

3D графика и трасиране на лъчи v.4.0



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

Тема 14

**Рейтрейсинг в реално време
Оптимизации
Vista буфери, адаптивен рейтрейсинг**

Съдържание

- Анонси
- Рейтрейсинг в реално време

Анонси

- Дата за тестове: 14.02, от 11:00, 210/ФМИ
 - Само за хора, които не са правили някои от тестовете
 - Тест 2 ще включва въпроси и от тази лекция (#14)
- Изберете си дата за защита на проекти (6 или 11 Февруари)
 - 11^{ти} засега се очертава по-натоварена

Рейтрейсинг в реално време

- Както споменахме в миналата лекция, един от начините за ускоряване на рейтрейсинга до Realtime скорости е като ползваме хитри алгоритми, които работят добре в повечето случаи (или при налагане на някои ограничения какво може да се рендерира)
 - В тази лекция ще обсъждаме основно този тип алгоритми
 - Повечето от тях също са тривиални за паралелизация, така че не изпускаме възможността да се възползваме от грубата сила, когато можем

LDR рендериране

- LDR = low dynamic range, т.е., обратното на HDR
- Например, да ползваме 32-битов целочислен цвят, вместо сегашния клас Color
 - **typedef unsigned** Color;
 - Само че трябва да реализираме и всички operator-и!
- Така представянето на цвета е много компактно и се вписва добре в архитектурата на 32-битовият x86
 - Например, една функция, която връща Color (а това са голяма част от функциите в рейтрейсъра!) ще я връща в регистър, вместо в структура в паметта

LDR рендериране

- Предимства
 - Скорост (по-малко операции, по-малко копиране)
 - По-малко памет (особено за текстури!)
 - По-малко трансфер между процесорите и паметта
 - Особено важно в много-процесорните системи
 - В някои случаи може да измислим как да реализираме някои от примитивните операции със SIMD-подобен код
 - Имаме „безплатен“ четвърти канал, който можем да ползваме за разнообразни трикове (алфа, маскиране, ...)

LDR SIMD пример

- Следният пример смесва цветовете от два буфера (например, текстури) в съотношение 50:50

```
// Mix two buffers using a 1:1 ratio. Result is stored in bufferA.
```

```
void mixArraysUint32(unsigned bufferA[], unsigned bufferB[], int numPixels)  
{  
    for (int i = 0; i < numPixels; i++)  
        bufferA[i] = ((bufferA[i] >> 1) & 0x7f7f7f) + ((bufferB[i] >> 1) & 0x7f7f7f);  
}
```

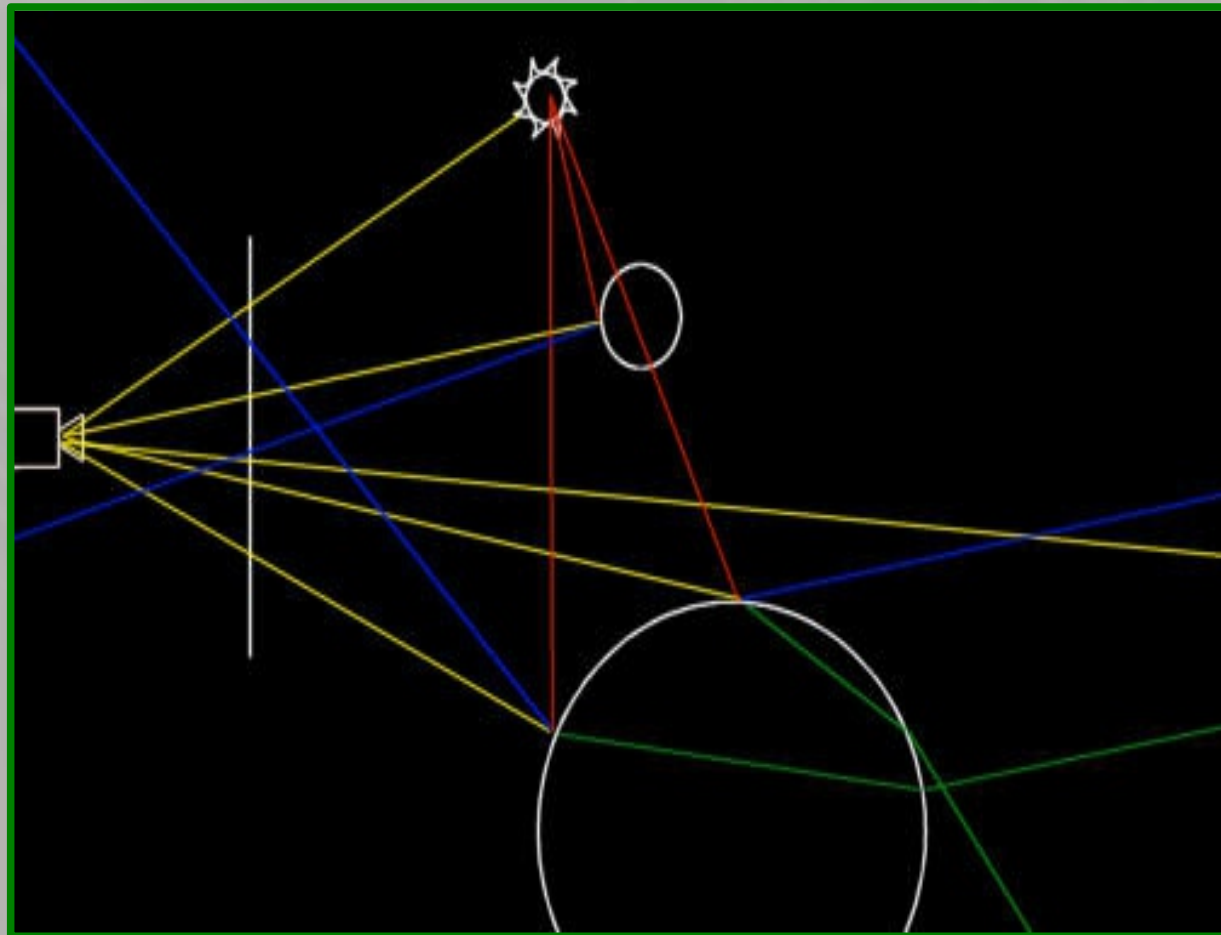
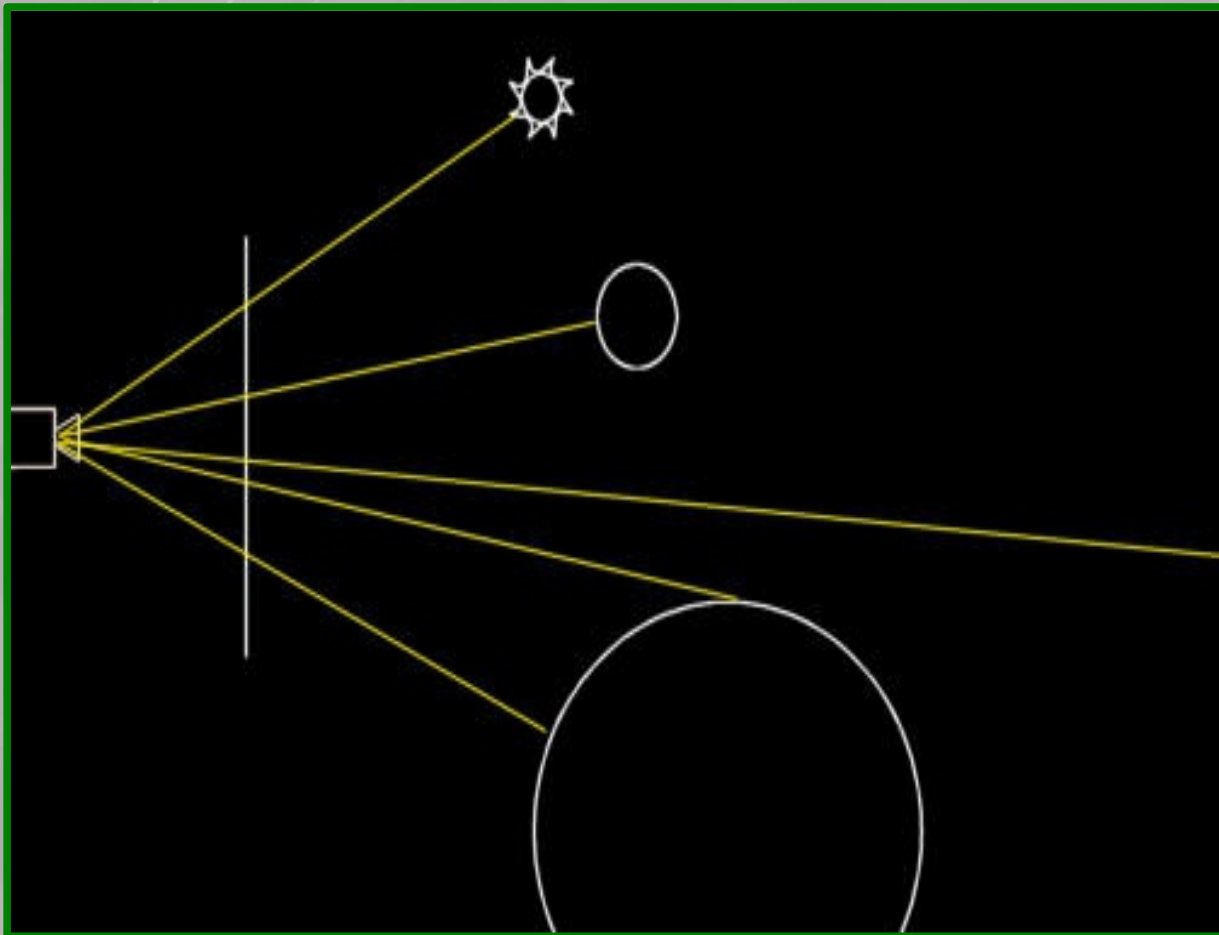

LDR рендериране

- Недостатъци
 - Очевидно, не е HDR
 - Усложняват се доста операциите с ярки цветове, например, както са лампите при нас
 - Всяко натрупване (например, за Glossy отражения, за DOF и прочее) трябва да се извършва временно в по-широк обхват и изисква специален код за всеки случай
 - Недостатъчна точност – ако цвета на един пиксел зависи от много различни приноси (различни светлинни източници, различни слоеве на шейдъра и пр.), 8 бита точност не стигат (получава се квантизация)

Първични и вторични лъчи

- Първични лъчи: лъчите, които излизат от камерата
- Вторични лъчи: всички останали лъчи (след първото пресичане от камерата с геометрията):
 - Лъчи за проверка за сенки (shadow rays)
 - Отразени и пречупени лъчи
 - Glossy лъчи
 - GI лъчи
 - ...
- Отношението първични:вторични при GI е нищожно

Първични и вторични лъчи



- Първичните лъчи са в жълто, вторичните са в другите цветове

Първични и вторични лъчи

- Причината да се третираат отделно е, че има съществена разлика между тях
 - Първичните лъчи са по-„важни“, не можем да минем без тях
 - Първичните лъчи са *кохерентни*, т.е., те се движат в приблизително подобна посока, тръгват от една точка и са „равномерно“ разпределени
 - За разлика от вторичните, които тръгват от произволни точки в произволни посоки – т.е., с доста по-хаотично поведение
 - Една немалка част от оптимизациите целят ускоряване на трасирането на първичните лъчи; за вторичните не може да направим нищо, освен да се стремим да ги намалим

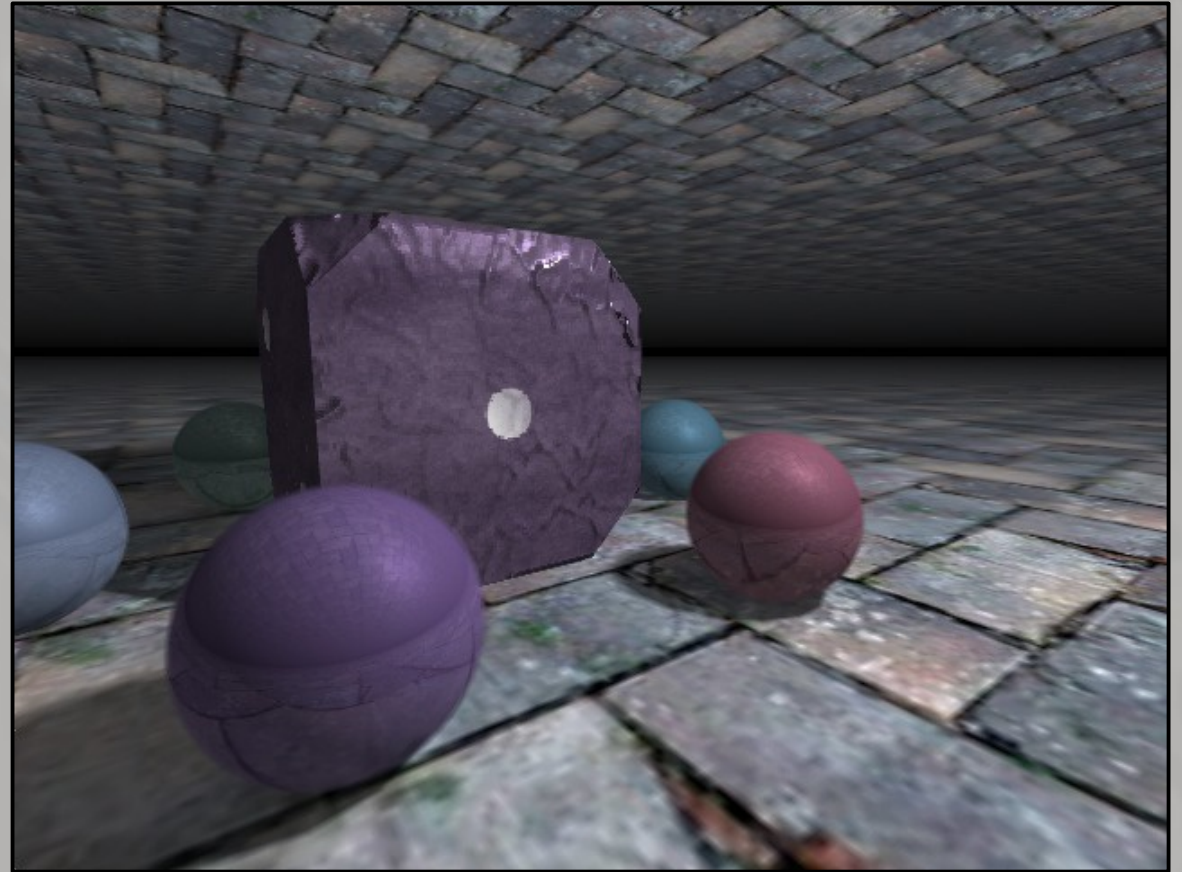
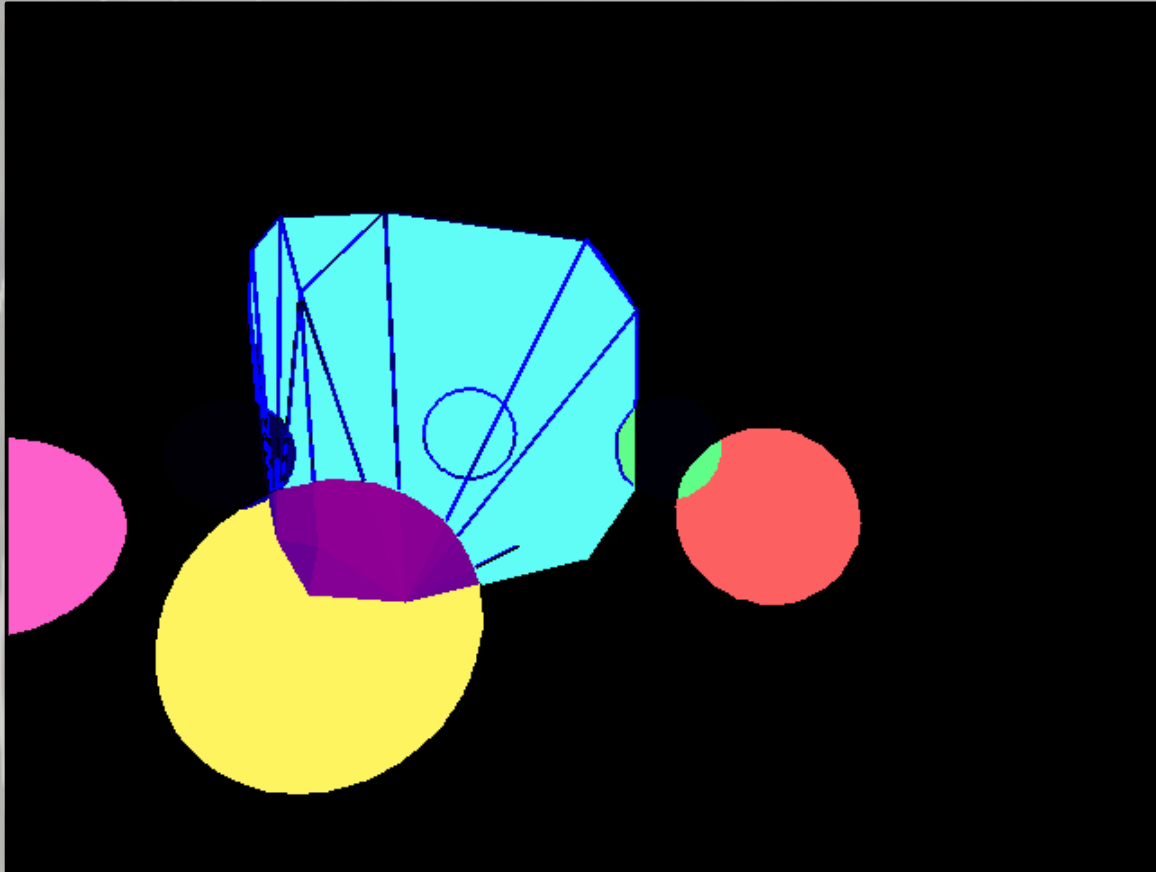
Vista буфери

- Идеята е да ползваме Z-buffer алгоритъм за да ускорим трасирането на първичните лъчи
- За целта, рендерираме триъгълниците от геометрията, ползвайки растеризация, в специален буфер (vista буфер). В буфера записваме id-тата на обектите-собственици на триъгълниците. Така накрая имаме образ, наподобяващ крайната картинка, само че, вместо цветове, в пикселите ще имаме id-та на обектите, които лъчите ще уцелят като ги трасираме
- Така си спестяваме трасирането на първичните лъчи

Vista буфери

- Т.е., след построяването на виста буфера, за всеки пиксел ще знаем кой ще е обекта, който лъча ще уцели, ако пуснем истинско трасиране
 - Затова може директно да пресечем лъча със самия обект и след което си викаме Shader-а на обекта, както и досега.
 - Ако все пак лъчът не пресече предсказания обект?
 - Можем да пуснем пълен рейтрейсинг; очакваме тези пиксели да са много малка част от картината

Vista буфер – пример



- Отляво е Vista буферът, отдясно е крайната картинка
- Демонстрация [fract]

Рейтрейсинг с Vista буфери

```
def computePixelWithVistaBuffer(x, y):  
    ray = getScreenRay(x, y)  
    objectId = vistaBuffer[x, y]  
    if objectId == None:  
        return getEnvironment(ray) // уцелваме задния фон  
    obj = getObjectById(objectId)  
    if obj.intersect(ray, info):  
        return obj.shader.computeColor(ray, info)  
    else:  
        return oldRaytrace(ray)
```

– (разчитаме oldRaytrace() да се вика доста рядко)

Рейтрейсинг с Vista буфери

- Как се построява самият Vista буфер?
 - Всяка геометрия трябва да може да генерира списък от триъгълници, които да я представляват
 - За Mesh е тривиално; но, например, сферата ще трябва да се превърне в кълбовиден многостен
 - Допуска се Level-of-Detail оптимизация; ако дадена сфера е надалеч, то нейният многостен може да е по-груб, с по-малко триъгълници
 - Всеки триъгълник носи някакво id, което еднозначно определя на коя геометрия принадлежи

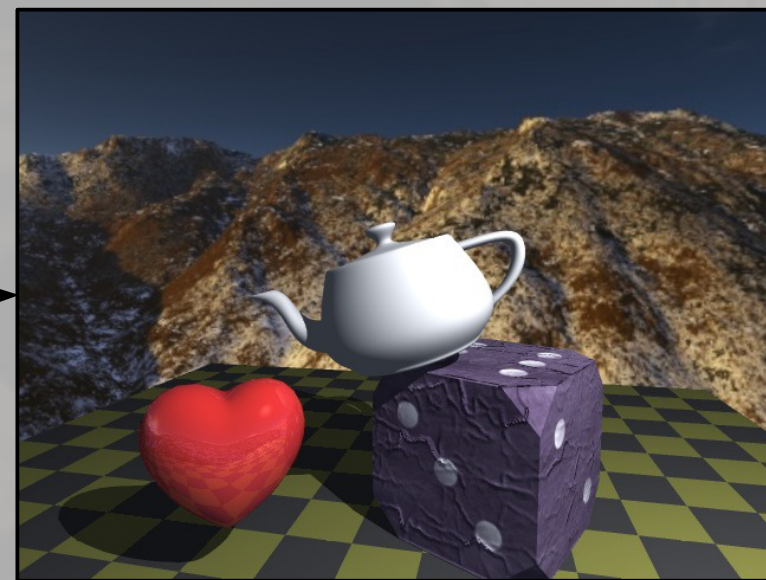
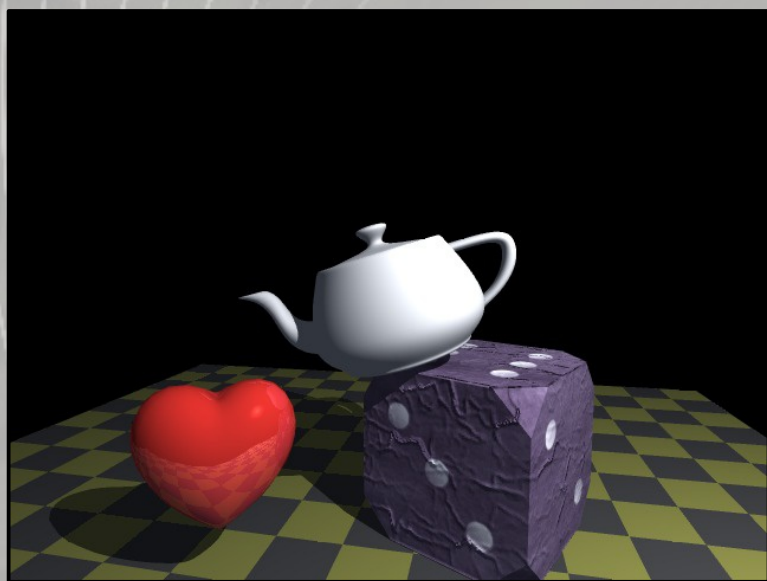
Рейтрейсинг с Vista буфери

- След което триъгълниците се изчертават във Vista буфера чрез растеризация
 - Use the GeForce, Luke...
 - ... но може и софтуерно
- Недостатъци на Vista буферите
 - Ползват както raytracing, така и Z-buffer, носят недостатъците и на двата свята
 - Не работят с не-правоъгълни камери (напр., fisheye)
 - Ускоряват само първичните лъчи
 - В затворена сцена, за всеки първичен лъч, ще има поне по един вторичен (shadow ray към лампата), така че ползата е max. 50%

Построяване на Vista буфера (софтуерно)

- 1) Обхождат се всички Node-ове и се взимат списъци с представящи триъгълници
- 2) Триъгълниците се изрязват спрямо зрителното поле (изхвърляме триъгълниците извън полезрението; от частично участващите изрязваме невидимата част)
- 3) Сортираме триъгълниците по разстояние до камерата
- 4) За всеки триъгълник, трансформираме 3-те му върха в 2D (чрез обратна на `getScreenRay()` функция) и го растеризираме ред по ред във Vista буфера, чертаейки с `id`-то, което е дадено на този триъгълник.

Многослойно рендериране



Многослойно рендериране

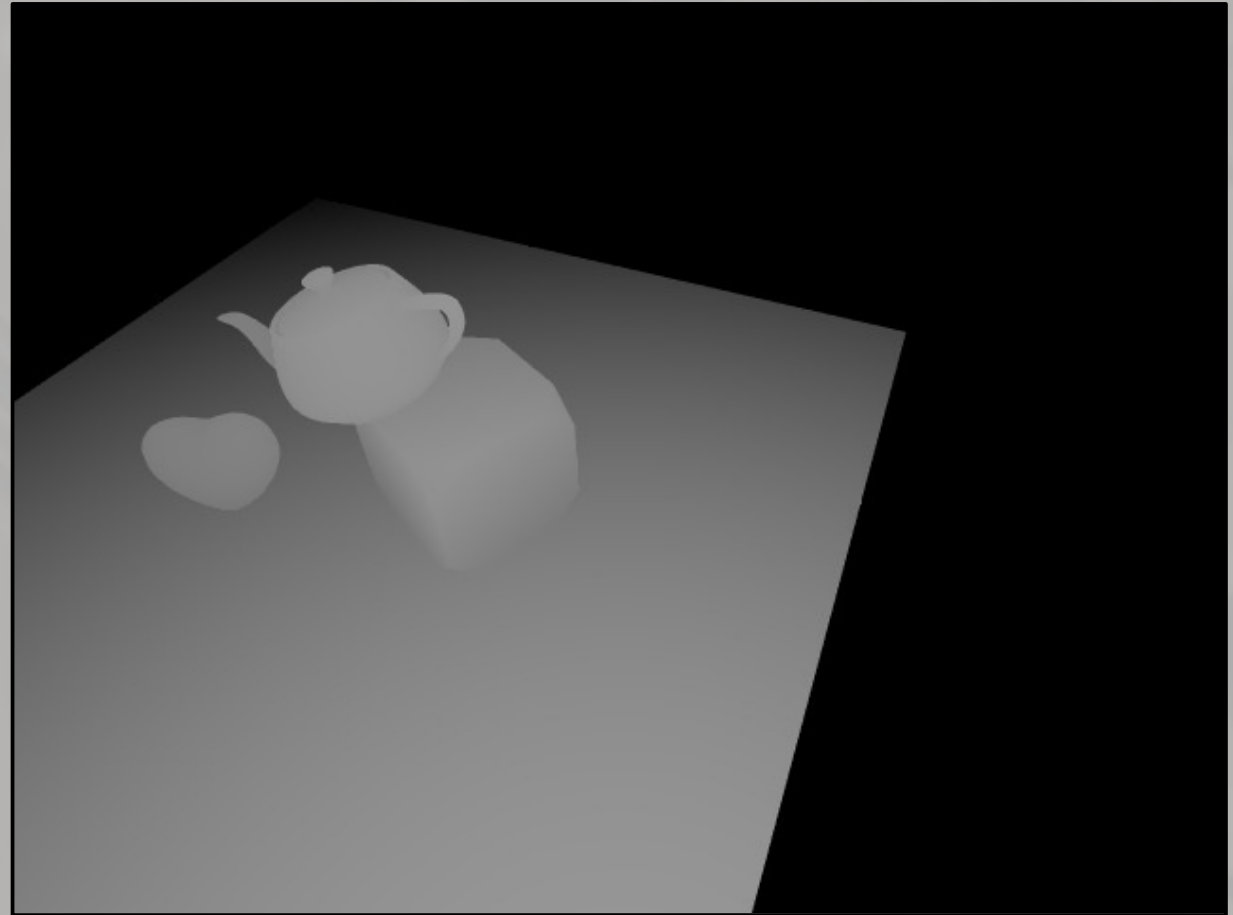
- В някои случаи се оказва удачно, различните елементи от сцената да се рендерират отделно, и след което да се композират по подходящ начин
 - Например, задния фон може да се рендерира чрез растеризация
 - Демонстрация [fract]
 - Може да симулираме DOF, като рендерираме само част от сцената (задната част), след което ѝ приложим замазване (gaussian blur), и впоследствие композираме с предната част от сцената, която е рязка

Shadow map

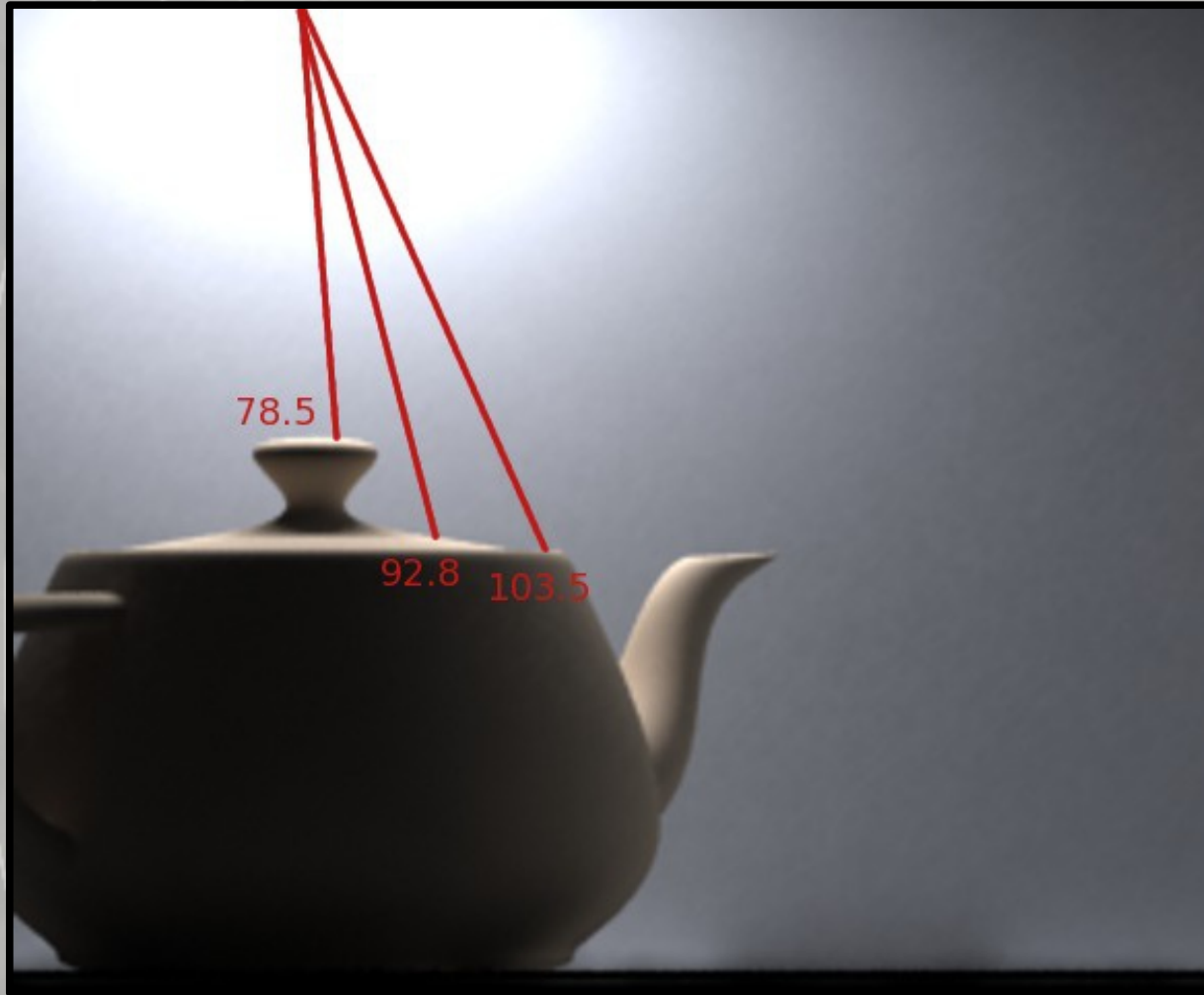
- Shadow map е една техника, позволяваща проверките за сенки да се правят без пускане на shadow лъчи; оригинално предложена от Lance Williams (1978г)
- Преди рендерирането на основното изображение се пуска един рендер от гледна точка на лампата. Обикновено се рендерира в 360° диапазон (напр., сферична камера). За всеки пиксел, търсеният резултат е разстоянието от лампата до най-близкия обект. Тези разстояния се записват в специална текстура, наречена shadow map (по една текстура за всяка лампа)

Shadow map

- Ето пример за shadow map
 - (заб: истинския shadow-map ще е 360-градусов, със сферична (или Cubemap) камера)
- Всеки пиксел показва дълбочината до най-близкият обект (т.е., depth map)

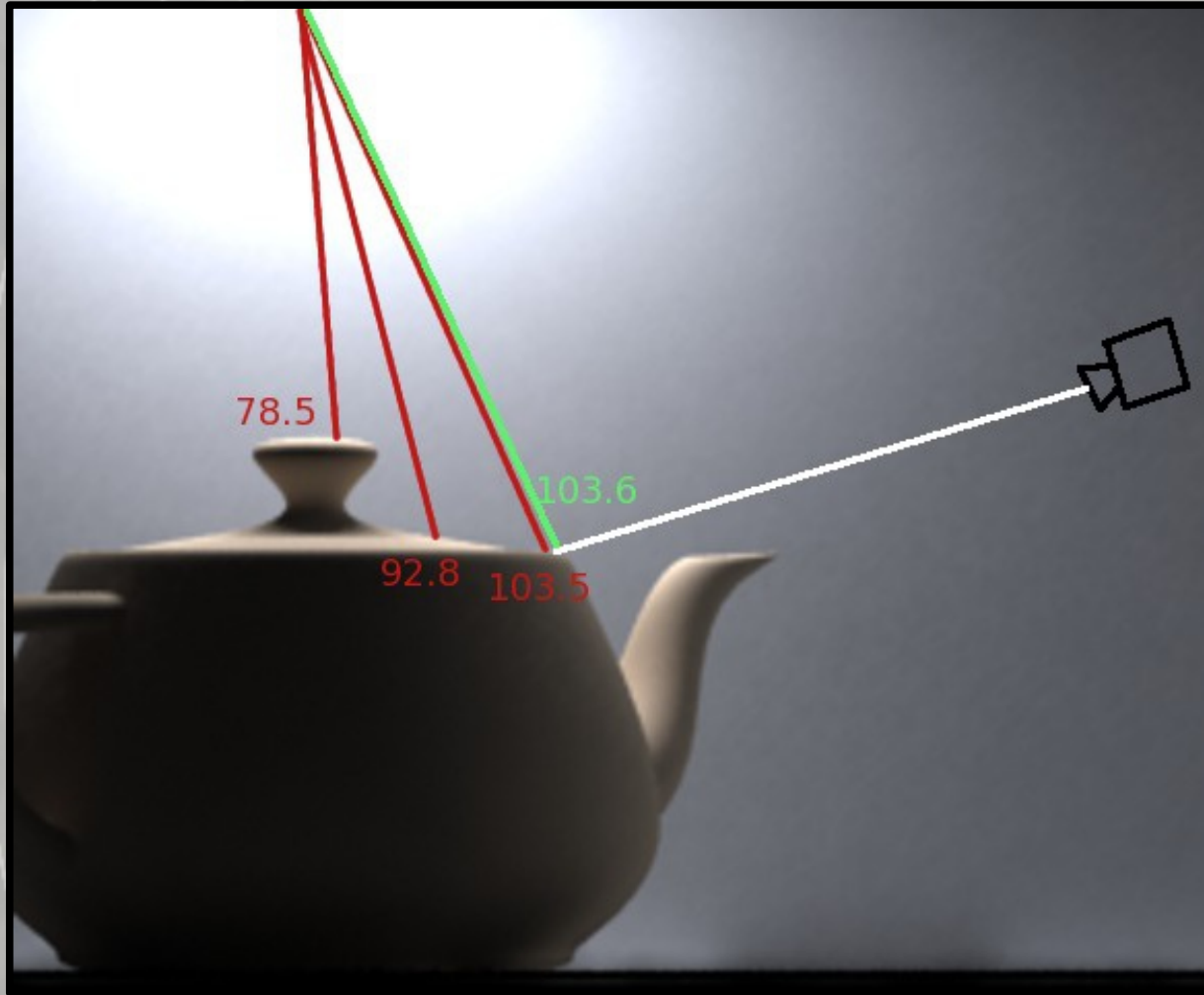


Shadow map



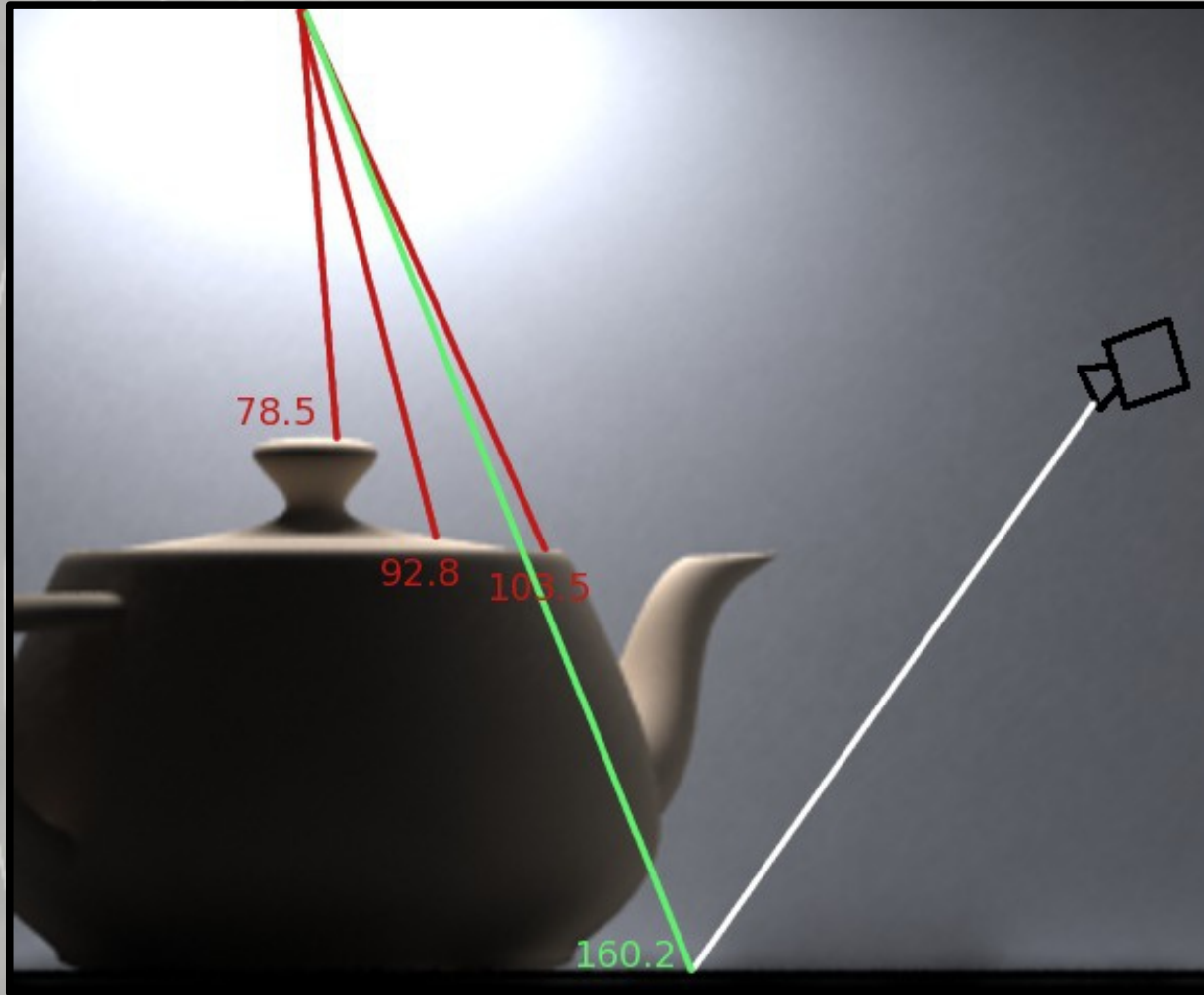
- Все едно, в shadow map-а имаме разстоянията от лампата до най-близките обекти във всяка посока
- С ограничена резолюция; по-висока резолюция ще даде по-точно представяне, но ще консумира повече памет

Shadow map



- Когато искаме да проверим дали дадена точка е в сянка, намираме разстоянието от нея до лампата и сравняваме с най-близката стойност в Shadow map-а. Ако са (приблизително) еднакви – няма сянка.

Shadow map

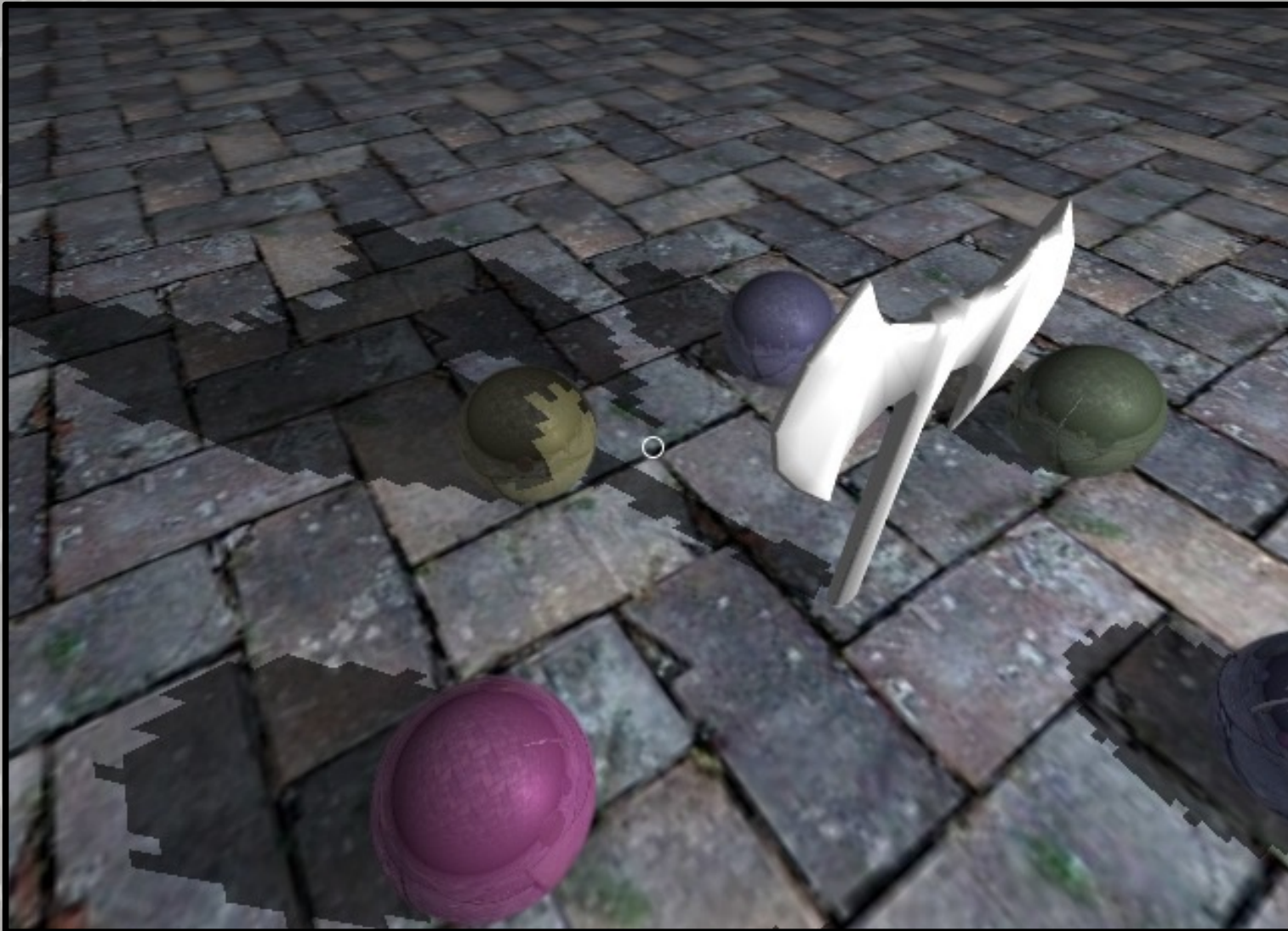


- Ако разстоянието от точката до лампата е значително по-голямо, значи сме в сянка
- Необходимо е да оставим малко аванс за да компенсираме неточностите заради дискретизацията в текстурата (shadow map bias)

Shadow map

- Предимства
 - Позволява значително намаляване на вторичните лъчи
 - Работи за всякакви геометрии и се вписва тривиално в един рейтрейсър
- Недостатъци
 - Разчита, че светлината и обектите са статични. При промяна на сцената, ще се наложи ново преизчисление на shadow map-а, което е потенциално тежко
 - Изисква се висока разделителна способност на shadow map текстурата, за да се избегнат назъбени сенки

Shadow map



- Кубичен shadow map, 128x128 текстури за всяка страна на куба

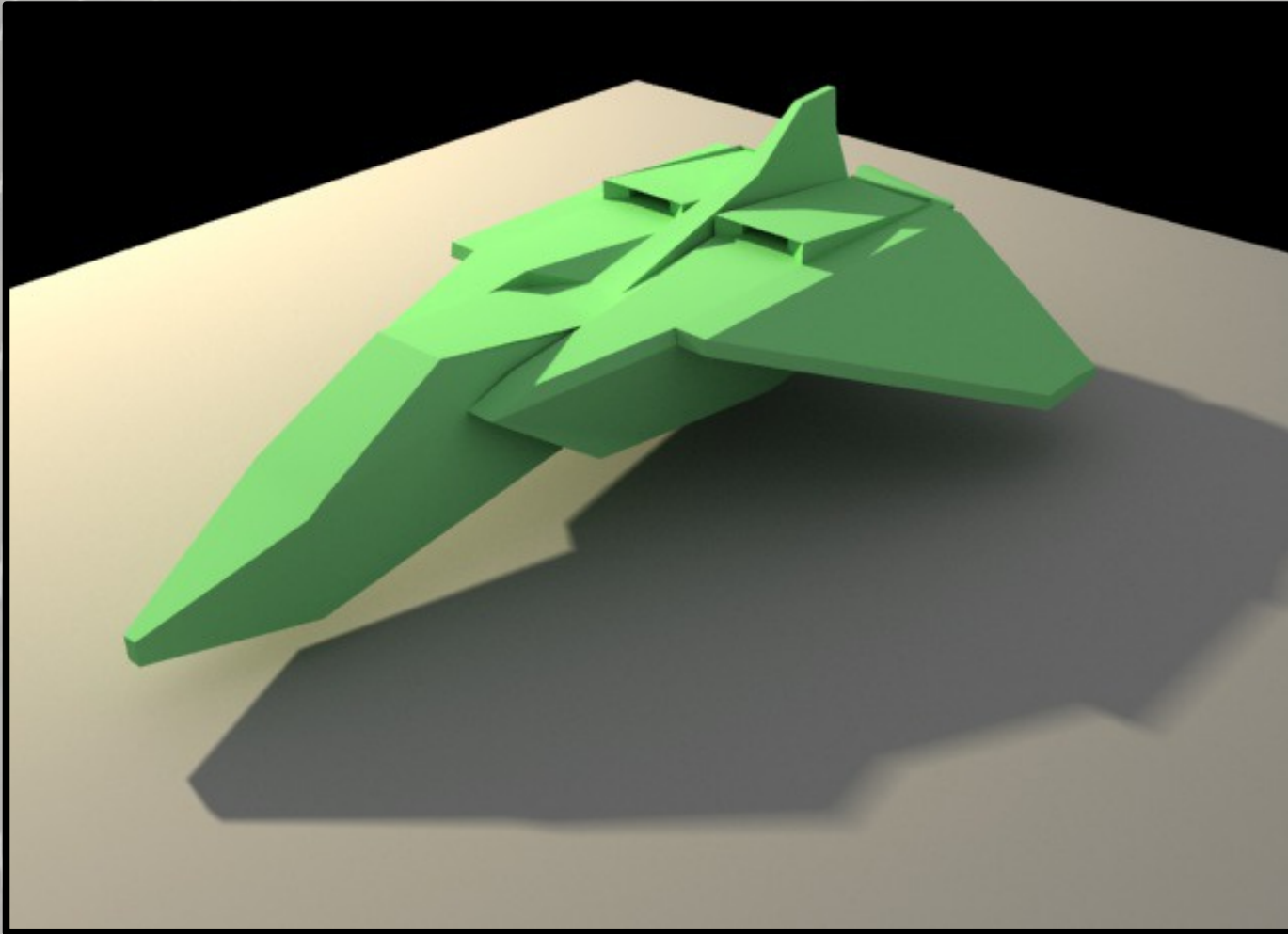
Shadow map

- Още недостатъци
 - Симулира само точкови светлинни източници; за да се симулира правоъгълна лампа, може да се приеме за съставена от множество малки точкови лампи, но това ще доведе до banding (слоеве) в полусенките (по подобие на феномена, който се получава, когато избираме фиксирани, а не случайни, точки от RectLight-a)
 - Демонстрация [fract]

Texture baking

- Ако част от лампите и обектите няма да се местят, то (по подобие на Shadow map, но в по-радикална посока), всички текстури в сцената могат да се „изпекат“, като им се сметнат крайните цветове, които биха излязли от тях при рендериране
 - Т.е., при рендериране с „изпечените“ текстури, вместо да смятаме осветление, сенки и прочее, просто вземаме цвета от текстурата и го връщаме
 - Например, ако имаме сцена с някаква равнина, и чайник отгоре ѝ, то, след изпичането, сянката на чайника ще се отпечата върху текстурата на равнината

Texture baking - пример



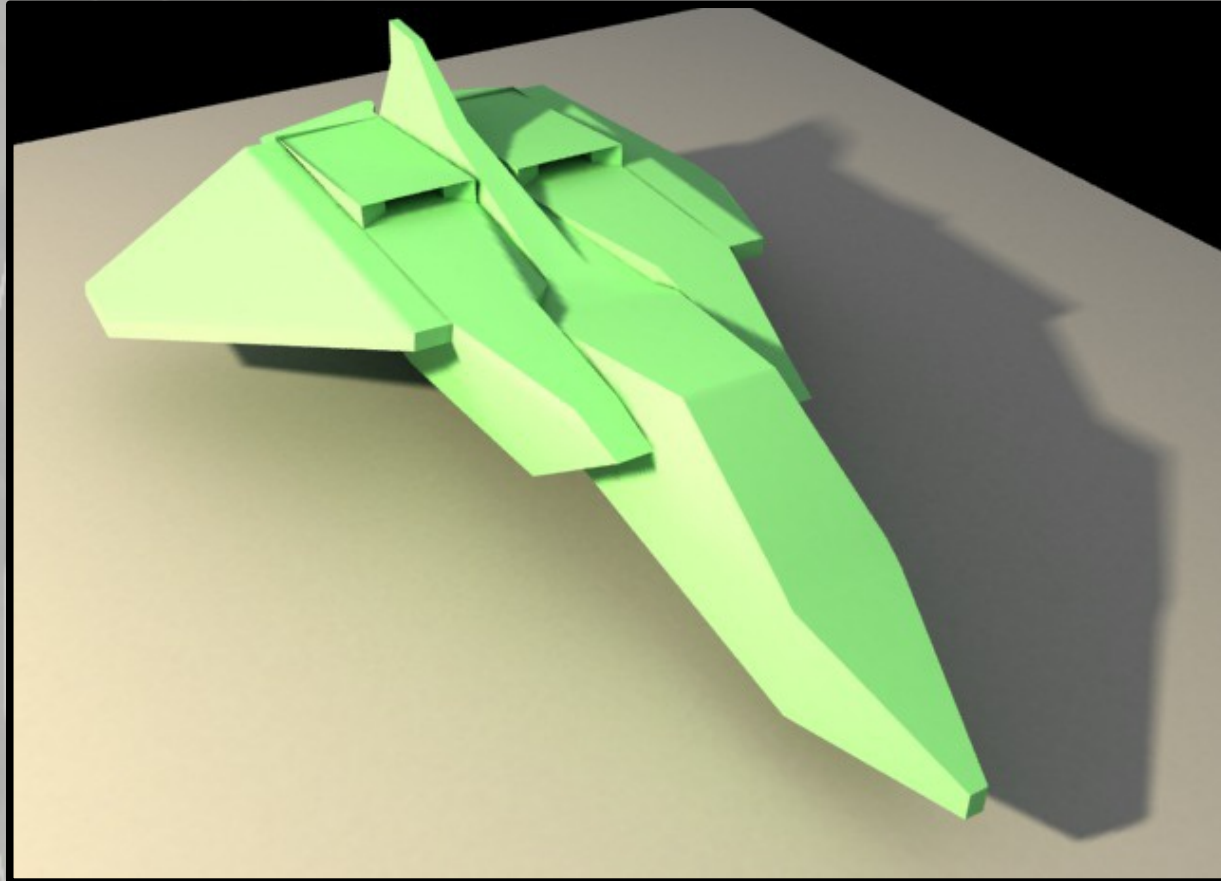
- Обикновен рендер с включен GI (отнема е 4.5 минути)

Texture baking - пример



- Изпечени текстури, съответно, на равнината под кораба, и на частите на кораба

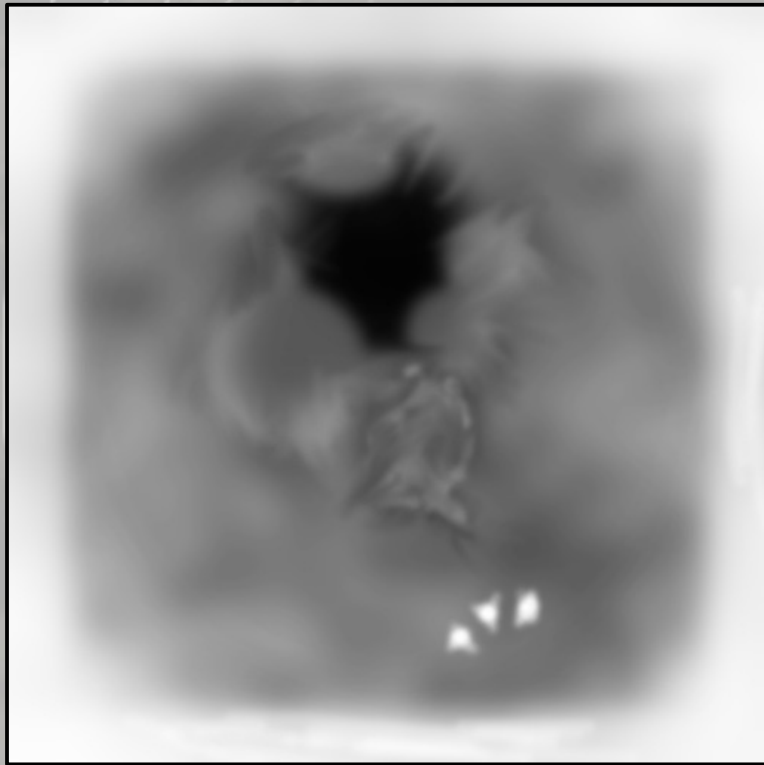
Texture baking - пример



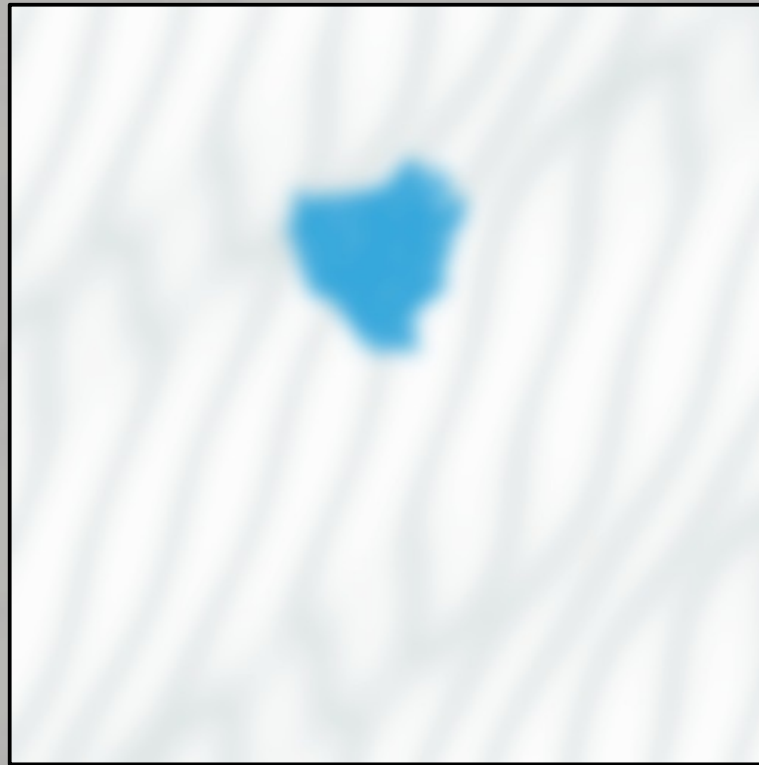
- Рендер без почти никакви ефекти – директно взимане на цветовете от текстурите (8 секунди)

Texture baking – пример

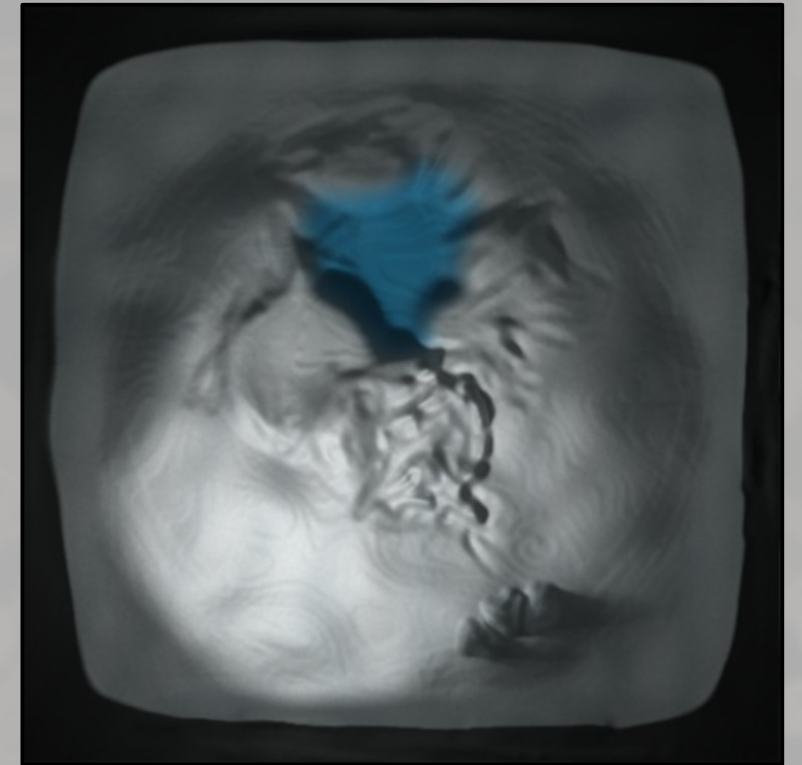
- Още един пример, със сцена, основно съставена от две релефни карти (и с много индиректно осветление):



• Релефна карта



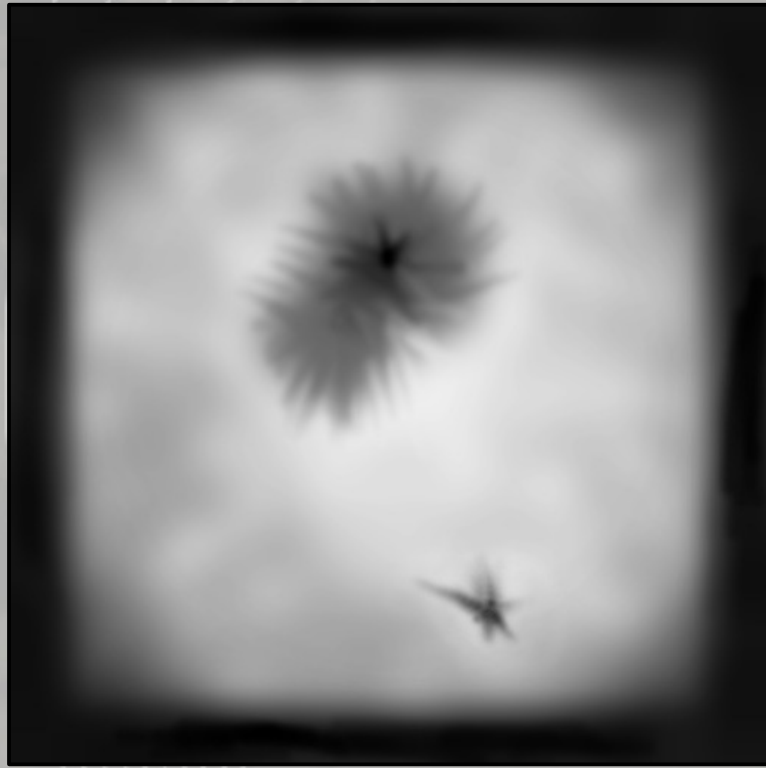
Неопечена текстура



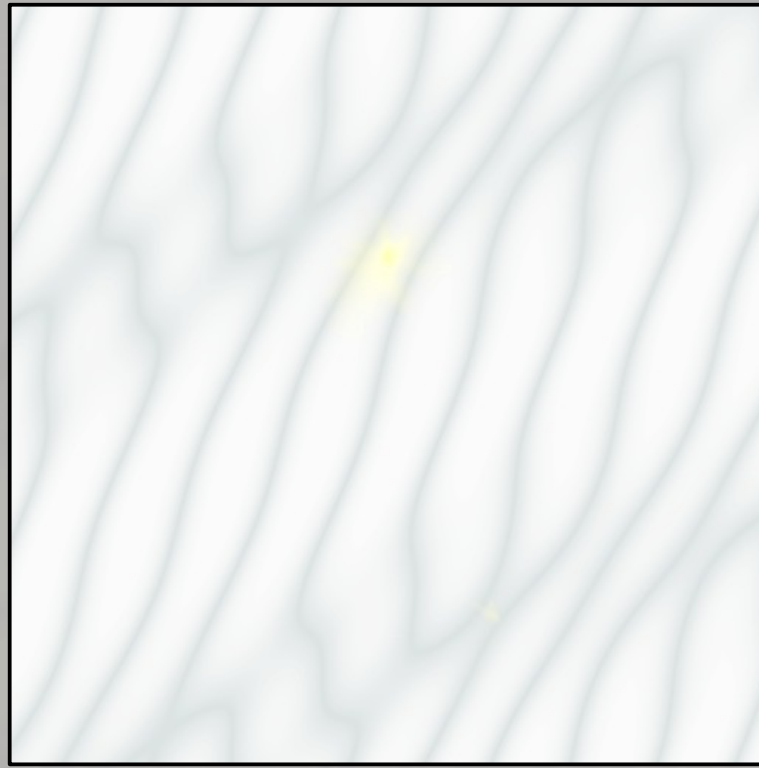
Опечена текстура

Texture baking – пример

- Аналогично за „тавана“:



• Релеф



Неопечена текстура

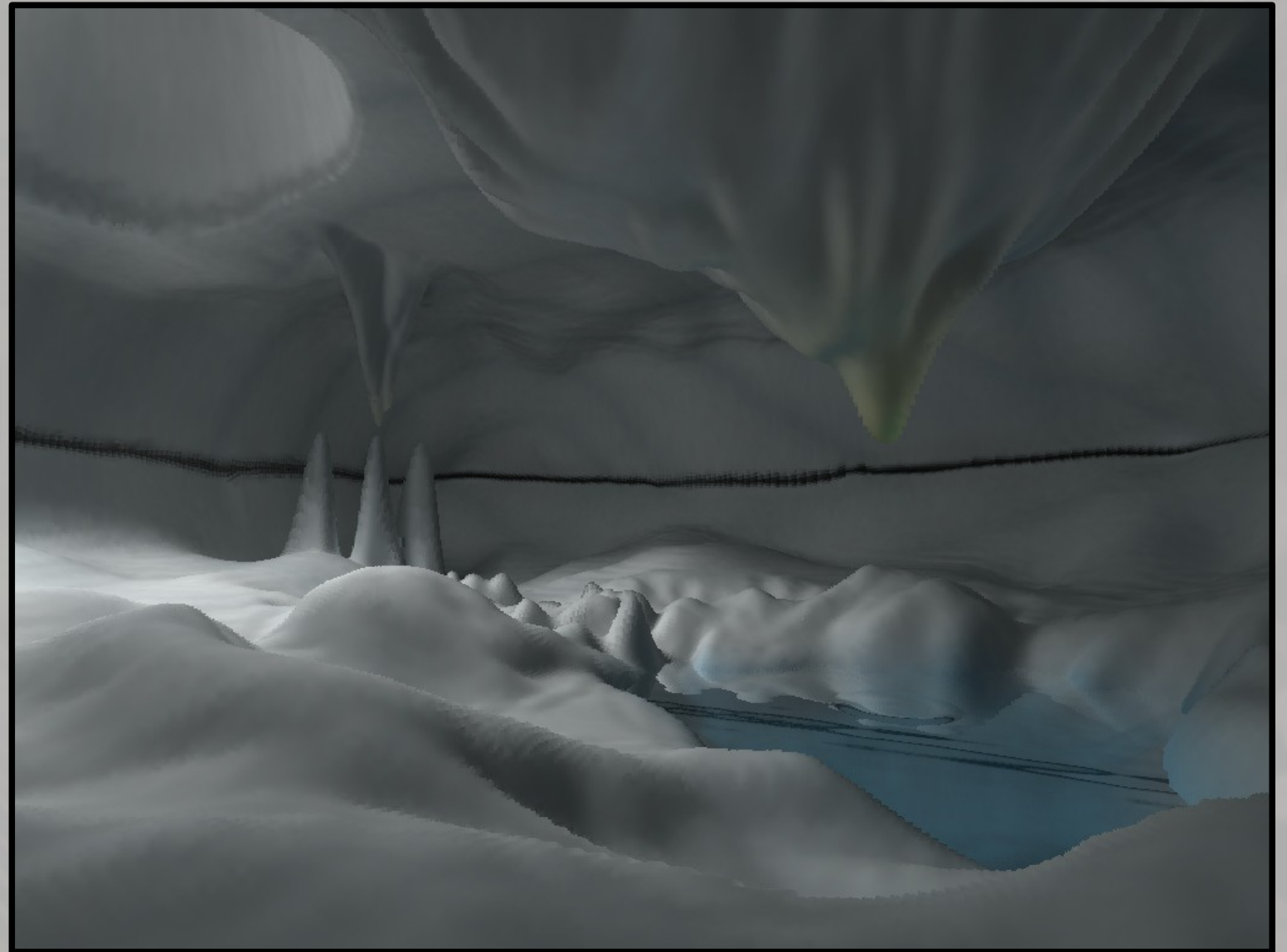


Опечена текстура

Texture baking – пример

Резултат:

(при трасирането, цвета, който трябва да се върне, се взима директно от текстурата, без никакви изчисления на осветление; това позволява рендериране с realtime скорости)



Адаптивен рейтрейсинг

- Основната идея на адаптивния рейтрейсинг е да намалим броя на първичните лъчи (евентуално да ги направим по-малко от броя на пикселите), като засечем „безинтересните“ области в изображението и там пускаме лъчите по-нарядко. За пикселите, за които няма лъчи, интерполираме между най-близките стойности, за които сме пуснали лъчи

Адаптивен рейтрейсинг

- Разделяме кадъра на квадратчета с някаква големина, например 8x8 (не е добре да са прекалено големи, че може да изпуснем някои фини детайли от сцената така)
- За всяко квадратче:
 - Пускаме лъчи в краищата на квадратчето и взимаме пълна информация за този пиксел: цвят, id на ударения обект, текстурни координати, дълбочина (съвкупността от тази информация обикновено се нарича *фрагмент*)
 - По някакъв евристичен критерии решаваме дали квадратчето е „интересно“

Адаптивен рейтрейсинг

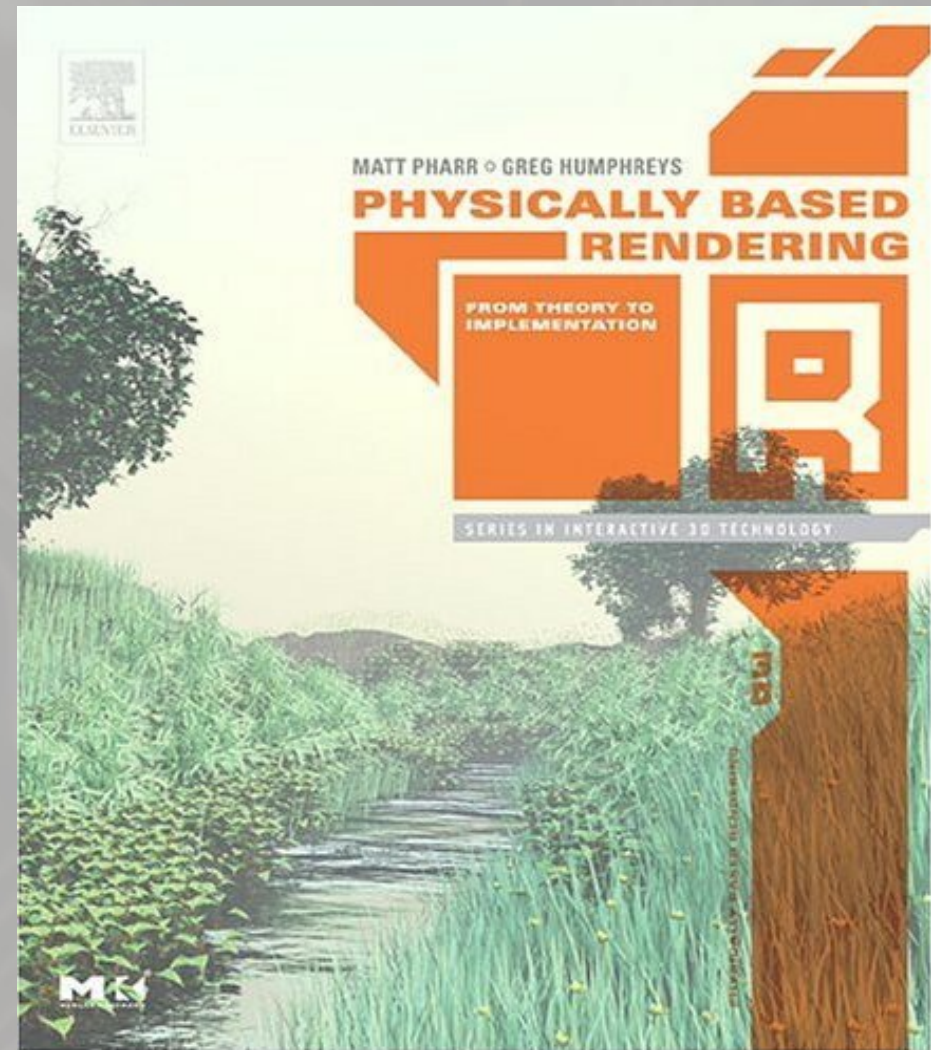
- Ако е квадратчето „интересно“, то го разделяме на 4 квадратчета (в случая 4x4). Пускаме необходимите лъчи за тях (още 5 фрагмента). С ново-получените данни решаваме за всяко 4x4 квадратче дали също е интересно, и ако е, разделяме отново и т.н.
- Ако квадратчето не е интересно, то може да интерполираме по някакъв начин информацията между фрагментите, за да сметнем цветовете на пикселите, за които нямаме фрагменти
 - Най-просто, може да интерполираме 4-те цвята (bilinear filtering), но това обикновено изглежда много грозно
 - Доста по-добър вариант е да интерполираме текстурните координати и да вземаме цветовете от текстурата

Критерии за „интересност“ - примери

- Какво критерии за „интересност“ на квадратчетата може да ползваме:
 - Depth – ако някой от фрагментите е доста по-далече, значи квадратчето е на някакъв ръб на обект, значи е интересно
 - Цвят – ако цветовете са прекалено различни
 - Id – ако 4-те Id-та не принадлежат на един и същ обект
 - ...и други, специфични за случая

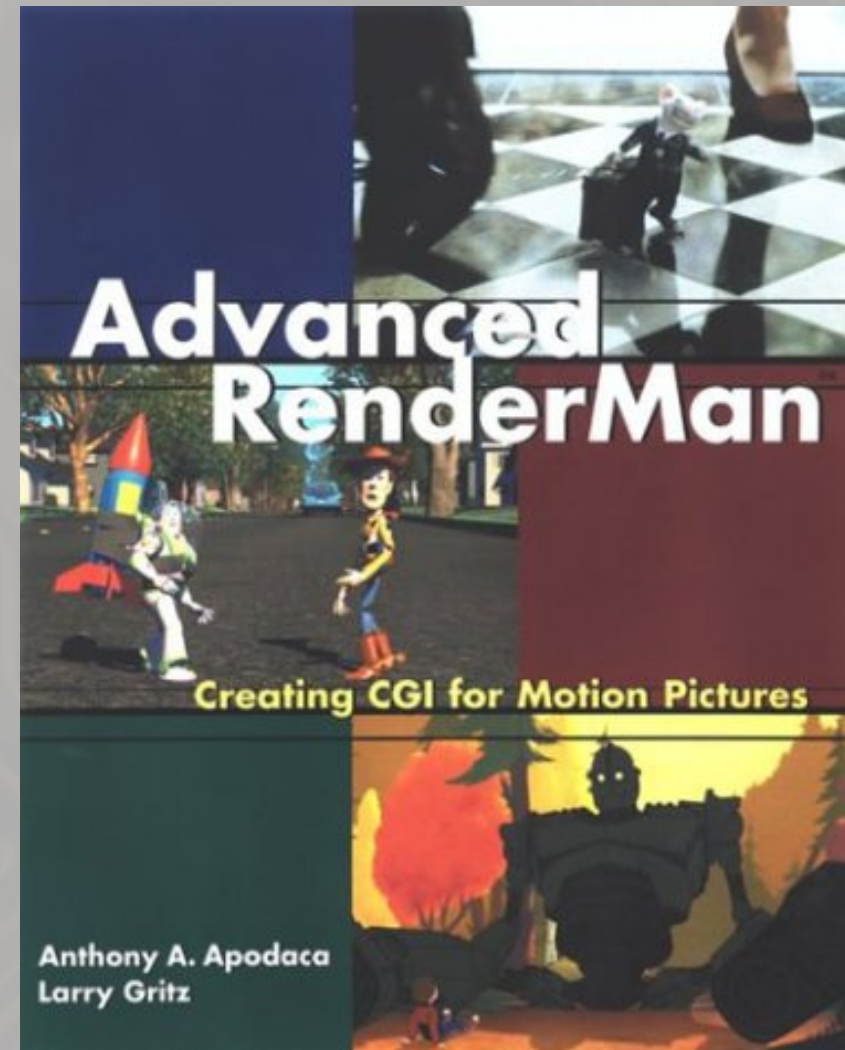
Books

- Physically Based Rendering
 - **Must-read** =-)
- Source at
 - github.com/mmp/pbrt-v1/
 - github.com/mmp/pbrt-v2/



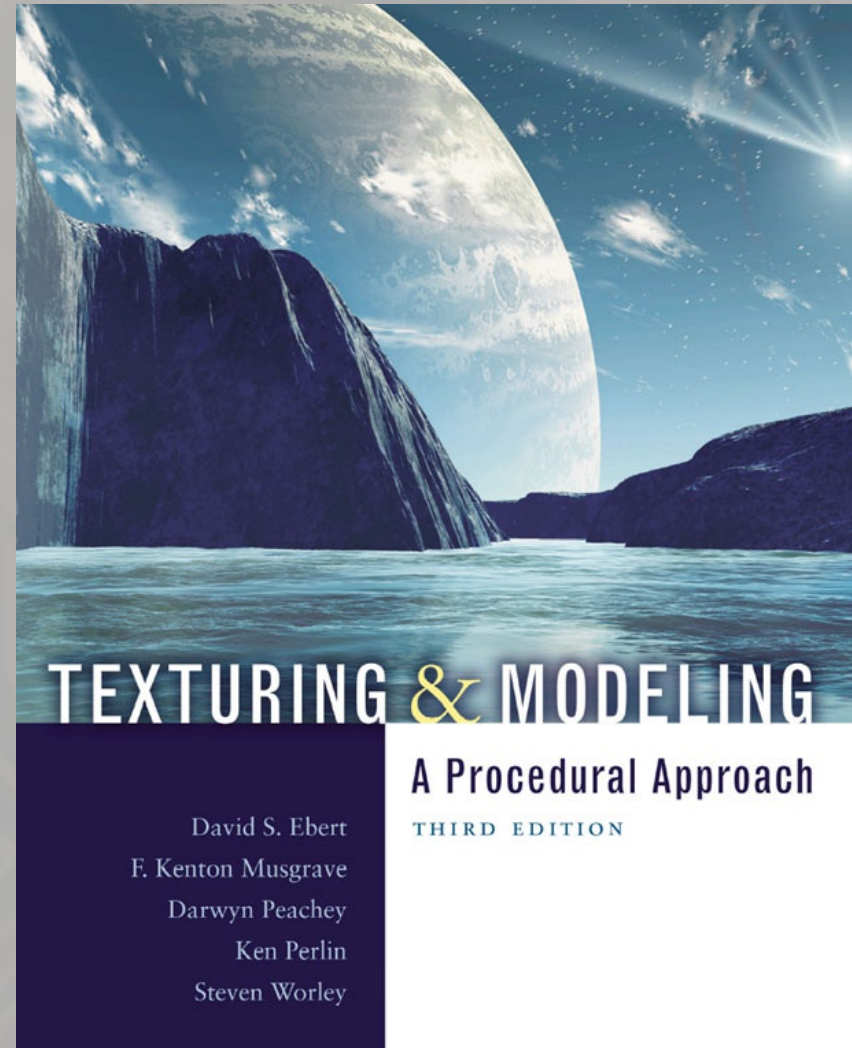
Books / RenderMan

- Also RiSpec
- A bunch of SIGGRAPH courses at renderman.org



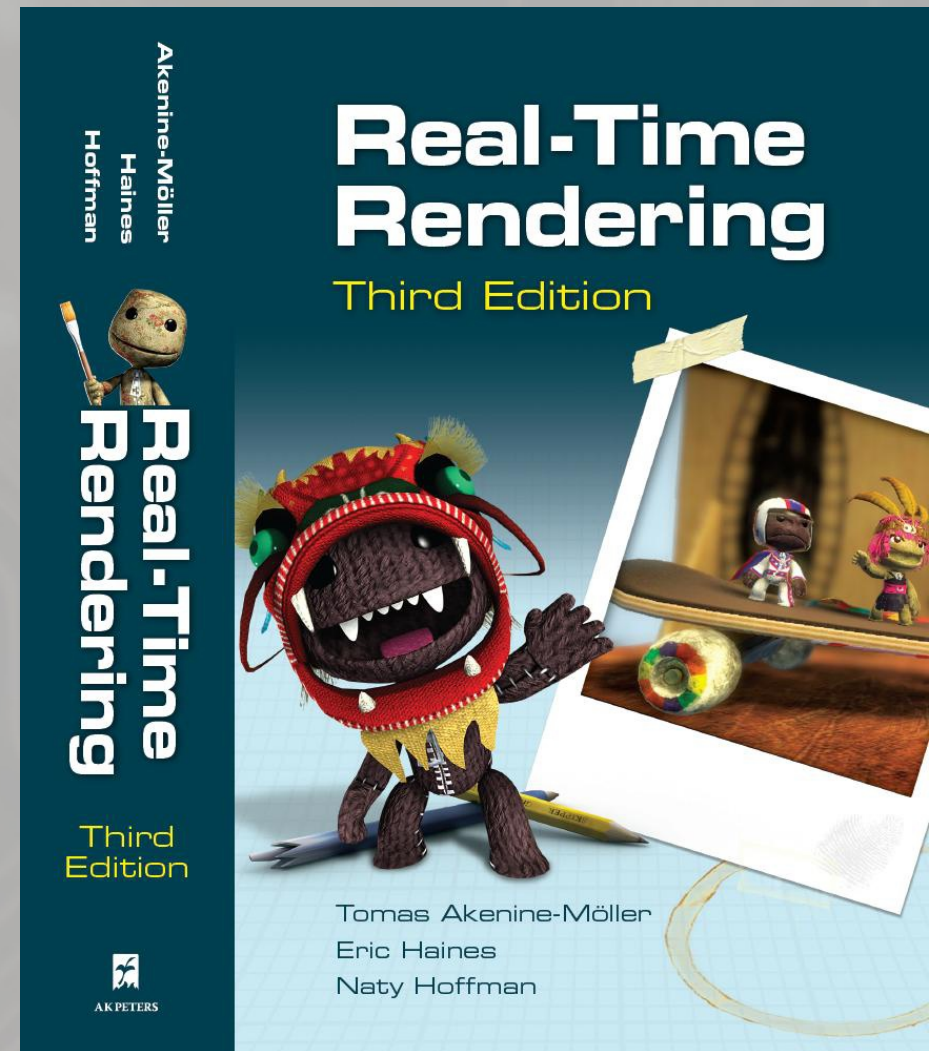
Books / Procedural

- Nice stuff



Books / Real time

- Also
 - OpenGL Distilled
 - OpenGL Shading Language



Papers

- SIGGRAPH papers index
 - <http://kesen.huang.googlepages.com/>
- Various
 - <http://graphics.pixar.com/research/>
 - <http://www.crytek.com/cryengine/presentations/>
 - <http://www.graphics.stanford.edu/papers/>
 - <http://raytracey.blogspot.com>

Communities

- Ray tracing forum
 - ompf.org/forum/
- Bulgarian game devs
 - gamedev-bg.net
- 3d artists' forums&lists
 - ...

Work in BG

- Chaos Group
- Haemimont Games, Black Sea – Crytek, Ubisoft Bulgaria, Gameloft, Masthead, Coherent Labs, etc.
- World-wide FX, etc.
- Work for yourself!

Work world wide

- Game industry - huge
 - Crytek, Blizzard, id, EA, Naughty Dog, etc...
- Movie industry
 - Pixar, Blue Sky, Sony Imageworks, Industrial Light & Magic, Weta Digital, etc...
- Various developing and consulting companies

To infinity & beyond!

