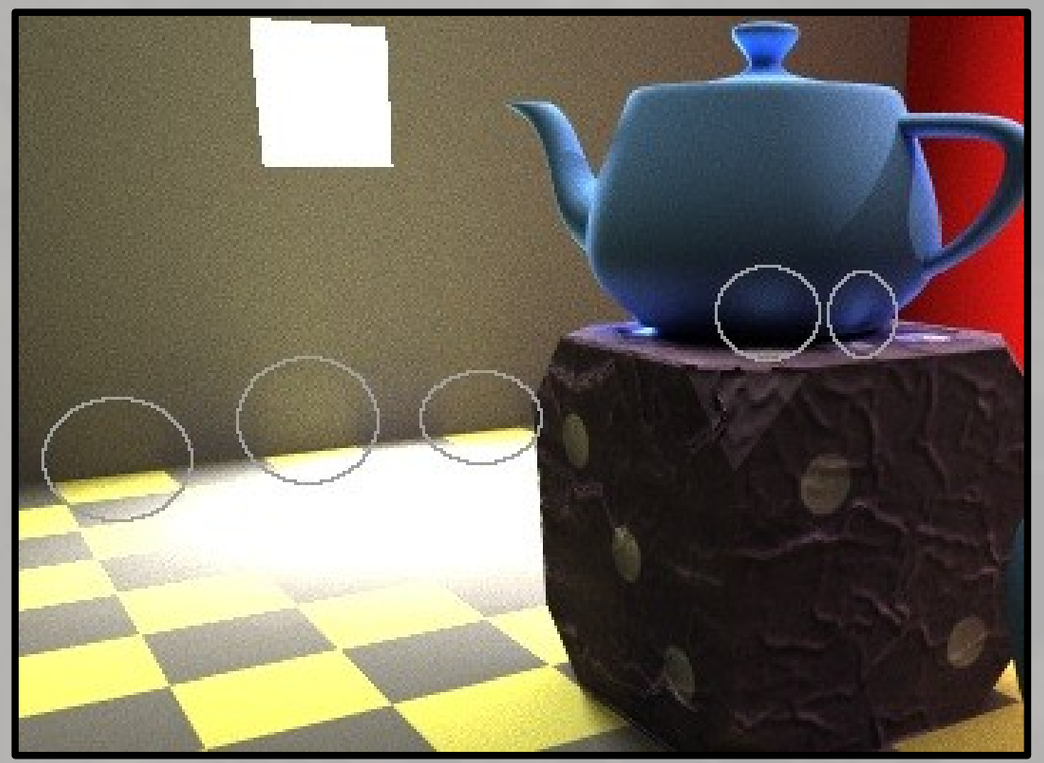


Сравнение



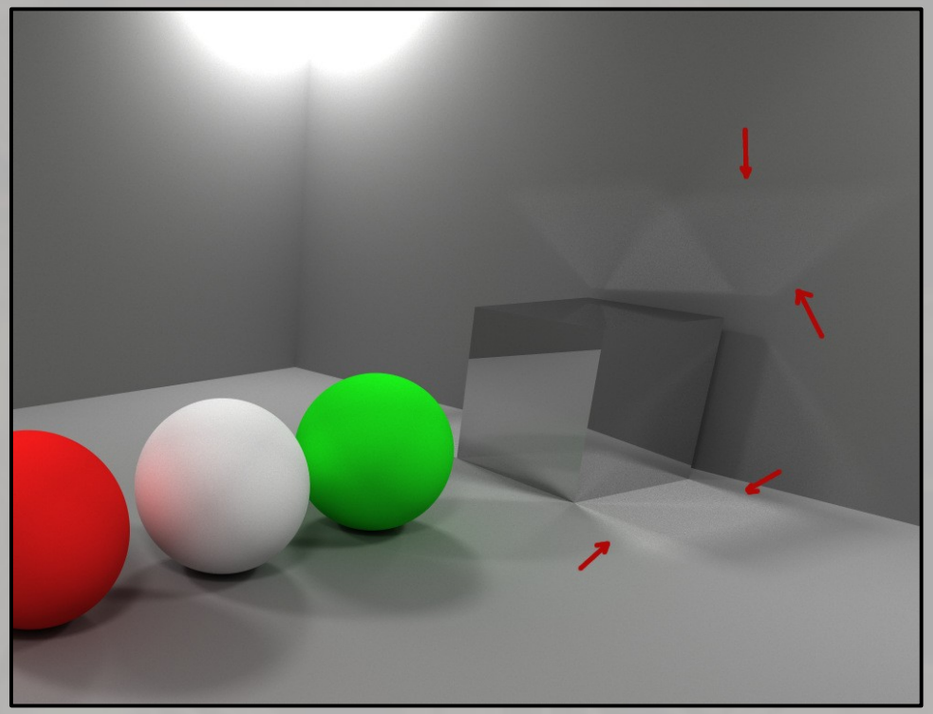
- Ambient light дава възможност да се види текстурата на лявата стена на зарчето, но заради липсата на светлосенки, релефът от bump mapping-а остава невидим
- Аналогично за сърцето: в горната картинка, то е осветено само от ambient light и изглежда плоско; GI вариантът възстановява усещането за обем на геометрията му

Сравнение



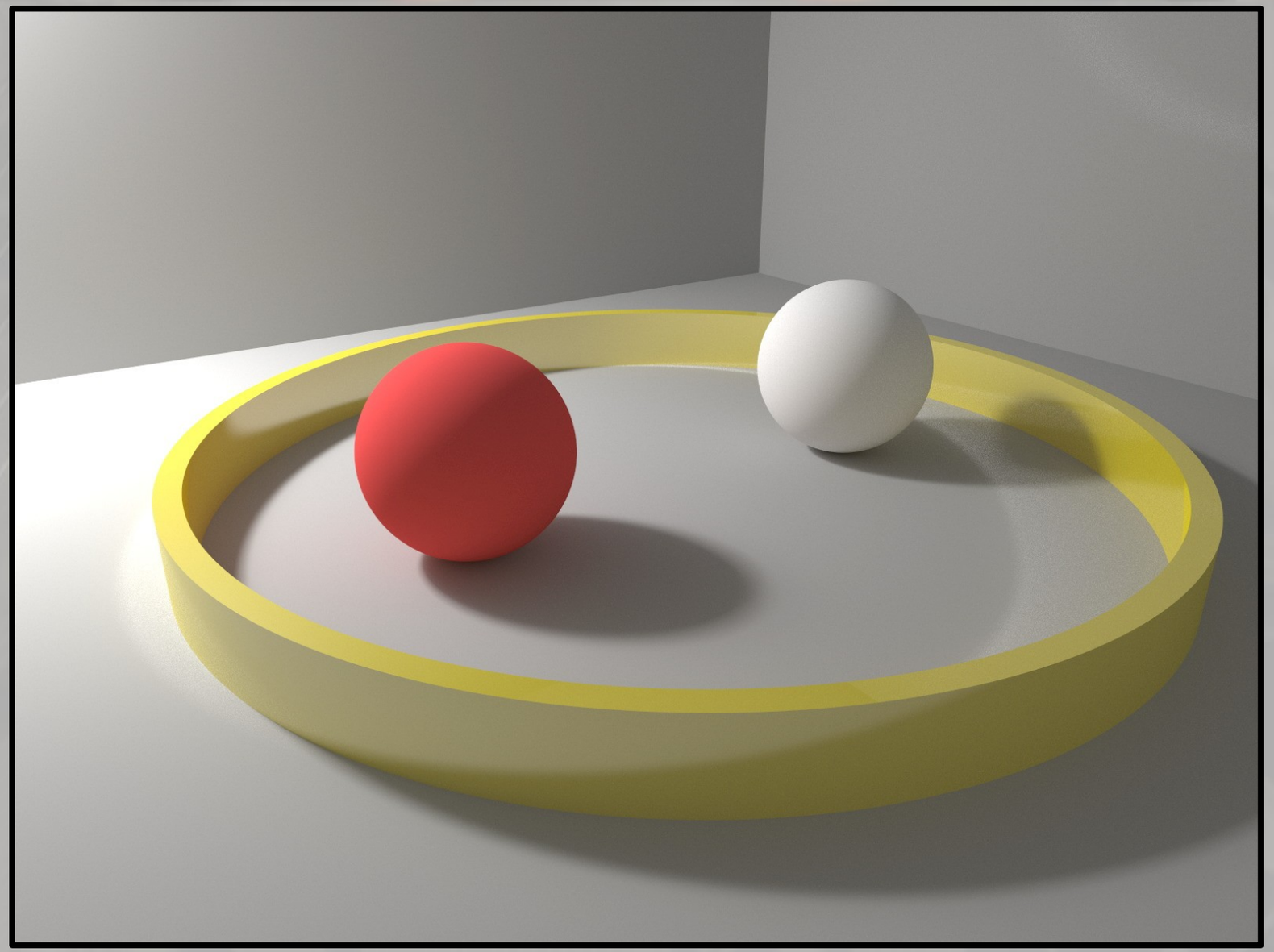
- Ясно различим color bleeding, заради разнообразие в текстурата

Слънчеви зайчета с PT



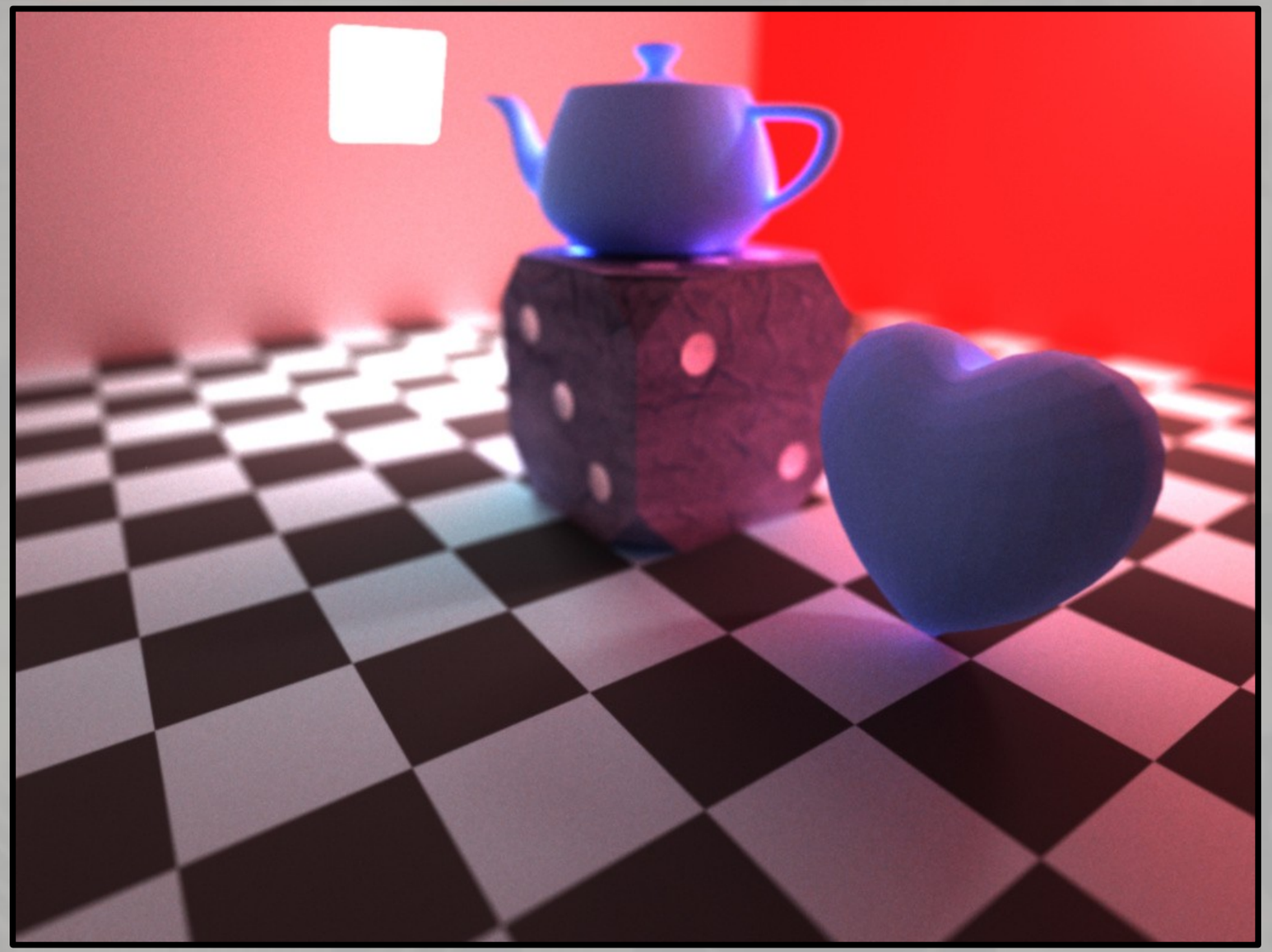
- Забележете отраженията от кубчето
  - Получават се от случайни пътища тип camera→diffuse→ →mirror→light!
  - Ефектът е много шумен: 6k пътя на пиксел за тази картинка

Слънчеви зайчета с PT (2)



GI + DOF

- Глобалното осветление се комбинира добре с дълбочината на полето
- Поредният пример за смесване на Монте-Карло стратегии



Потребителят има думата

- Последните примери с boxed сцената са с изкуствено засилен GI
  - Резултатът не е физически верен; алгоритъмът става отместен
  - Ако потребителят всъщност иска точно това, ние нямаме право да го ограничаваме

Недостатъци

- Path tracing с 1600 пътя на пиксел и разделителна способност 1920×1080 отнема няколко часа на съвременна машина (а дори при 1600 има видим шум)
  - Има разнообразни техники за ускоряване и подобряване на работата на основния Path tracing алгоритъм: адаптивен path tracing, „руска рулетка“, ...
  - От неотместените алгоритми, Bi-directional path tracing е значително по-сложен за реализация, но често дава сериозно подобрение спрямо path tracing (от гледна точка на шум)
  - В професионалните 3D пакети често се ползват отместени алгоритми (photon map, irradiance map)