3D графика и трасиране на лъчи v.3.0

http://raytracing-bg.net/

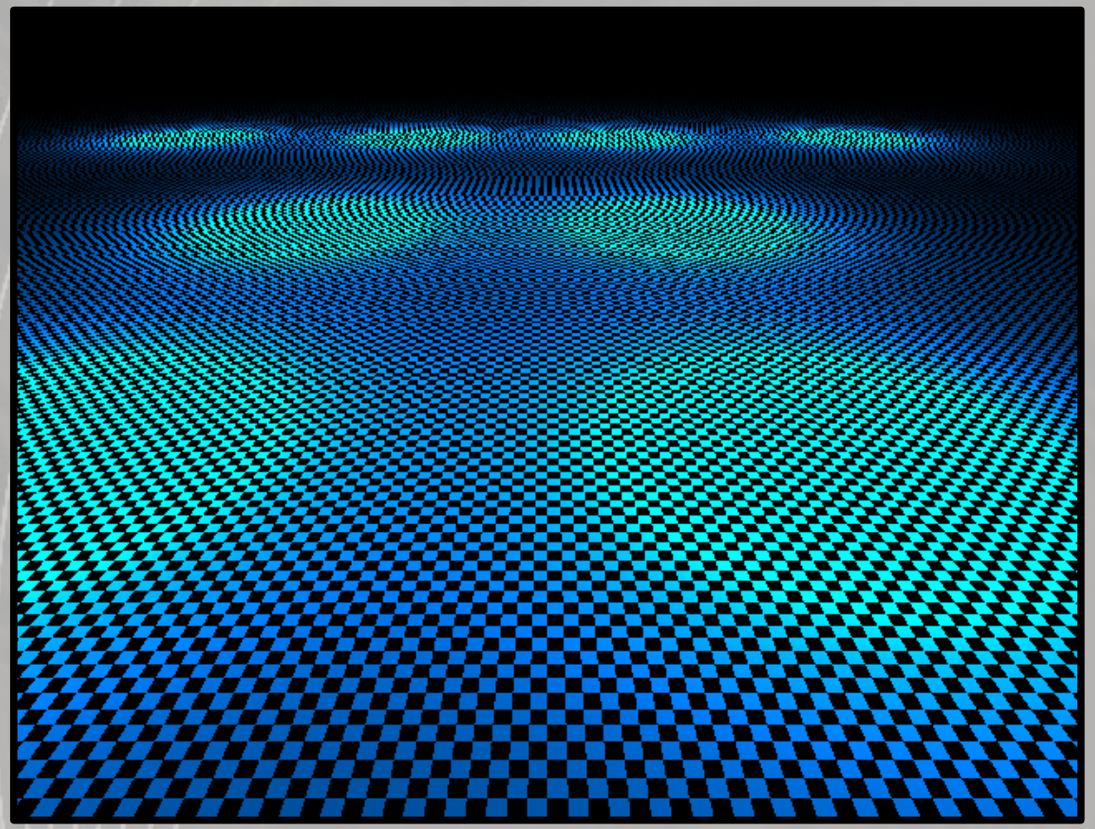
Тема 6

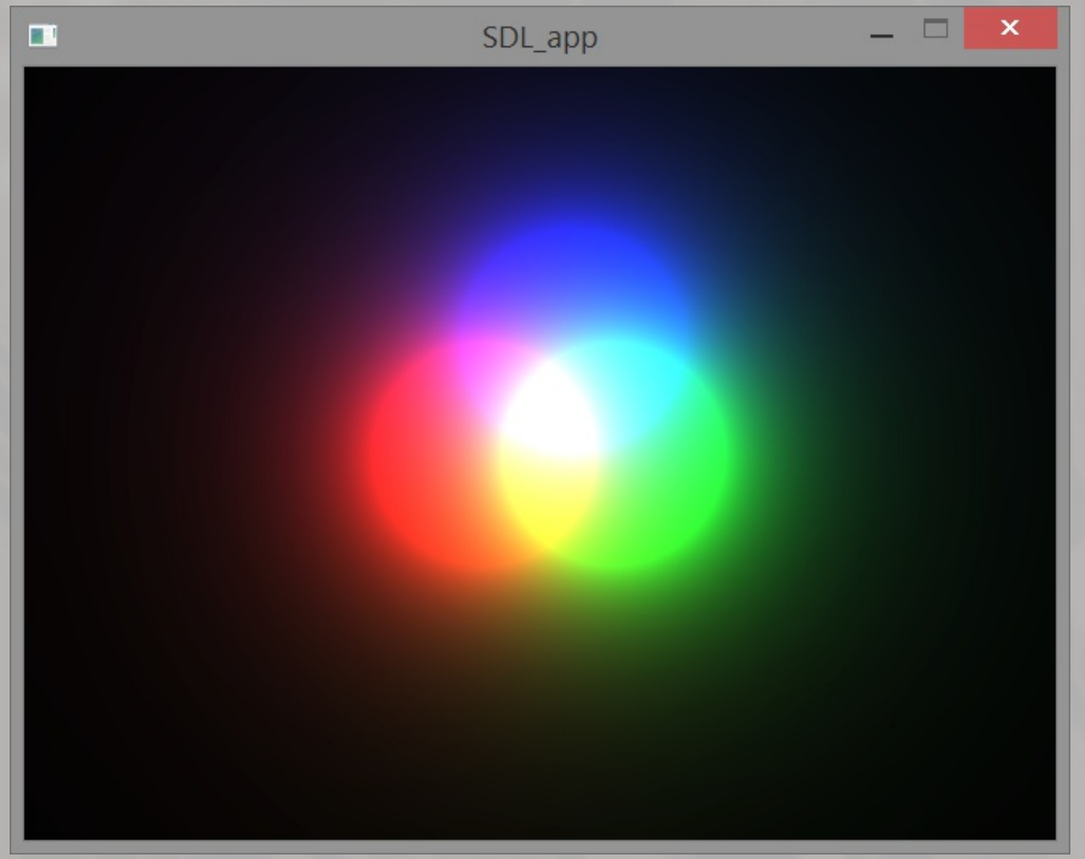
Заден план
Отражения и пречупвания
Текстури

Анонси

- Тест #1 на 29ти Ноември
- Домашни
- Генериране на сцената
  - От следващата лекция няма да я генерираме, а ще четем от файл

BoFHs





Съдържание

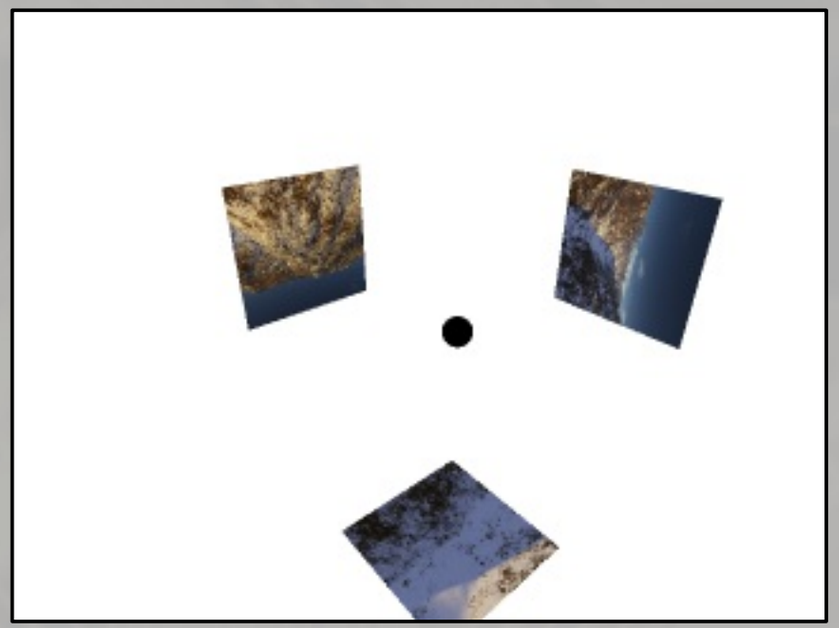
- Заден фон (обкръжение)
- Отражение
- Пречупване
- Текстури
- „Грапави отражения“
- Хроматично пречупване
- Caustics/Photon Mapping

Заден фон

- Задният фон определя какво да се вижда извън сцената, т.е. когато raytrace() не удари нито един Node
  - Досега връщахме черно
- Обикновено задния фон запълваме с картинка, вместо с обекти – например, гора, планини, небе, космос
  - Изгледът на задния фон не зависи от позицията на камерата – само от посоката й!
- Почти винаги ще ползваме някаква картинка, взета от текстура, за заден фон

Заден фон

- Задният фон се възприема като някакъв безкрайно голям, обхващащ сцената обект, например куб или сфера. Сцената се намира в средата на този обект [cubemap.m4v]



Интерфейс за заден фон

```
class Environment {
public:
   virtual Color getEnvironment(const Vector& dir) = 0;
};
```

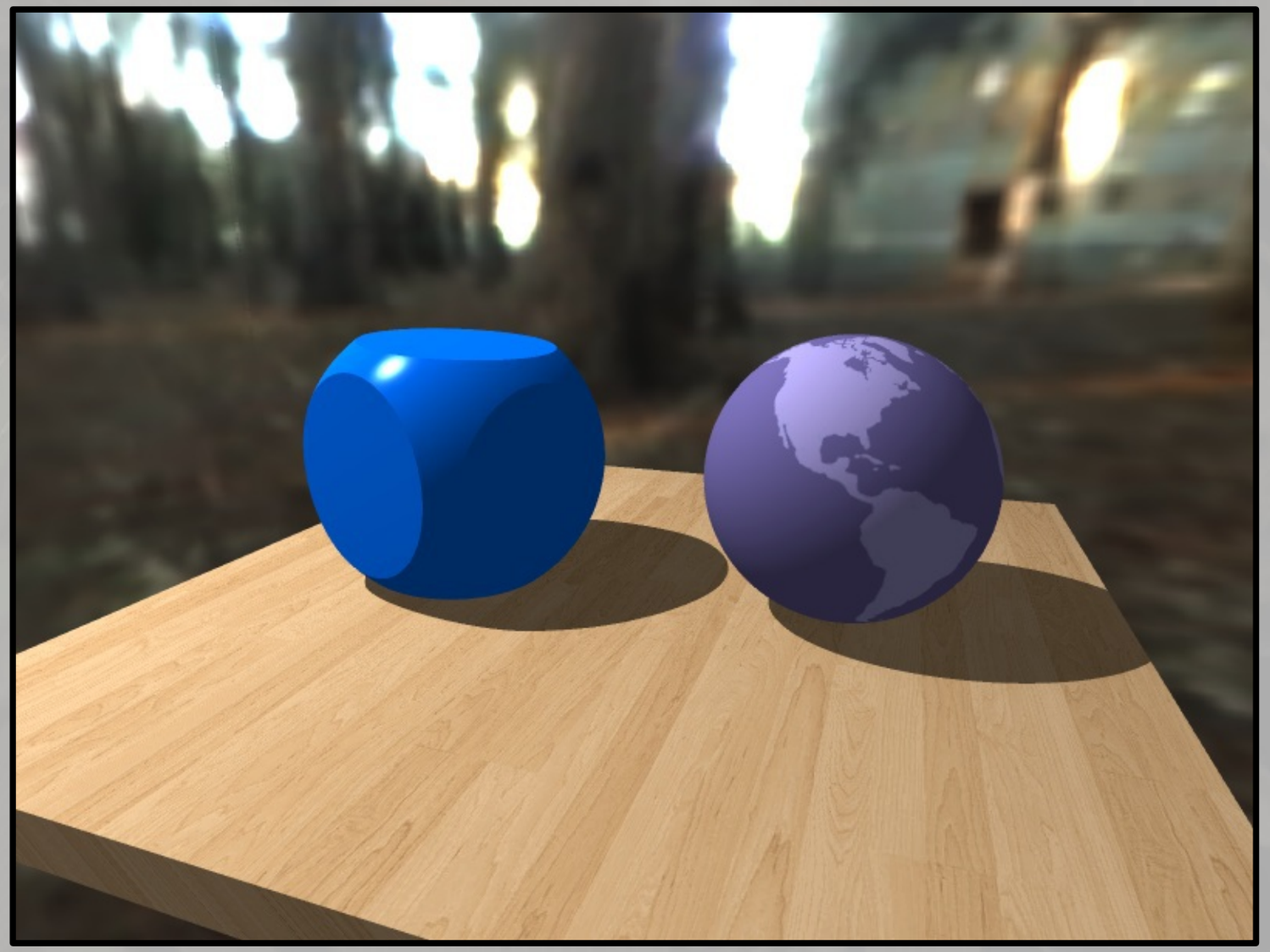
Видове заден фон

- Сферичен (Spherical mapping)
  - Използва се само една картинка, облепена върху сфера
  - Посоката на лъча (ray.dir – единичен вектор) се превръща в сферични координати и те се ползват да се индексира текстурата
- Кубичен (Cubemap)
  - Използват се 6 картинки, за 6-те стени на куб
  - Най-големият по абсолютна стойност компонент на вектора определя коя страна удряме; а къде точно в съответната текстура сме се определя от другите 2 компонента

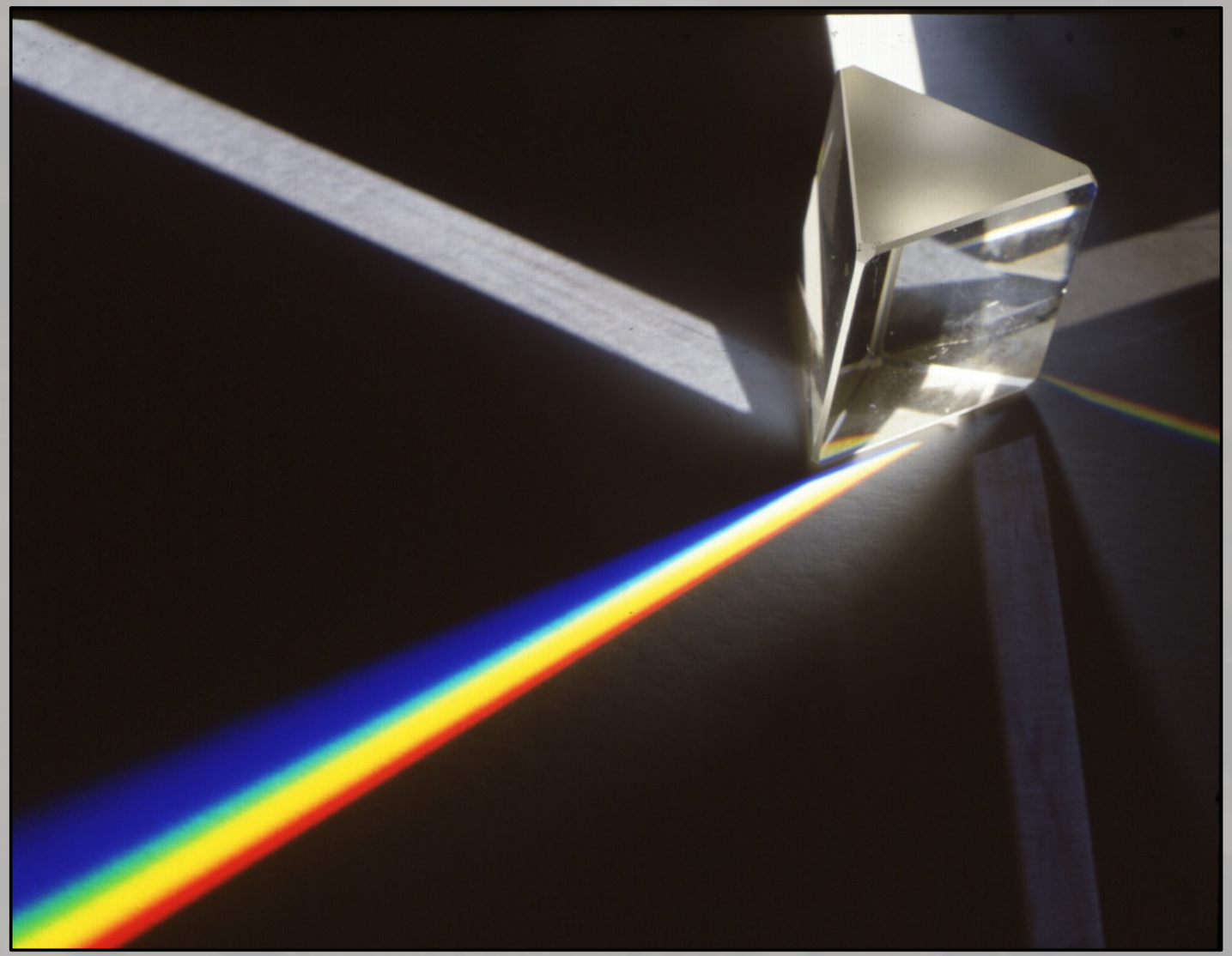
Видове заден фон

- Кубичния фон често се предпочита пред сферичния, тъй като разтеглянето на пикселите е по-равномерно, и може да ползваме по-малки картинки за едно и също качество

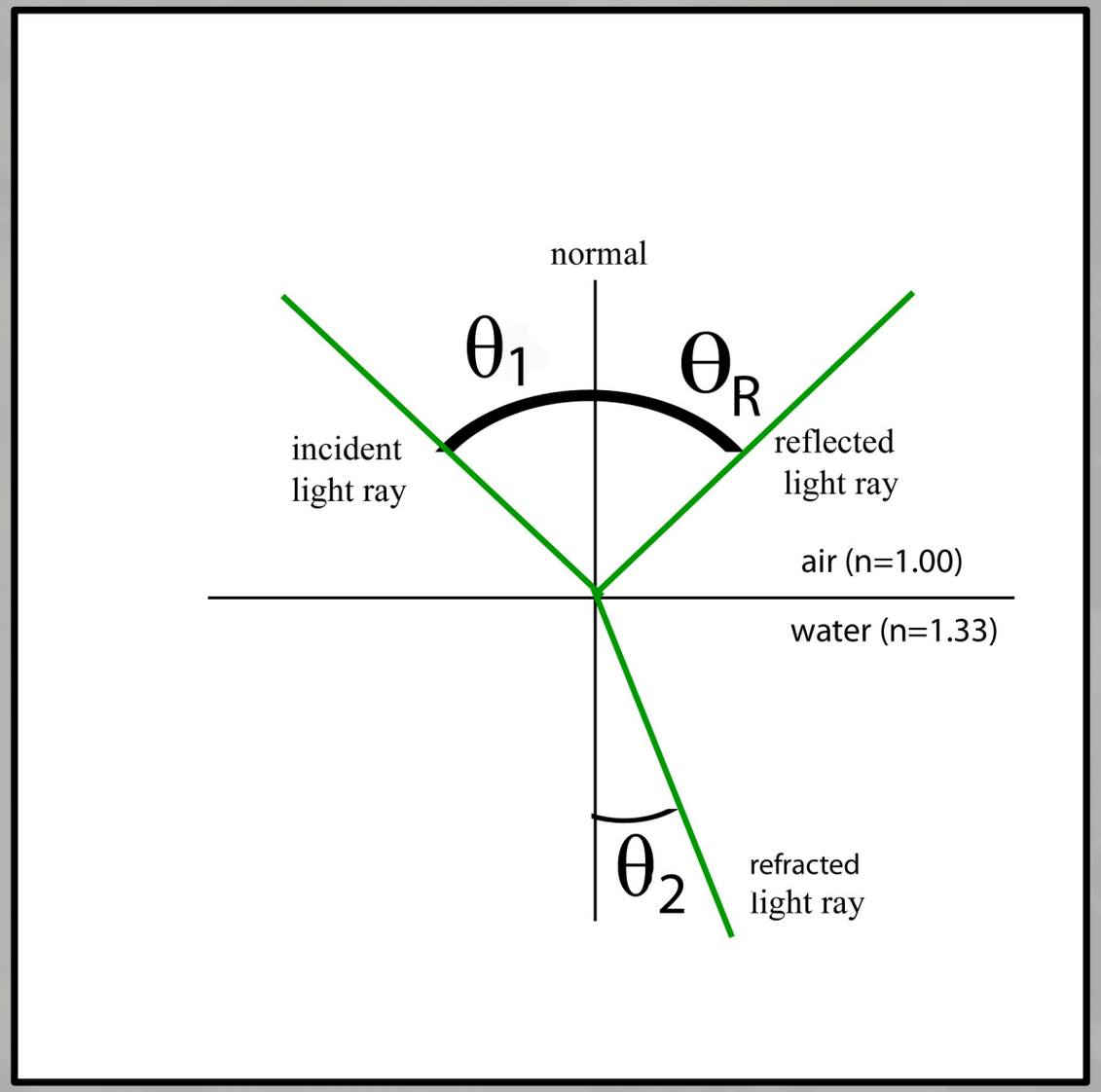
Какво имаме досега



Reflection & Refraction

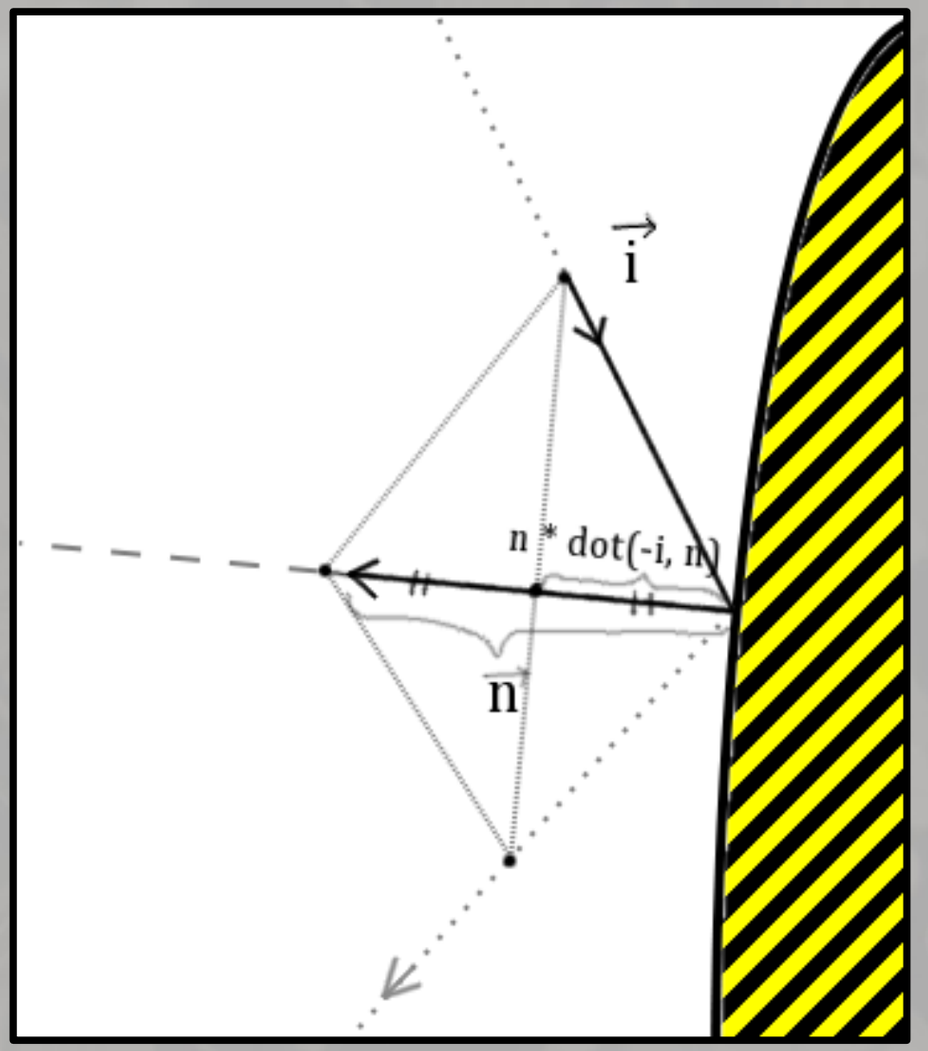


Reflection & Refraction



Reflect

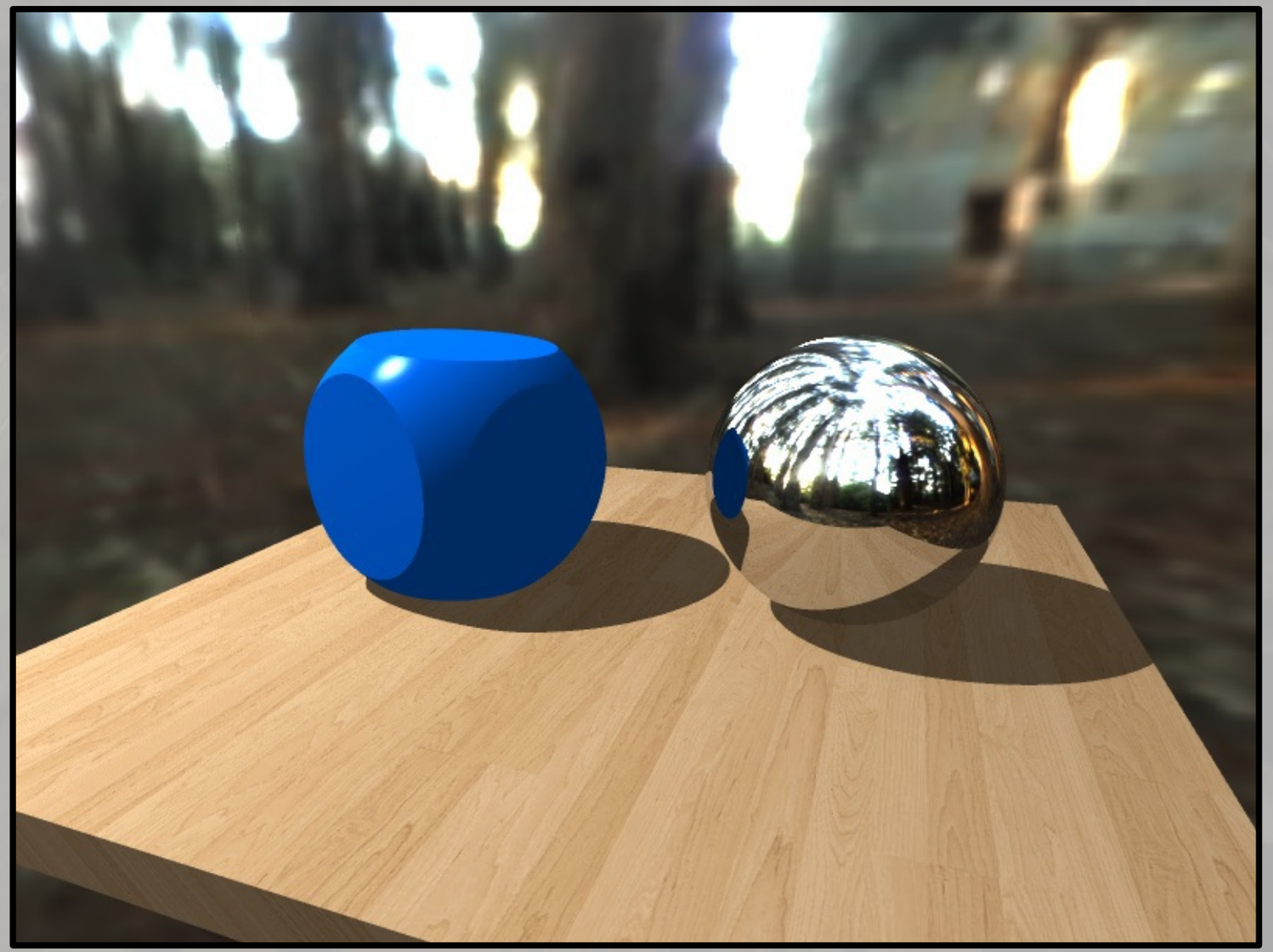
```
// i е входящия вектор (от камерата)
// n е нормалата
inline Vector reflect(const Vector& i,
                      const Vector &n)
{
    return i + 2 * dot(-i, n) * n;
}
```



Reflection v0.1

```
Color Reflection::shade(Ray ray, const IntersectionInfo& info)
{
    extern Color raytrace(Ray ray);

    Vector n = faceforward(info.norm, ray.dir);
    Ray newRay = ray;
    newRay.start = info.p + 1e-6 * n;
    newRay.dir = reflect(ray.dir, n);
    return raytrace(newRay) * 0.8f;    // 20% загуба на енергия
}
```

Reflection v0.1



Отражения и ограничаване на рекурсията

- Рекурсивното извикване на raytrace() от шейдъра създава опасност от бездънна рекурсия, ако имаме два отражателни обекта, и те са разположени по такъв начин, че да си отразяват лъчите напред-назад до безкрайност
  - Например, две огледала, разположени едно срещу друго
- За да решим този проблем, ще въведем ограничение на максималния брой отражения, които raytrace може да трасира...

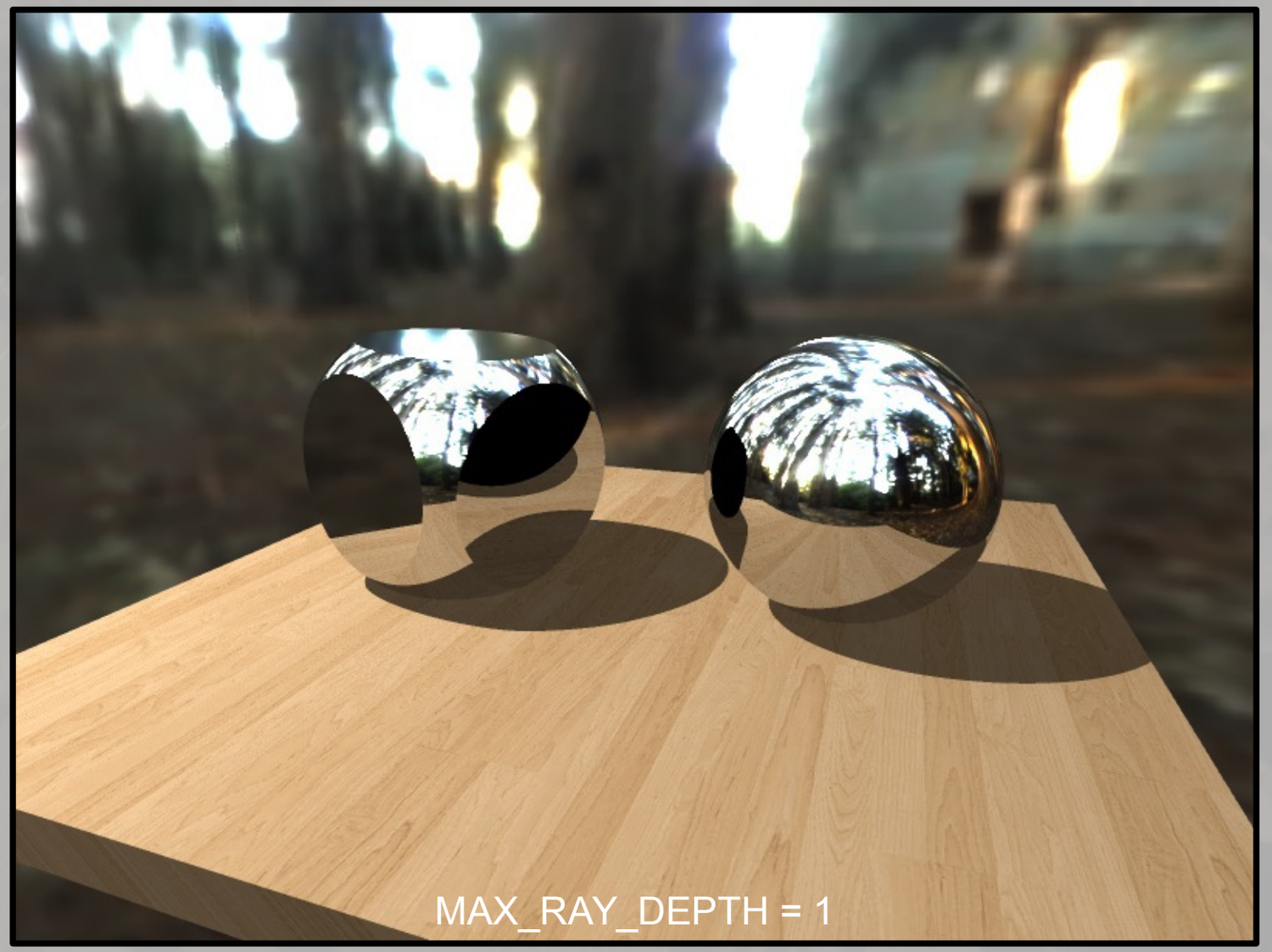
Adding reflection/refraction levels to Ray

```
struct Ray {
    Vector start, dir;
    int depth;
    Ray() { depth = 0; }
};

....
Color raytrace(Ray ray)
{
   if (ray.depth > MAX_RAY_DEPTH) return Color(0, 0, 0);
   ....
```

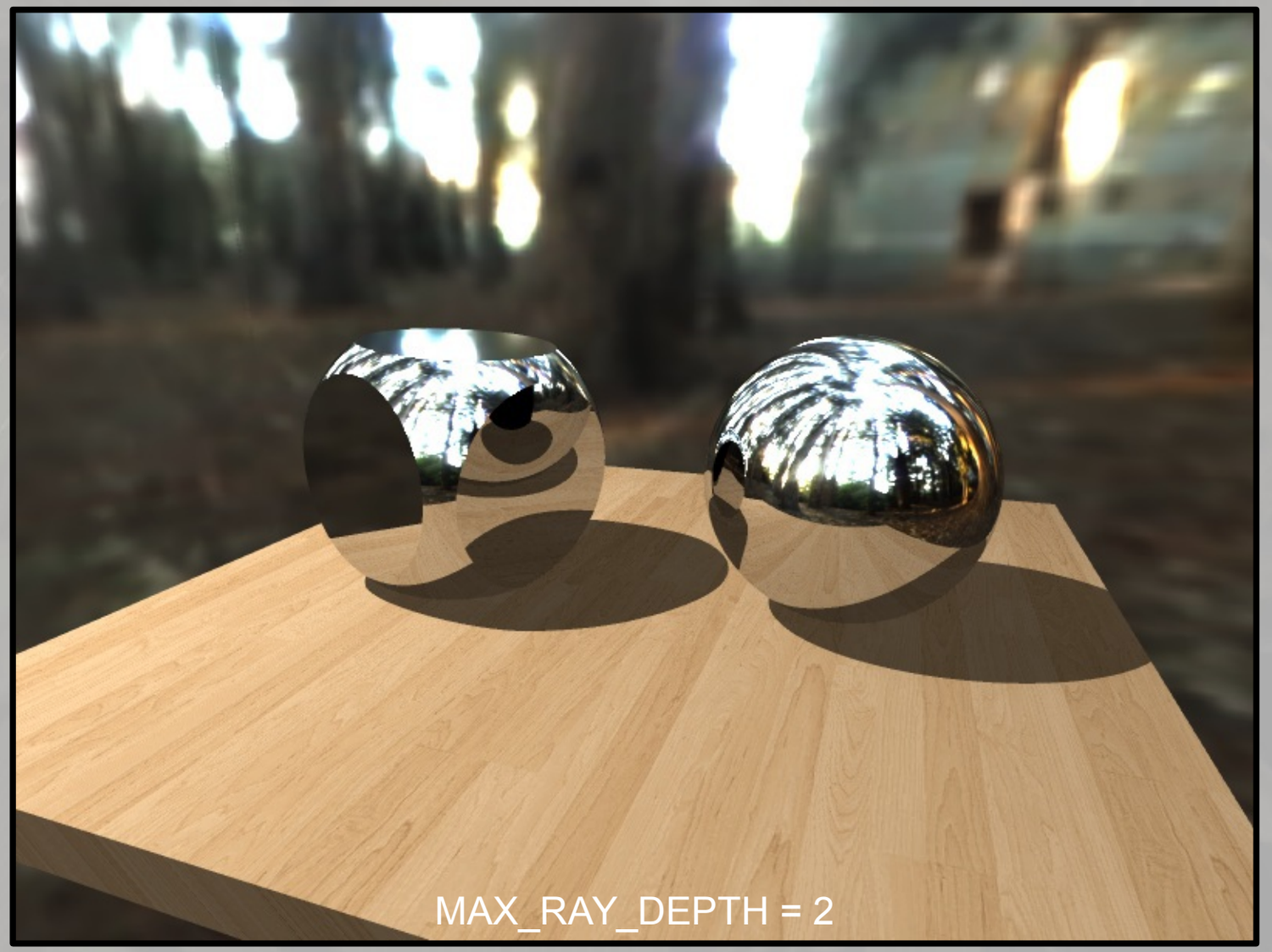
Reflection v0.2

```
Color Reflection::shade(Ray ray, const IntersectionInfo& info)
{
    extern Color raytrace(Ray ray);

    Vector n = faceforward(info.norm, ray.dir);
    Ray newRay = ray;
    newRay.start = info.p + 1e-6 * n;
    newRay.dir = reflect(ray.dir, n);
    newRay.depth = ray.depth + 1;
    return raytrace(newRay);
}
```

Reflection v0.2



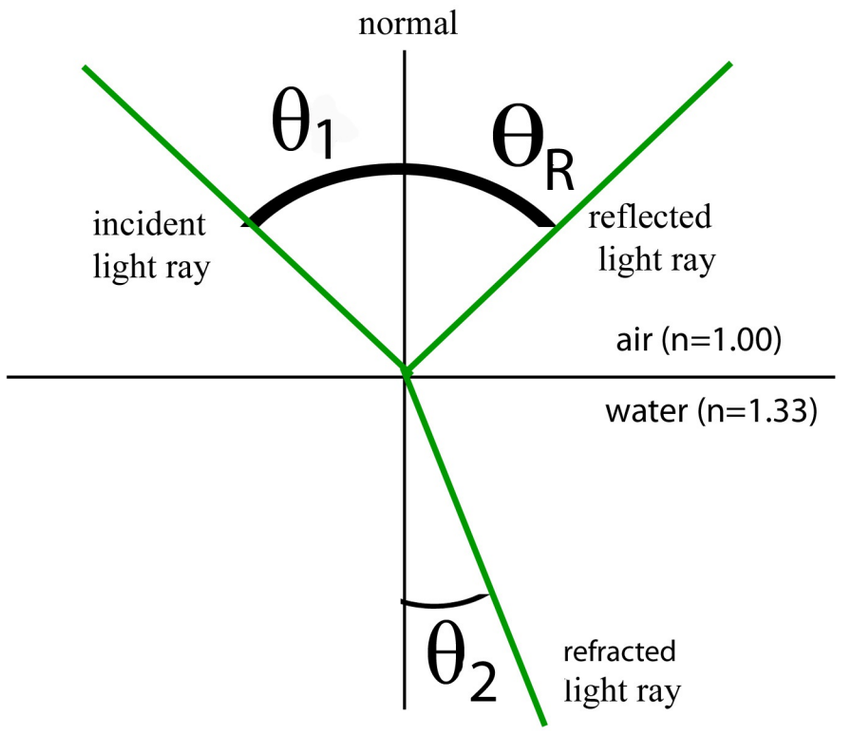
Reflection v0.2



Reflection v0.2

Reflection & Refraction

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{v_1}{v_2} = \frac{n_2}{n_1}$$



- Θ_1 е ъгълът между нормалата и входящия лъч
- Θ_2 е ъгълът между нормалата и изходящия лъч
- n1 изразява „плътността“ на материята, от която идваме
- n2 изразява „плътността“ на материята, в която отиваме

Refract

```
// ior = eta2 / eta1
inline Vector refract(const Vector& i, const Vector& n, float ior)
{
    float NdotI = (float) (i * n);
    float k = 1 - (ior * ior) * (1 - NdotI * NdotI);
    if (k < 0.0f)       // Check for total inner reflection
        return Vector(0, 0, 0);
    return ior * i - (ior * NdotI + sqrt(k)) * n;
}
```

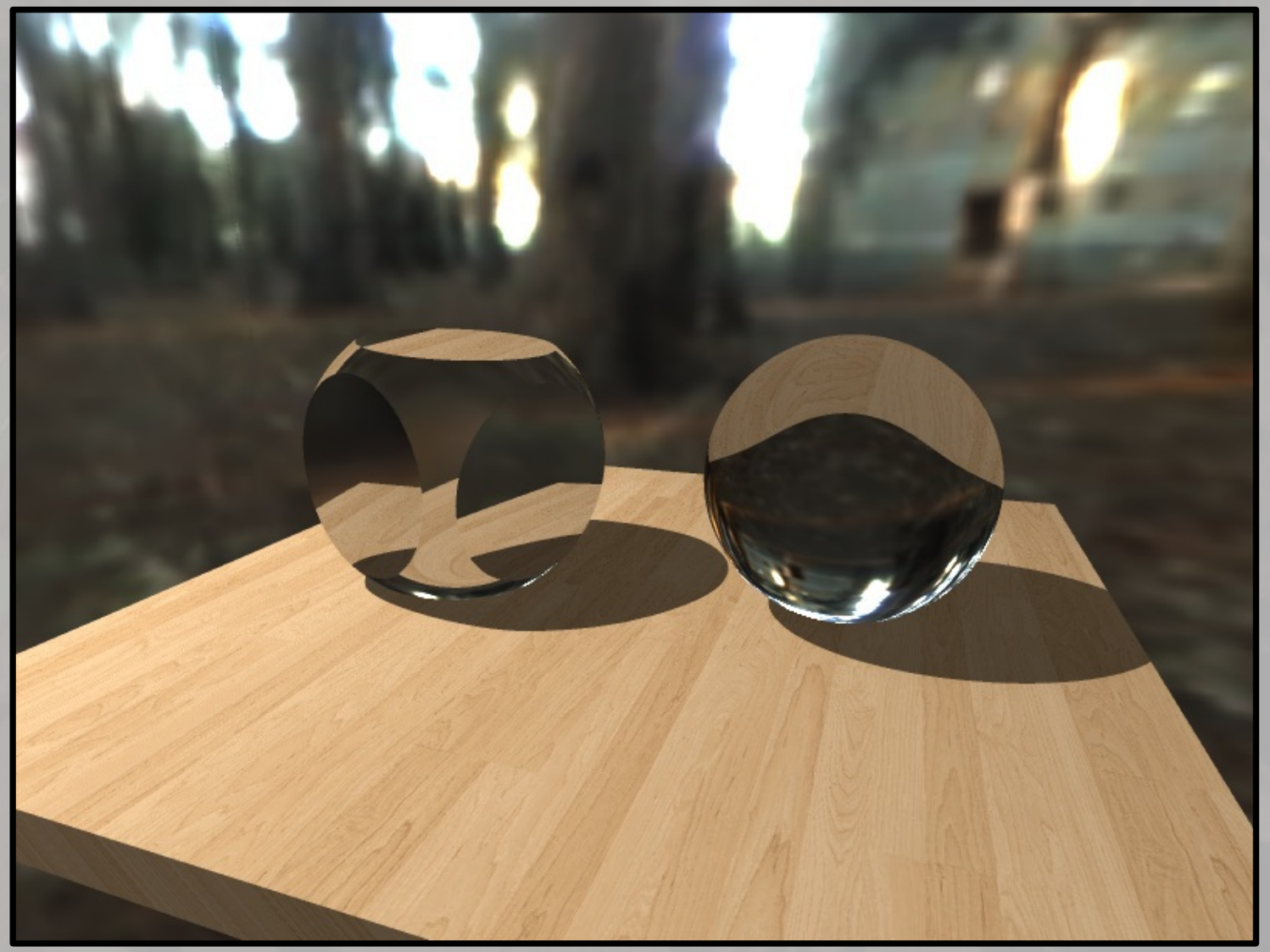
Refraction 0.2

```
Color Refraction::shade(Ray ray, const IntersectionInfo& info)
{
    extern Color raytrace(Ray ray);

    float eta = ior;
    if (info.norm * ray.dir < 0)
        eta = 1 / eta;
    Vector n = faceforward(info.norm, ray.dir);
    Vector r = refract(ray.dir, n, eta);
    if (r.length() == 0.0f) // Check for total inner reflection
        return Color(0, 0, 0);
    .....
```

Refraction 0.2 (continued)

```
    ....
    Ray newRay = ray;
    newRay.start = info.p + 1e-6 * ray.dir;
    newRay.dir = r;
    newRay.depth = ray.depth + 1;
    return raytrace(newRay);
}
```

Refraction 0.2



Комбиниране на шейдъри

- В природата, повърхностите на материалите често са изградени от няколко слоя, и всеки слой има различни отражателни свойства (и различна пропускливост)
  - Например, лакирано дърво
    - Долен слой (дърво) – дифузен, не отразява нищо, непрозрачен
    - Горен слой (лак) – леко отражателен и полупрозрачен
- Искаме да създадем шейдър, който да може да обедини няколко други шейдъра
  - И освен това го прави по някакъв интуитивен начин

Комбиниране на шейдъри

- Ще създадем Layered shader, който ще симулира ефекта в природата
  - Слоевете ще са зададени отдолу нагоре
  - Всеки слой ще задава своята пропускливост, по един от два начина
    - Статично (т.е. като число от 0 до 1)
    - Чрез текстура (така пропускливостта се мени за различните части на обекта)
    - 0 = напълно прозрачен, 1 = напълно непрозрачен

Layered Shader

- На практика, вместо с числа, пропускливостите ще задаваме с Color
  - Така пропускливостта може да е различна за различните R, G, B канали
  - Има ситуации, в които това има смисъл. Засега просто ще ползваме черно-бели изображения за текстурите и цветовете

Layered Shader

```
class Layered: public Shader {
    struct Layer {
        Shader* shader;
        Color blend;
        Texture* blendTex;
    };
    static const int MAX_LAYERS = 32;
    Layer layers[MAX_LAYERS];
    int nrLayers;
    ....
```

Layered Shader (cont.)

```
....
public:
    Layered() : nrLayers(0) {}
    void addLayer(Shader* shader, const Color& blend, Texture*
blendTex);
    Color shade(const Ray& ray, const IntersectionInfo& info);
};
```

Layered Shader (cont.)

```
void Layered::addLayer(Shader* shader, const Color& blend, Texture*
blendTex)
{
    Layer& layer = layers[nrLayers++];
    layer.shader = shader;
    layer.blend = blend;
    layer.blendTex = blendTex;
}
```

Layered Shader (cont.)

```
Color Layered::shade(const Ray& ray, const IntersectionInfo& info)
{
    Color res(0.0f, 0.0f, 0.0f);
    for (int i = 0; i < nrLayers; i++) {
        Layer& layer = layers[i];
        Color blend = layer.blendTex ? layer.blendTex->getTexColor(ray,
info) : layer.blend;
        Color color = layer.shader->computeColor(ray, info);
        res = res * (Color(1.0f, 1.0f, 1.0f) - blend) + color * blend;
    }
    return res;
}
```

Fresnel

- Ефектът на Френел (на името на френския математик Augustin-Jean Fresnel) се изразява в това, че някои материали (например водата) имат свойството едновременно да отразяват и да пречупват лъчи
  - Каква част от светлината ще се отрази, и каква ще се пречупи, зависи от ъгъла, под който лъча пада спрямо повърността
  - Уравненията на Френел изразяват именно отношението Reflected / (Reflected + Refracted)
  - Уравненията са доста сложни, но за щастие, Schlick измисля проста апроксимация, която работи добре на практика

Fresnel

```
inline float fresnel(const Vector& i, const Vector& n, float ior)
{
    // Schlick's approximation
    float f = sqr((1.0f - ior) / (1.0f + ior));
    float NdotI = (float) -dot(n, i);
    return f + (1.0f - f) * pow(1.0f - NdotI, 5.0f);
}
```

Fresnel Texture

```
Color Fresnel::getTexColor(const Ray& ray, const IntersectionInfo& info)
{
    float eta = ior;
    if (info.norm * ray.dir > 0)
        eta = 1 / eta;
    Vector n = faceforward(info.norm, ray.dir);
    float fr = fresnel(ray.dir, n, eta);
    return Color(fr, fr, fr);
}
```

Let's buld a Fresnel Shader (with layers!)

```
Layered* layered = new Layered;
layered->addLayer(floorShader, Color(1, 1, 1), NULL);
layered->addLayer(new Refraction(IOR), Color(0.3, 0.3, 0.3), NULL);
layered->addLayer(new Reflection, Color(1, 1, 1), new Fresnel(IOR));
```

Fresnel

Текстури

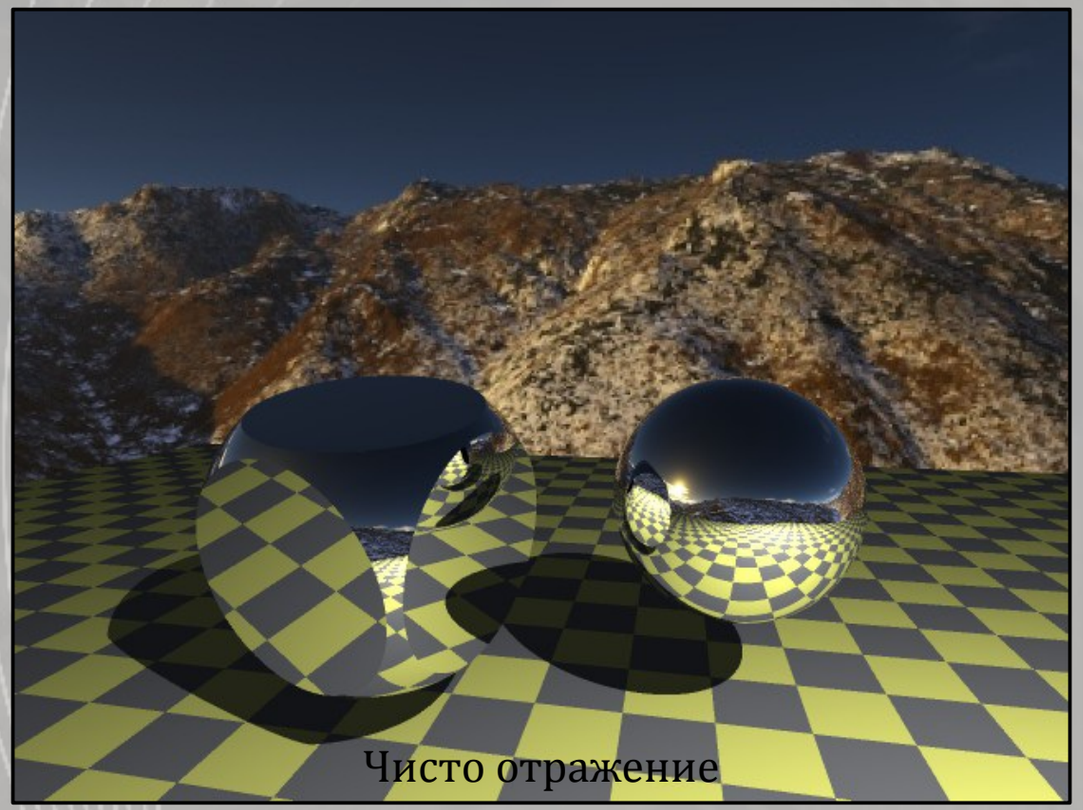
- Досега се занимавахме само с дифузни текстури, т.е. такива, които променяха цвета на обекта
- Но текстури могат да се закачат към много други свойства на шейдърите – вместо да се ползва константно количество от даденото свойство за целия обект, може това количество да се определя (взема) от текстура, и така да се мени за различните части на обекта
  - Fresnel текстурата е един пример

Примери за текстури

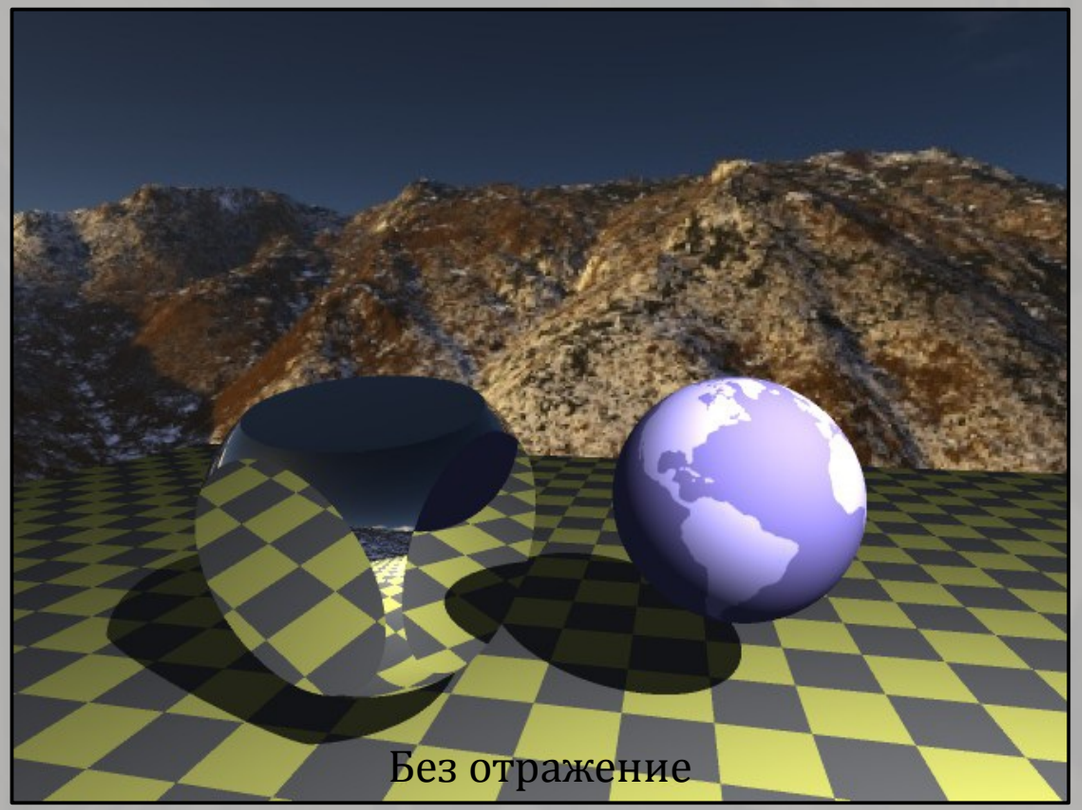
- Няколко примера:
  - Текстура, променяща цвета на отраженията (за Reflection шейдър)
  - Текстура, променяща IOR-а на материала
  - Текстура, променяща експонента при Phong
  - Текстура, променяща грапавината на материала
  - И прочее
- Ще дадем конкретен пример

Пример за текстура

- Тази текстура променя blending-а на Layered шейдър между горния (Reflection) и долния (Flat + Texture) слой

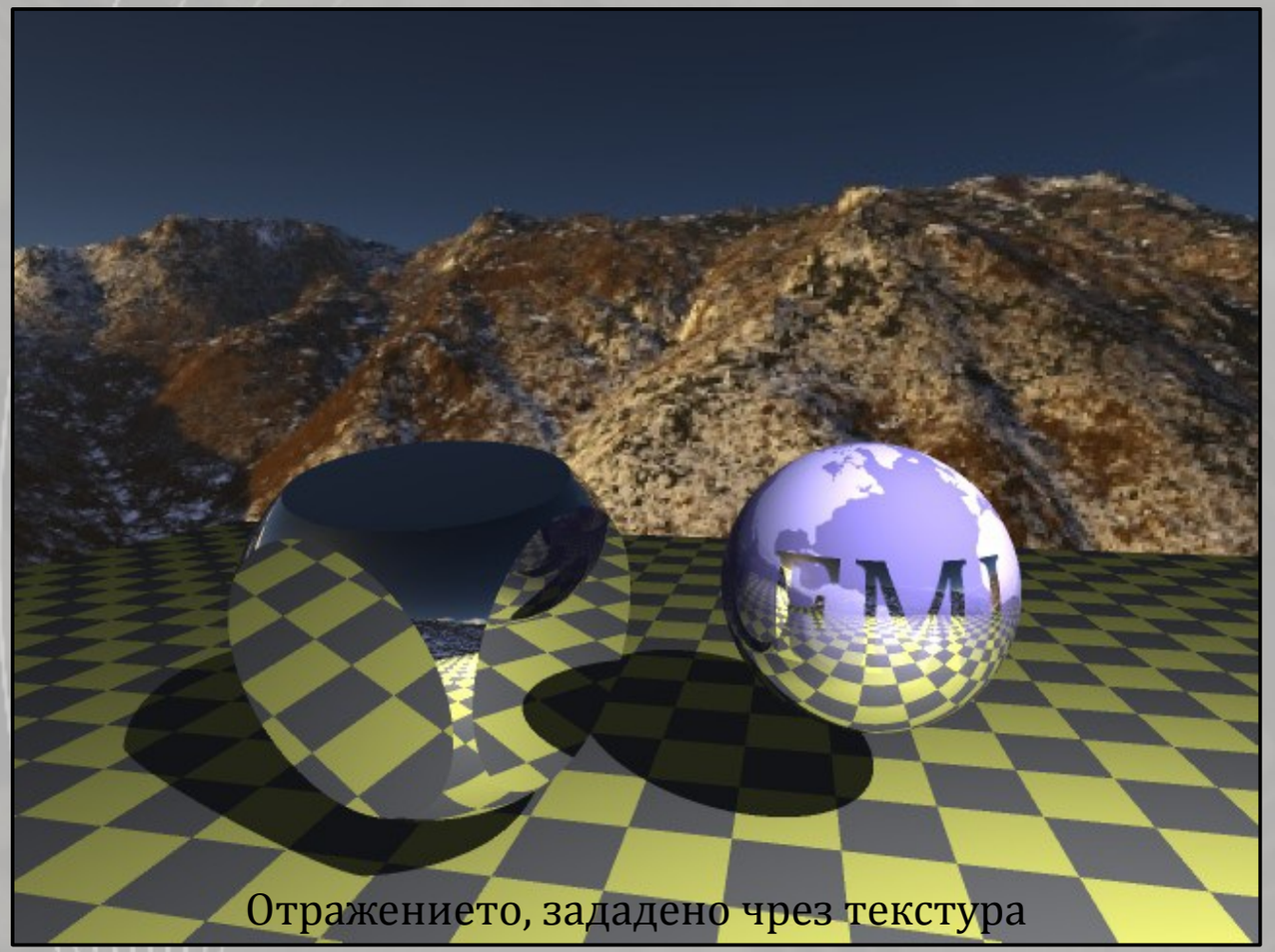


Чисто отражение



Без отражение

Пример за текстура

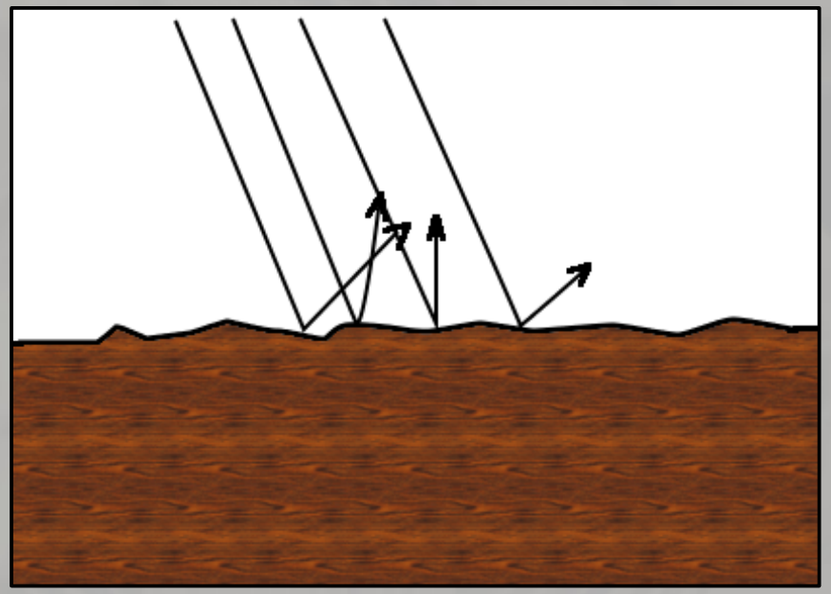


Отражението, зададено чрез текстура

Самата текстура

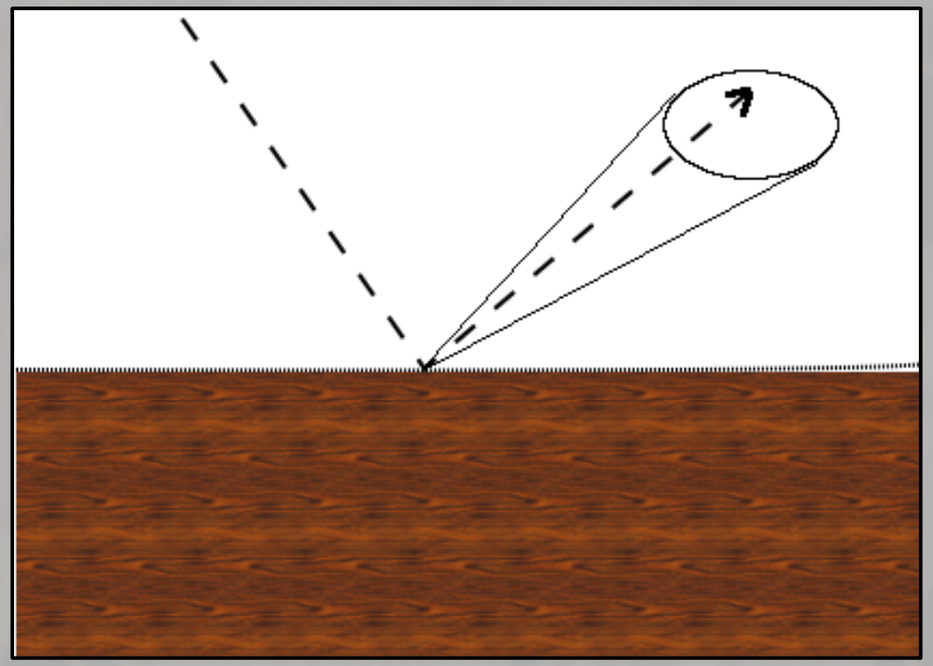
Грапави (glossy) отражения

- Грапавите отражения в природата се срещат при материали, чиито повърхности не са идеално гладки. Лъчите, които падат на тях, отскачат в произволни посоки

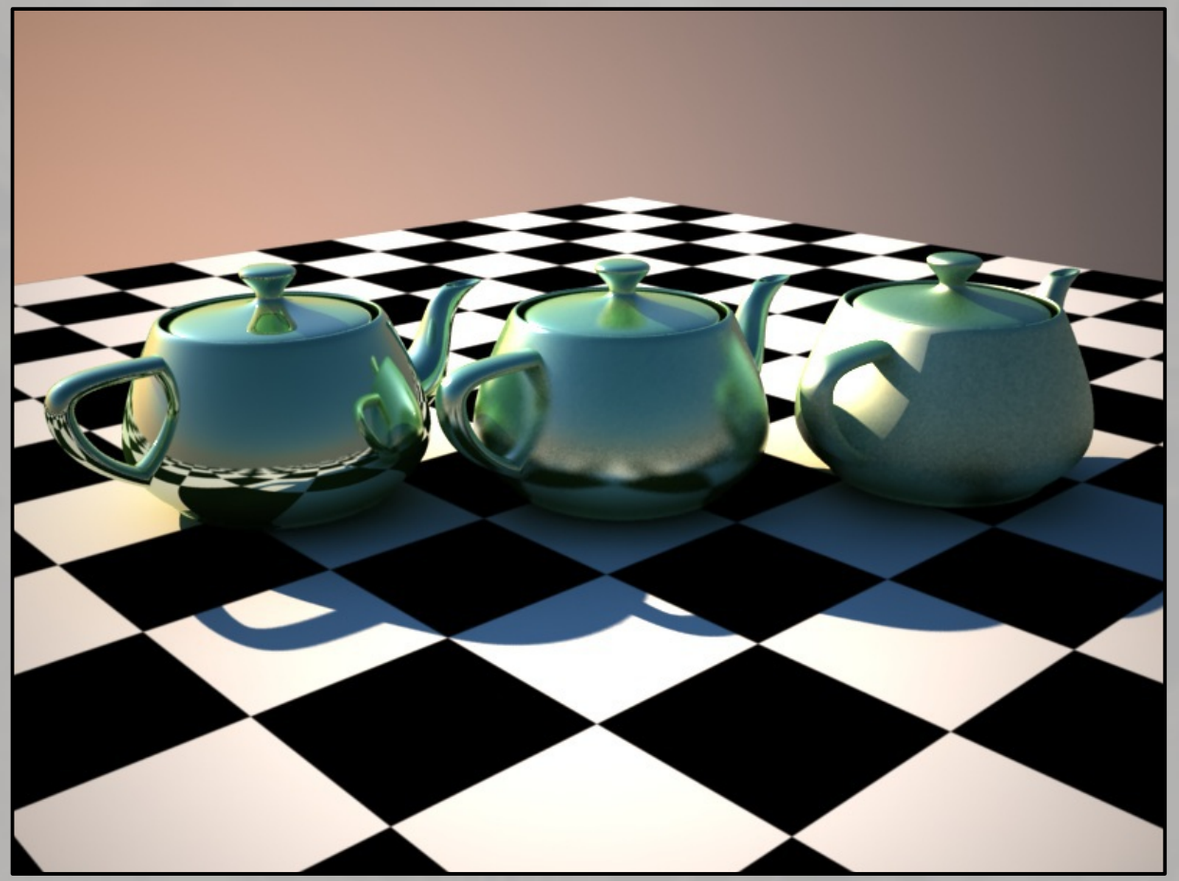


Грапави отражения

- Видимият ефект от тази особеност е, че отраженията се размиват. Светлината, идваща от една посока и попадаща в една точка, след отражението представлява конусовиден сноп от лъчи



Грапави отражения



- Колко да е разпръснат снопа зависи от glossiness параметъра. Той се движи от 0 (напълно матов материал) до 1 (чисто отражателен). Тук са показани 0.95, 0.80, 0.50

Симулиране на грапави отражения

- Ще симулираме снопа, като генерираме много лъчи, трасираме ги всичките, и усредним резултата
  - Това е бавно, но няма как иначе да постигнем същия ефект
- Лъчите в снопа ще генерираме на случаен принцип
- За да реализираме самото отклонение, ще си представим, че се отклонява нормалата на повърхността, т.е. всеки от лъчите ще смятаме както при обикновено отражение, но със случайно отклонена нормала

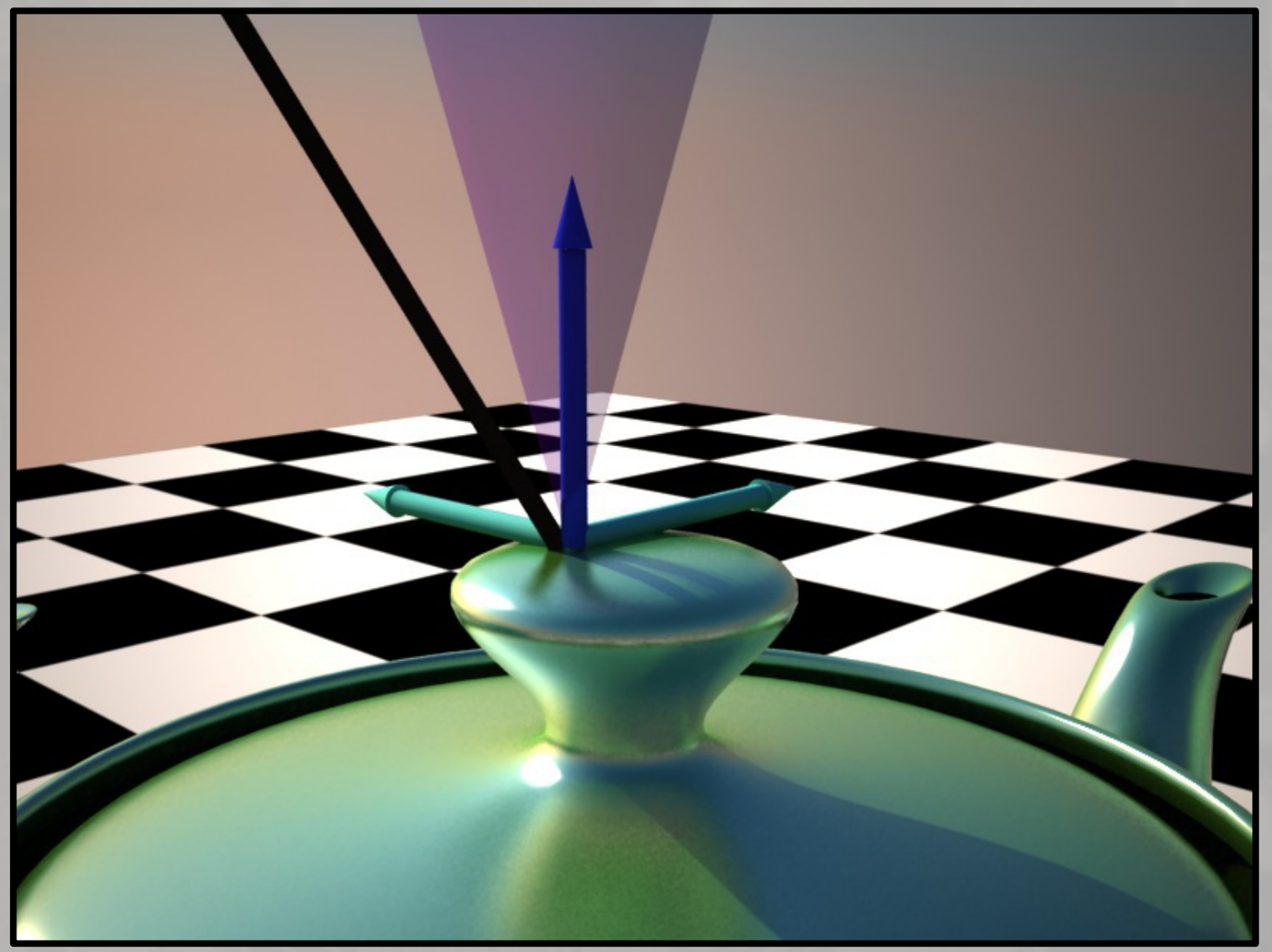
Симулиране на грапави отражения

- Алгоритъм за грапави отражения:
  1. Намираме двойка перпендикулярни на нормалата вектори
  2. Генерираме случаен 2D вектор в единичния кръг
  3. Удължаваме го в зависимост от glossiness параметъра
     - По някаква нелинейна формула, която за glossiness = 1 го нулира, а за 0 го умножава по безкрайност
     - Пример за такава формула: scaling = tan((1 – glossiness) * π/2)
  4. Използваме получените x, y за множители на векторите, които дефинирахме в стъпка 1
  5. Добавяме към нормалният вектор тези две отклонения и нормираме отново резултата – това е отклонената нормала

Симулиране на грапави отражения

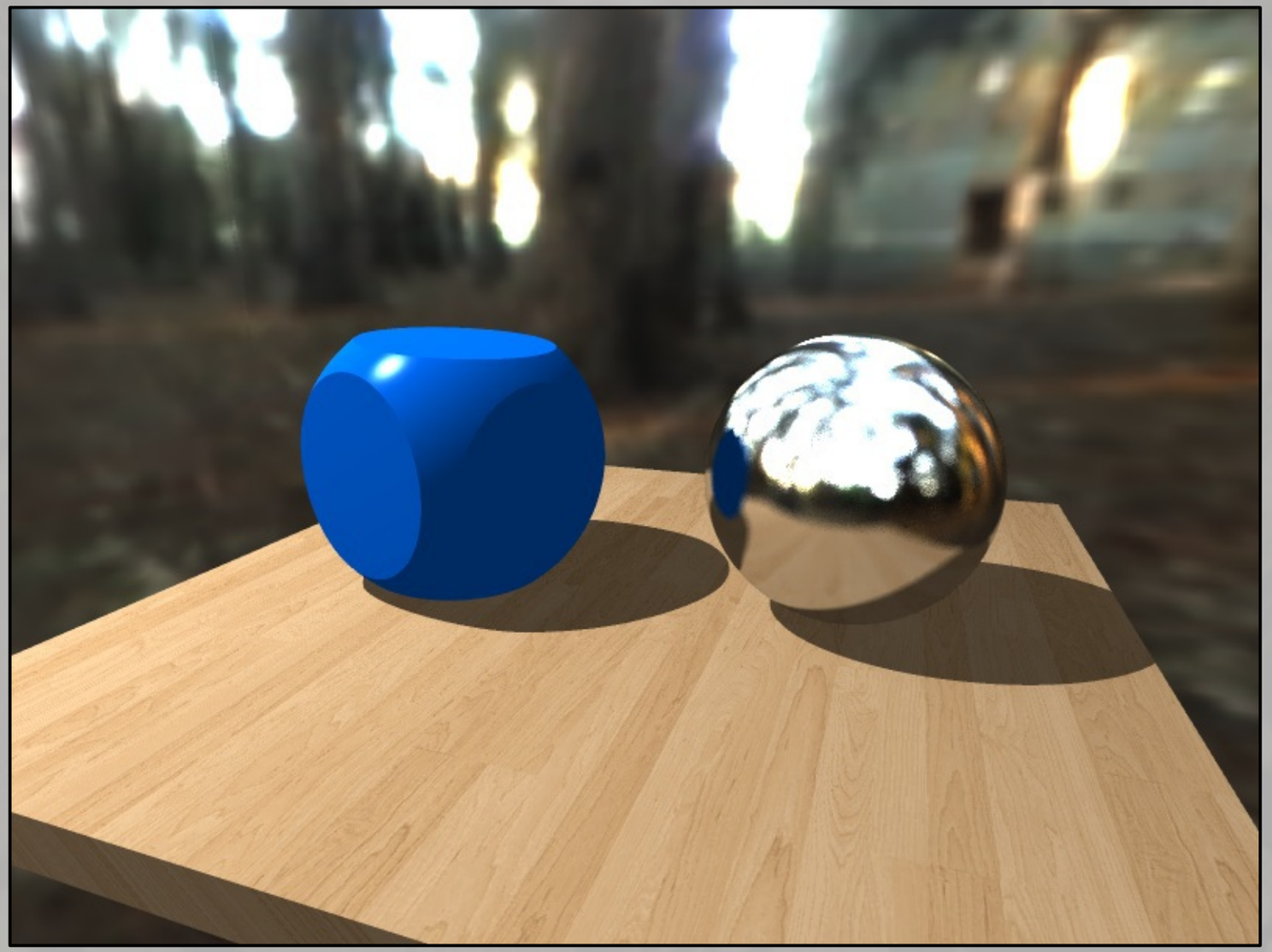
6. Трасираме обикновено отражение, използвайки модифицираната нормала
   - Тук трябва да внимаваме дали отразеният лъч е валиден – той може да задълбава под повърхността на материала!
7. Повтаряме многократно стъпки 2..6 и усредняваме резултатите им

Симулиране на грапави отражения



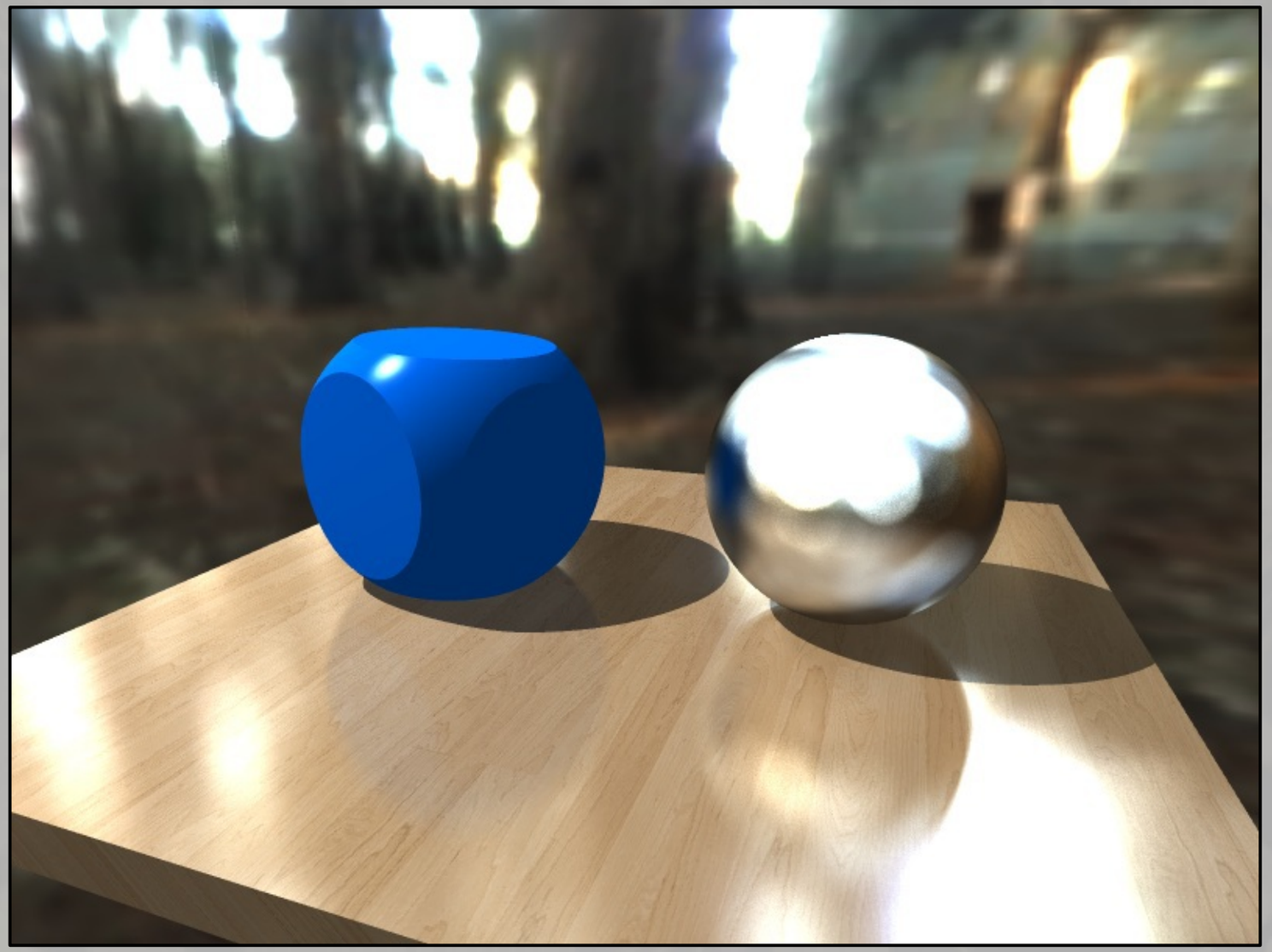
Резултати (1)

Glossiness = 0.96
Брой лъчи = 50



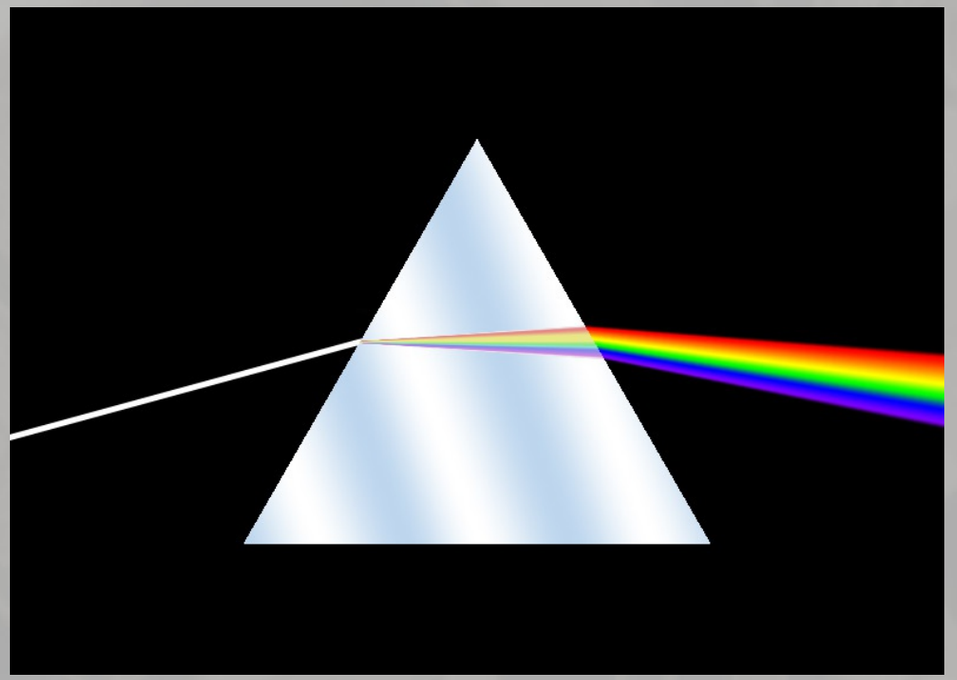
Резултати (2)

Glossiness = 0.85
Брой лъчи = 50 / 200



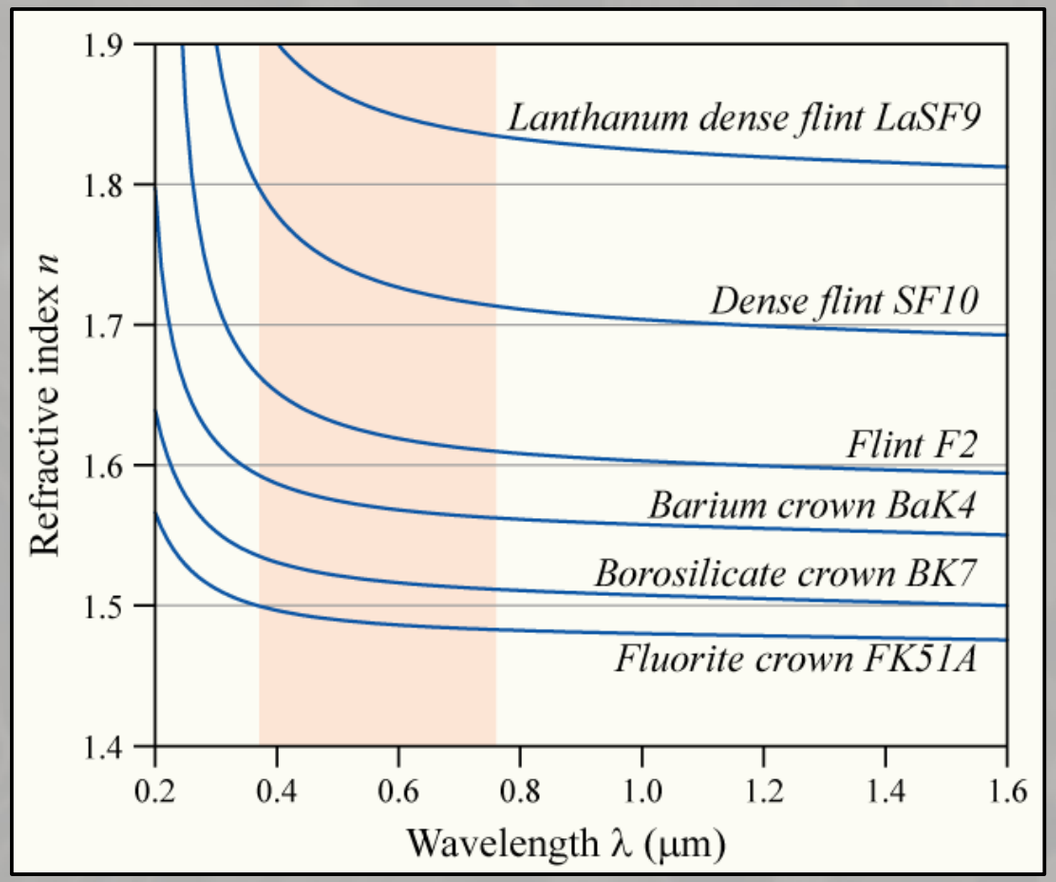
Хроматично пречупване (дисперсия)

- Индексът на пречупване като цяло зависи от дължината на вълната – по-късите вълни се пречупват повече (IOR при тях е по-голям)
- Това означава, че компонентите на бялата светлина могат да поемат по различни пътища...
  - ... и точно така става при призмата



Хроматично пречупване

- Точно каква е зависимостта между дължина на вълната и IOR е свойство на самия материал
  - Като правило, при по-“тежките“ материали, разликата е по-голяма
  - В оптиката се използва т.нар. Abbe number, който изразява тази характеристика



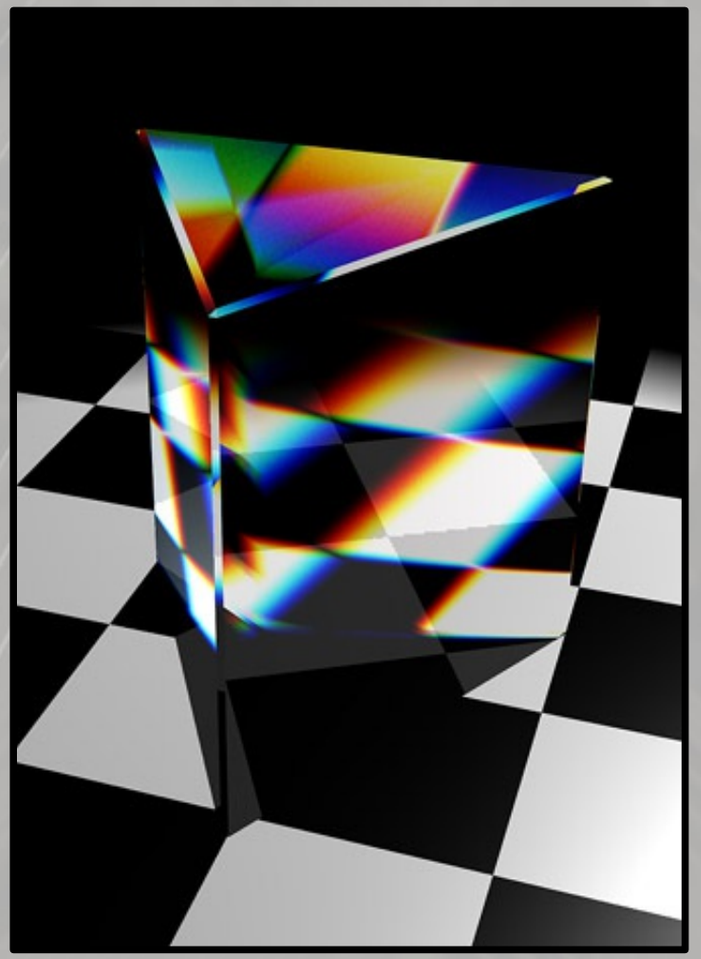
Прояви на хроматичното пречупване

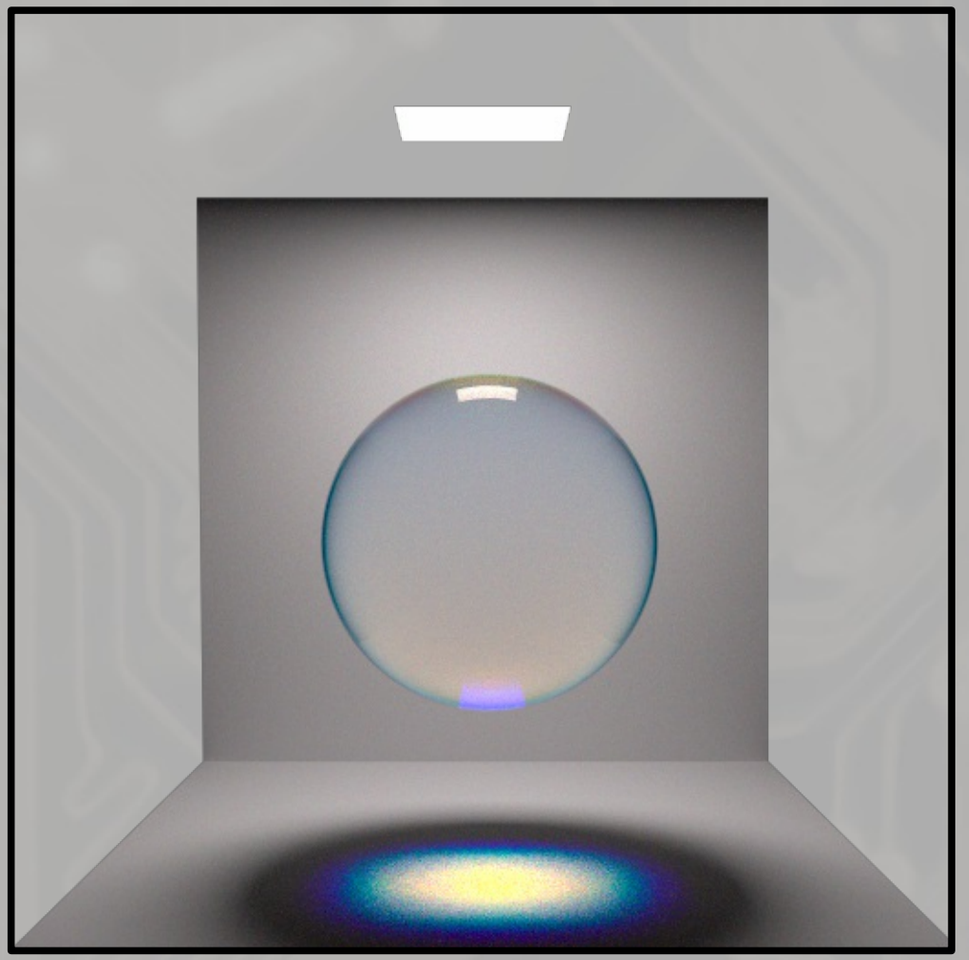
- Няколко примера
  - Дъгата
  - Разлагането на цветовете от призма
  - Цветните отблясъци на диаманти
  - Оптически дефекти във фотообективите

Симулиране на хроматично пречупване

- Както бяхме споменали в по-предните лекции, хроматичното пречупване трудно може да се симулира като ползваме само 3-компонентния модел
- Но за сметка на това може да пуснем симулации с няколко (например 8) дължини на вълната – т.е. 8 отделни рендерирания, като за всяко от тях ще сменяме с малко IOR на пречупващите материали.
- Накрая смесваме 8-те кадъра, като всеки кадър умножим по монохроматичния цвят, отговарящ на дължината на вълната, която сме симулирали за кадъра

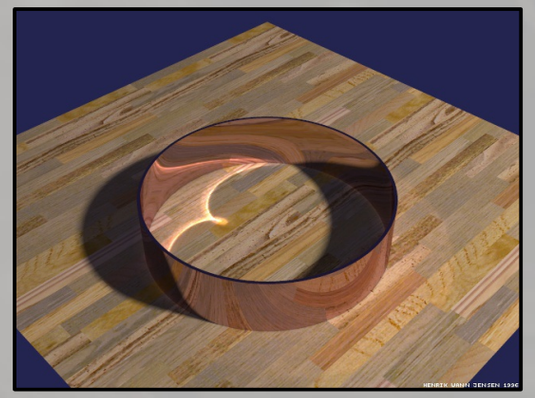
Примери за хроматично пречупване

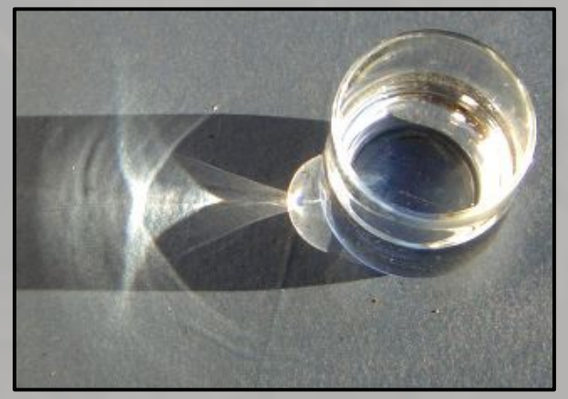




Caustics

- В компютърната графика, „caustics“ се наричат ефектите на „слънчевите зайчета“, които се получават заради концентрирането на светлинните лъчи на по-гъсти и по-редки области, при пречупване/отражение
  - Например, бялата точка фокусирана светлина зад една изпъкнала леща



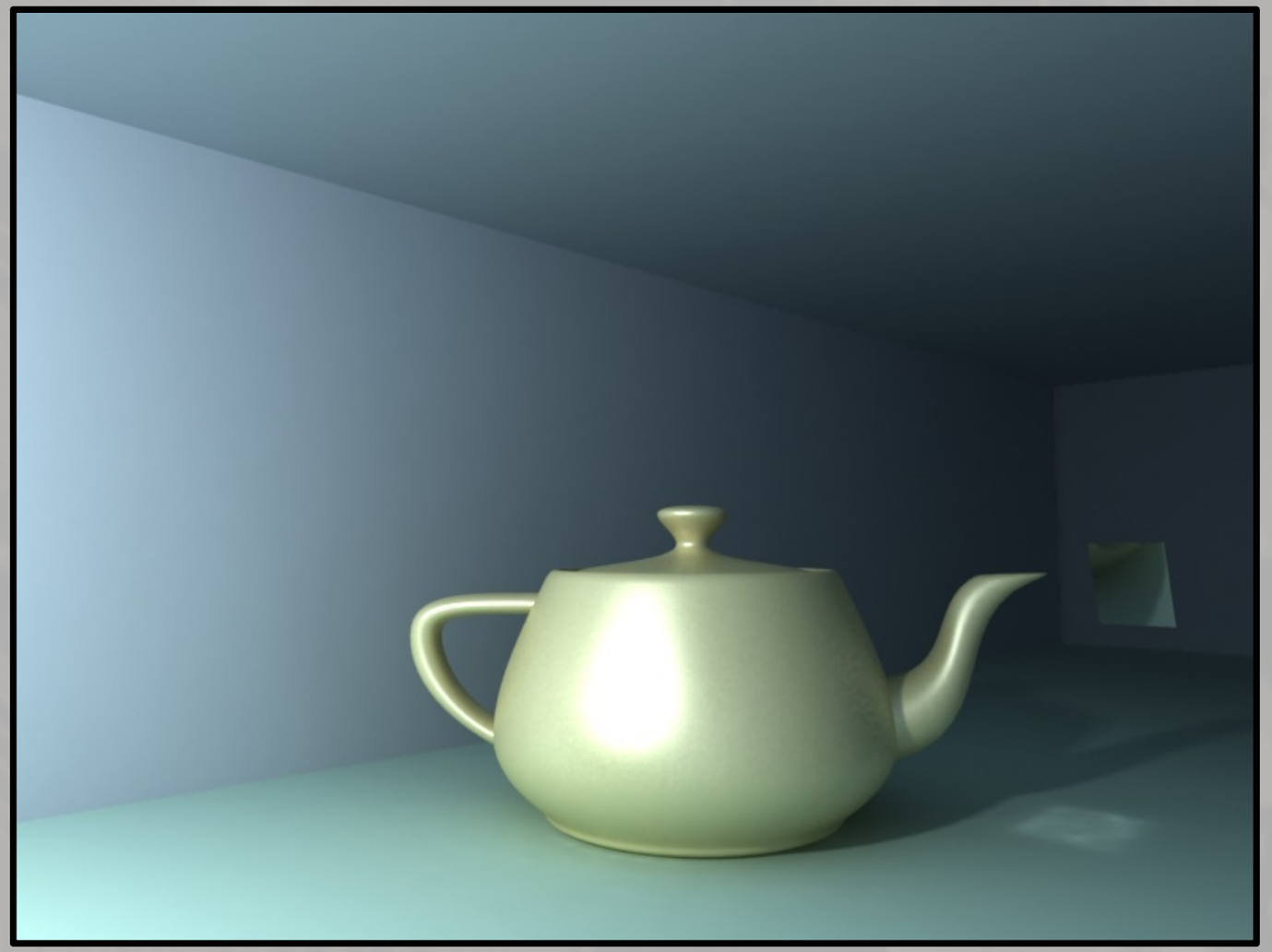


Caustics

- Caustics се дължат изцяло на факта, че осветлението е неравномерно – до някои области достигат много повече фотони от други
- Симулирането на caustics посредством досега разглежданите алгоритми за осветление е практически невъзможно
  - Осветлението не е директно, минава през пречупващ материал – не знаем в каква посока да пуснем shadow rays
  - Ако пускаме във всички посоки, ще е много неефективно – например, при сцена с малко огледало, разположено далече

Пример

- В примера, слънчевото зайче от огледалото ще се генерира много трудно, ако просто пускаме равномерно лъчи от него. Шанса да ударим огледалото е много малък



Photon Mapping

- Photon Mapping е алгоритъм, който се справя изцяло с гореописаните недостатъци, като преизчислява осветеността (концентрацията на светлината) чрез обратен raytracing
  - Т.е., лъчите тръгват от лампите
  - Изстрелват се голям брой частици (фотони) във всички посоки от лампата, и се следи къде попадат (след едно или повече отражения и пречупвания)
  - Всичките фотони се записват в някаква структура, позволяваща бързо претърсване (намиране на K-те най-близки съседа на произволна 3D точка)

Photon Mapping

- След тази преизчисляваща фаза, работим по схемата на класическия raytracing, но добавяме и caustics частта при изчисляване на осветлението – намираме K-те най-близки съседа до пресечната точка в photon map-а, и, спрямо разстоянията до тях, изчисляваме колко осветление идва от caustic-ите.
- При photon mapping може да се симулира и хроматично пречупване при caustic-ите – трябва да се пускат повече caustic-и, и всеки да носи информация само за определена дължина на вълната

Пример за photon map

- Пример за фотоните в един photon map. Дясната сфера е стъклена