3D графика и трасиране на лъчи v3.0

http://raytracing-bg.net/

Тема 1

Увод

Съдържание

- Какво ще представлява курсът?
- Понятието „rendering“
- Рендериращи алгоритми
  - Z-buffer
  - Ray-tracing
  - Scanline
  - Сравнение

Какво представлява курсът?

- Курсът е въвеждащ в областта на 3D графиката и алгоритъма на трасиращият лъч (raytracing)
- На семинарите ще разказваме по презентациите...
- ... и ще показваме/пишем код (C++)
  - Кодът по време на семинарите ще показва ядрото на обсъждания алгоритъм
  - Несъществените части ще правим извън лекциите
  - Всичко ще е достъпно през *github*

Какво представлява курсът?

- Ще се концентрираме основно върху алгоритъма raytracing
  - Z-buffer и Scanline ще обясним накратко в тази лекция, но няма да навлизаме в много детайли
- Пълен тематичен план има на сайта на курса: http://raytracing-bg.net/
- На сайта също така ще качваме всички презентации, клипове, новини и прочее – т. е., всички материали по курса

Изисквания

- Средно ниво умеене на C++
- Аналитична геометрия, линейна алгебра
- Операционни системи и среди
  - Можете да ползвате Windows, Linux или Mac OS X, ще поддържаме и трите

Оценяване

- Два теста по 15 въпроса (общо 30 т. max)
- Курсов проект (max 40 т)
- Домашни работи (max 30 т)
  - Тестовете и проектът са задължителни, домашните са по желание
- *Оценка* = floor((*точки* – 5)/10)
  - 35-44 = „среден“; 45-54 = „добър“; 55-64 = „много добър“
  - 65+ = „отличен“

Новини

- Това е третото издание на курса
  - git
  - C++11
  - GPU
- CG^2 2013
  - 26.10.2013г за програмистите
  - Challenge

Какво **не** е курсът?

- Курсът е с некомерсиална цел
- Не е всеобхватен
  - Няма да разглеждаме графичните API-та като OpenGL или DirectX...
  - Няма да изучаваме програмите за моделиране и рендеринг като 3ds Max, Maya, Blender...
  - Курсът като цяло е с въвеждаща роля
- Отношението *кредити/трудност* не е високо :)

Мотивация

- Съвременната компютърна графика е на много високо ниво
  - Все по-голям реализъм в игрите
  - Когато има достатъчно процесорно време, може да се постигне висока доза фотореализъм
    - Първият изцяло CGI филм е Toy Story (1995), а Final Fantasy: The Spirits Within (2001) е първият фотореалистичен
    - Удивителни компютърно-генерирани части има и в The Matrix: Reloaded, Avatar, Oblivion, ...
  - CGI части за намаляване на бюджета на филмите

Мотивация

- Човек трудно вече може да различи истинското от изкуствено-генерираното
  - Autodesk challenge: http://area.autodesk.com/fakeorfoto
- В днешно време компютрите са достатъчно мощни за много сложни и убедителни ефекти
  - Ограничението обикновено се оказва само въображението и уменията на дизайнерите
- Въпреки бурния прогрес на алгоритмите за рендериране, техните основи не са се променили особено през последните 20 години

Какво означава понятието „rendering“?

- В контекста на 3D графиката, rendering е процесът на интерпретация на някаква информация от абстрактен към по-конкретен вид
  - Основната идея е, че можем да настройваме изходния конкретен вид според нашите желания
- Ще поясним с няколко примера

Пример: Web browser

- Например: даден уеб браузър (като Firefox) чете HTML код и генерира конкретното съдържание (във вид на bitmap), което се показва на екрана:

```
<html>
  <head>
    <title>This is an apple test</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  </head>
  <body>
    <p>This is a test page</p>
    <hr>
    <center>
      <h1>This is an apple</h1>
      <img src="apple.jpg" alt="apple">
      <p><font size="1">The apple of sin</font></p>
    </center>
    <hr>
    <p><b>Adios!</b><br>
       Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum sed erat id urna
lobortis lacinia. Nulla malesuada fermentum accumsan. Suspendisse in velit sit amet
erat euismod accumsan vel non arcu. Pellentesque lacus est, eleifend vitae posuere
et, semper pharetra urna.
    </p>
    <font color="#ff2200"><h2>That's all, folks!</h2></font>
  </body>
</html>
```

This is a test page

This is an apple

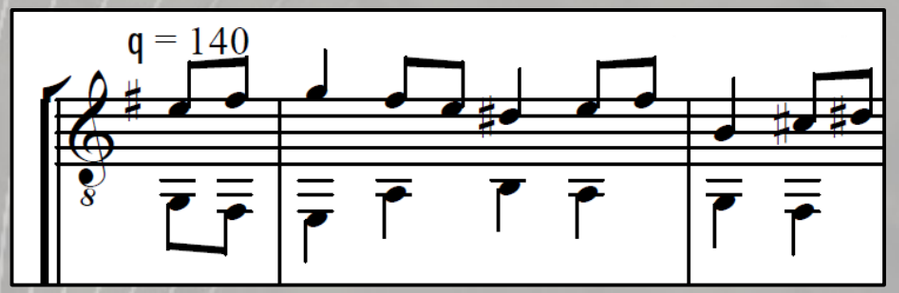
The apple of sin

Adios!
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum sed erat id urna lobortis lacinia. Nulla malesuada fermentum accumsan. Suspendisse in velit sit amet erat euismod accumsan vel non arcu. Pellentesque lacus est, eleifend vitae posuere et, semper pharetra urna.

That's all, folks!

Пример: класически музикант

- Музикантът превръща нотите в звукови вълни:

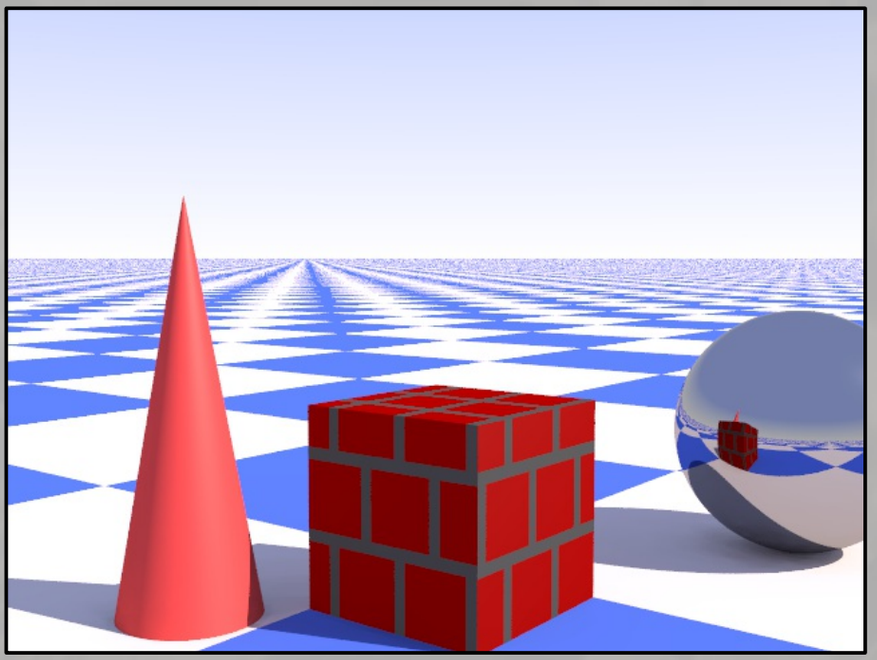


Т.е., той рендерира музиката

Пример: рейтрейсър (POV-Ray)

- POV-Ray прочита текстово описание на колекция от стереометрични примитиви и ги рендерира във вид на растерно изображение:

```
87
88  camera {                  // --- Camera 1 ---
89    location    <4.5, 1.7, 3.5>
90    right       (4/3)*x
91    look_at     <0, 1.3, 0>
92    angle       45
93  }
94  light_source {            // --- Light 1 ---
95    <-5000, 14000, 15000>
96    color rgb <1.0, 0.9, 0.78>*2.3
97  }
98
99  plane {
100   y, 0
101   texture { Base }
102 }
103 box {
104   <1,1,1>, <0,0,0>
105   texture { Bricks }
106 }
107 sphere {
108   <-1.5, 0.7, 1.5>, 0.7
109   texture { Mirror }
110 }
111 cone {
112   <1.4,0,-0.4>, 0.35, <1.4, 2, -0.4>, 0.0
113   texture { RedPaint }
114 }
```



Дизайн

- Дизайнът на 3D моделите, които се ползват в компютърната графика, обикновено се прави със специализиран софтуер (3ds Max, Maya, XSI, Blender, AutoCAD, SolidWorks, ...)
  - Няма да застъпваме тази част много в курса
- Пример за дизайн [spliner]

Алгоритми за рендериране

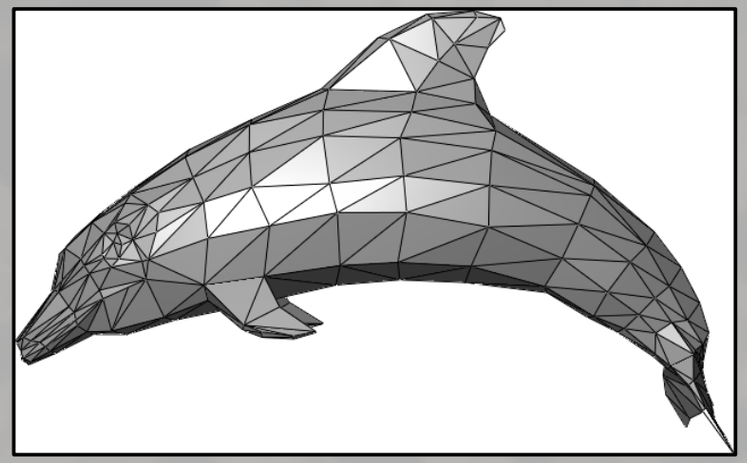
- Подобно на различните web browser-и, има различни алгоритми за рендериране, предлагащи различен компромис от качество на картината, скорост, възможности...
- Но най-основните подходи за рендериране са само три:
  - Z-buffer (или растеризация)
  - Raytracing
  - Scanline rendering

Z-buffer

- Z-buffer (растеризация) е много широко-използван алгоритъм, основа на графичните API-та OpenGL и Direct3D
  - Традиционно най-бърз алгоритъм, ползва се изключително в игрите
  - Позволява да се ускори с порядъци чрез специализиран хардуер (GPU-тата във видеокартите)

Z-buffer

- Основната идея е да представим сложните тримерни обекти чрез съвкупност от малки примитиви, например триъгълници
- Всичко може да се сведе до триъгълници
  - Изкуствено моделирани 3D обекти
  - Гладките повърхности се апроксимират с многостени
  - 3D скенерите извеждат триъгълни мрежи

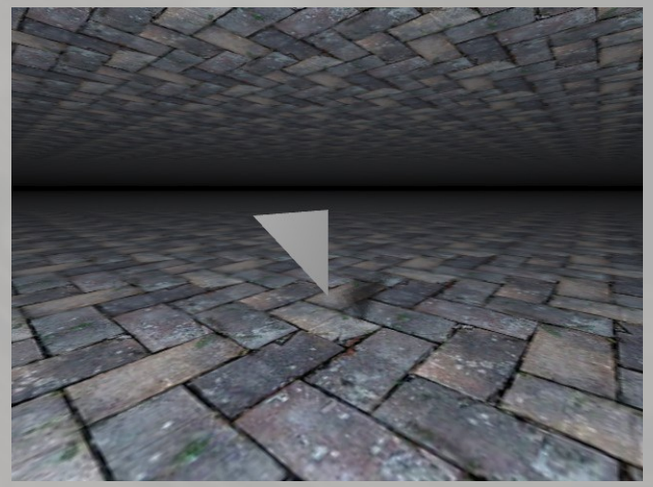
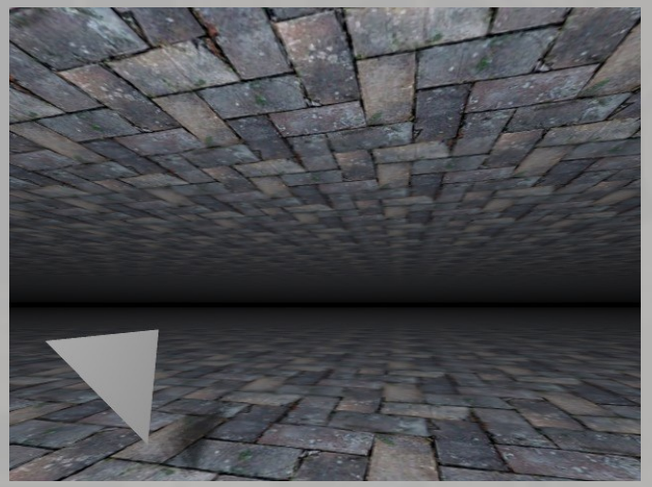
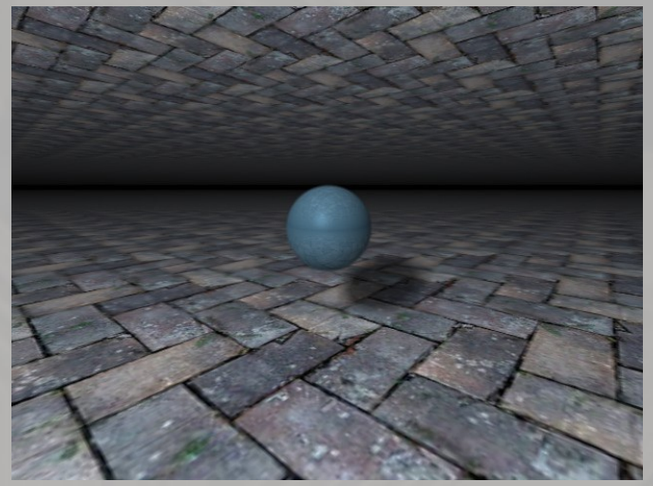
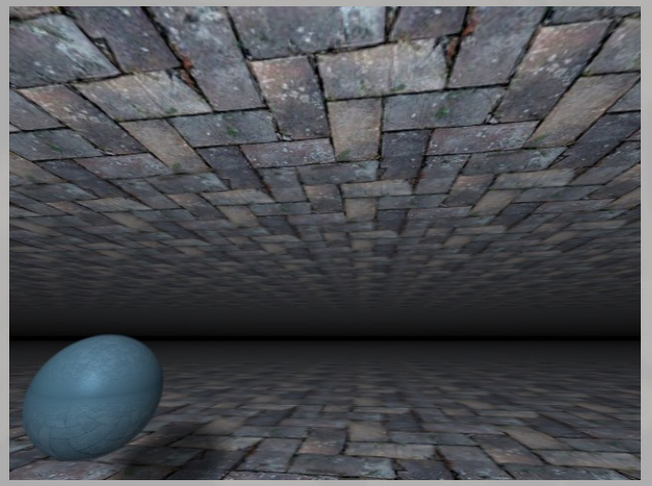


Триъгълни мрежи

- Реално, обикновено се ползват полигонални мрежи, но разликата е несъществена
- Триъгълникът има удобното свойство да запазва формата си при проекция в 2D, т.е. 3D триъгълникът, след трансформация 3D->2D, остава (2D) триъгълник
- При повечето други примитиви, това не е така; ще дадем пример

Пример: проекция 3D към 2D

- Триъгълник остава триъгълник в 2D, независимо къде се намира на екрана
- Сферата се изобразява в кръг, ако я гледаме централно, но иначе заема друга форма

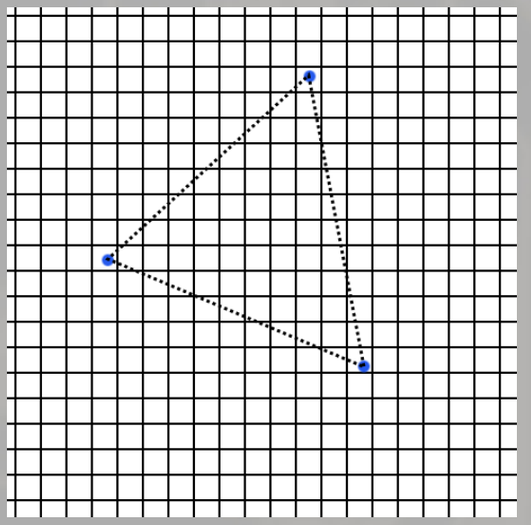





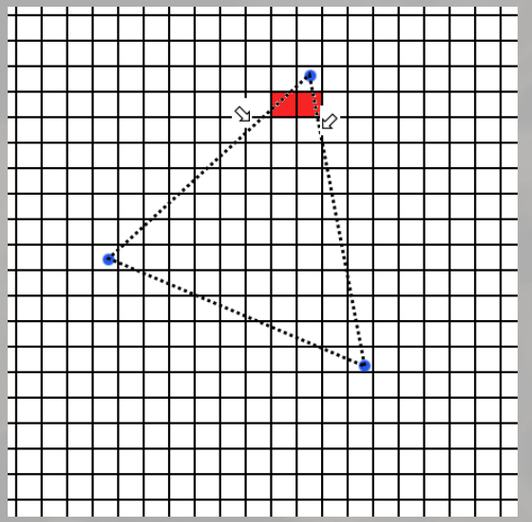
Растеризация на триъгълник

- Тримерните координати на върховете на триъгълника се свеждат до двумерни екранни координати чрез подходяща перспективна трансформация
  - В OpenGL и Direct3D реално се ползват хомогенни (4D) координати, но това не е задължително
- Изрязват се частите от триъгълника, които няма да се виждат (напр., излизат извън екрана)
- Триъгълникът се запълва с необходимия цвят
  - Това е лека 2D операция

Растеризация на триъгълник

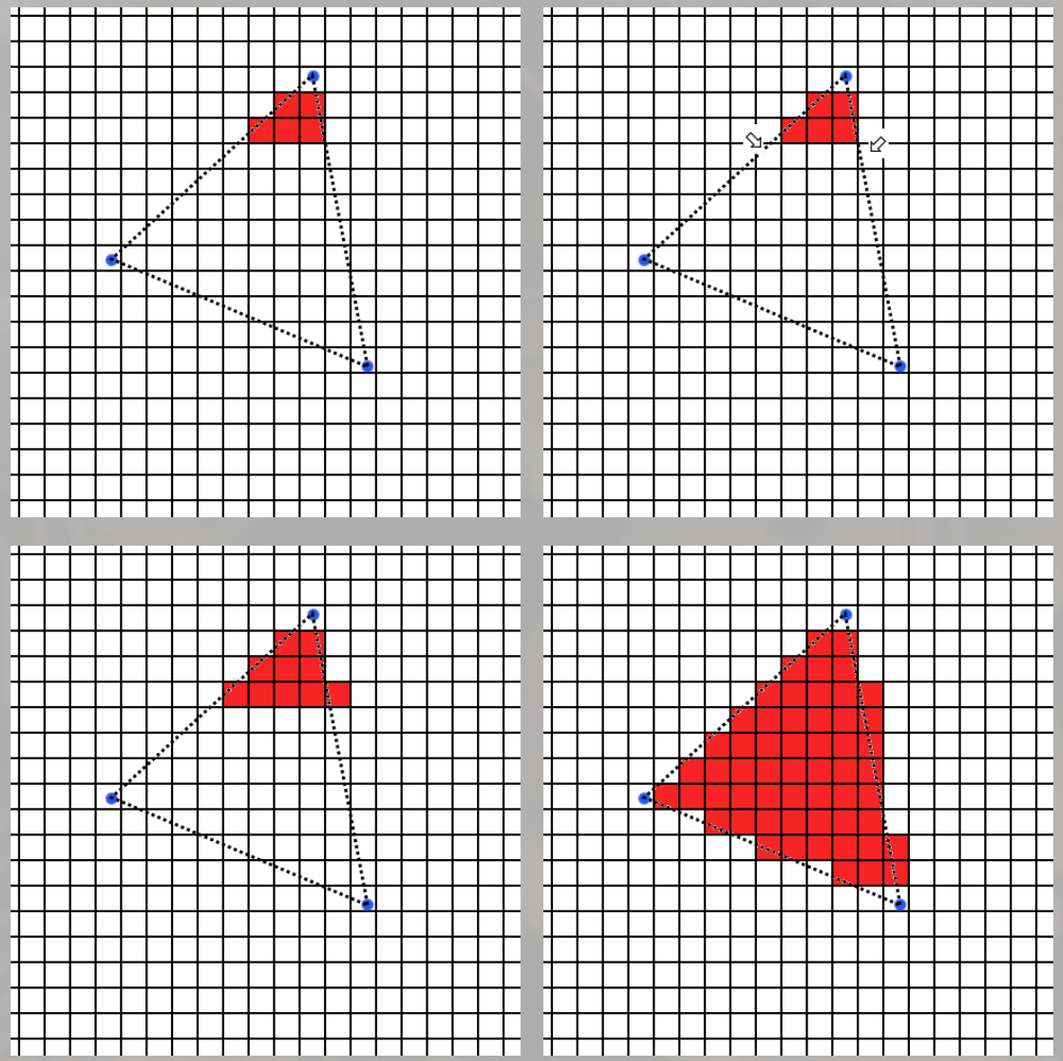
- Върховете се сортират по Y – горен, среден и долен
- Обхождат се редовете между горния и средния
- За всеки ред се намират най-левият и най-десният пиксел, който трябва да бъдат оцветени
  - Чрез линейна интерполация между горния и средния връх, и между горния и долния връх





Растеризация на триъгълник

- Редът се запълва с избрания цвят, след което се минава на следващия
- Аналогично за редовете между средния и долния връх
- Тези операции могат да се изпълняват изключително бързо на GPU-то

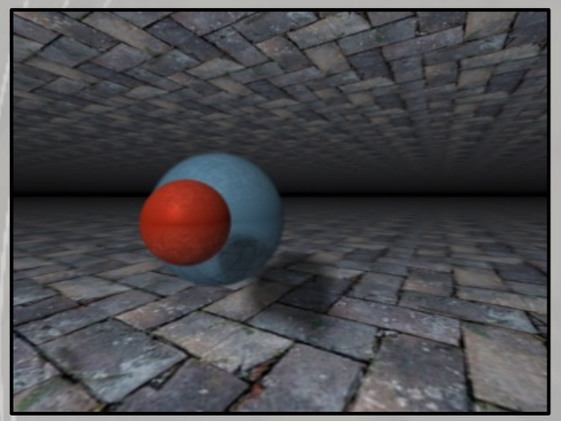


Растеризация на триъгълник

- Вместо да е еднакъв за всички пиксели, цветът обикновено се интерполира между върховете, или се взема от текстура; могат да се пишат и подпрограми (fragment shaders), които определят цвета на отделен пиксел
  - Нещата малко се усложняват: например, текстурните координати не могат да се интерполират линейно, а е необходима перспективна корекция
  - Но въпреки това, алгоритъмът остава много бърз
- Но трябва да решим един проблем: в какъв ред да чертаем триъгълниците

Ред на изчертаване при растеризацията

- По-предните обекти трябва да се изчертават последни
  - Това е т.нар. „Алгоритъм на художника“ („Painter's algorithm“)
  - Обикновено сортирането на ниво обекти не е достатъчно – трябва да се сортират отделните триъгълници
  - Редът може да се променя през отделните кадри [order.m4v]
    - Тоест, имаме по едно сортиране на всеки кадър

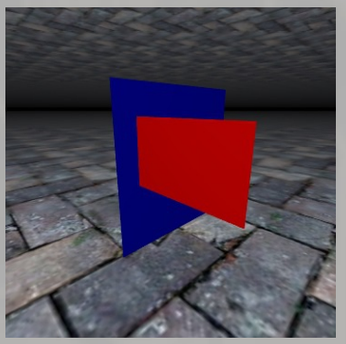


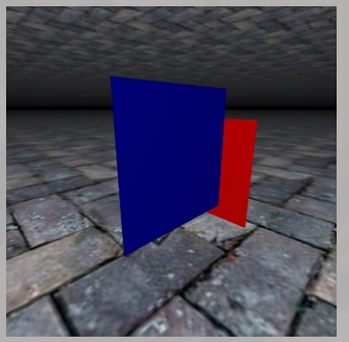
Проблеми при алгоритъма на художника

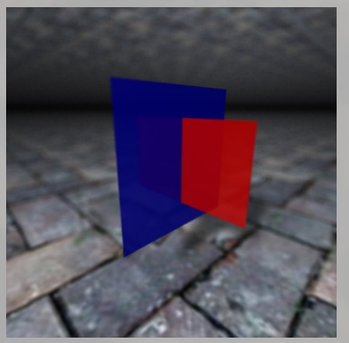
- Overdraw: когато имаме много обекти в сцената, даден пиксел ще бъде презаписван много пъти, но ще се вижда само последният запис
- За да се избегне това, се ползват алгоритми, които разделят обектите в отделни, по-големи логически групи (например, отделните стаи в една сграда) и елиминират предварително невидимите групи
- Алгоритъмът на художника не се справя с някои конструкции поради фундаментален проблем

Проблеми при алгоритъма на художника

- Ако два триъгълника се пресичат, то никоя подредба на изчертаване не е вярна
- Това може да се реши, като се цепят триъгълниците по пресечните ръбове
  - Но става доста усложнено, и не скалира добре при много триъгълници
  - Освен това, проблемът с overdraw остава





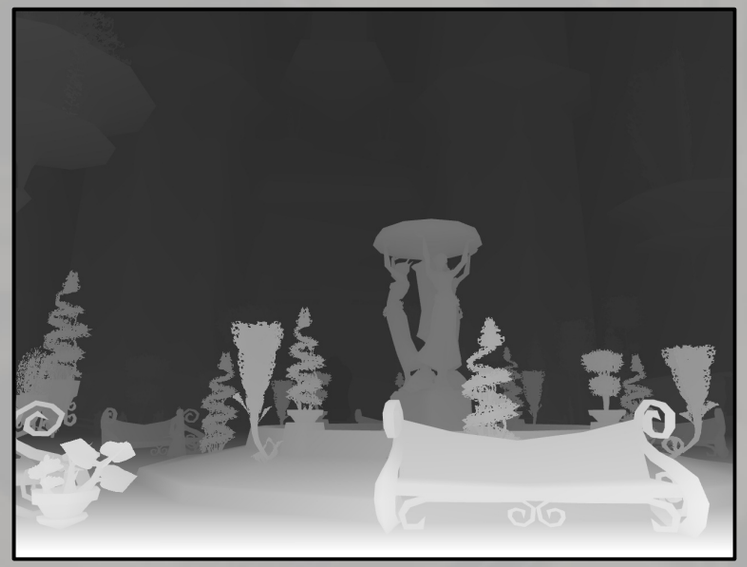


Z-buffer

- Решението на тези проблеми е техниката „Z-buffer“. При нея се поддържа отделна картина, всеки пиксел от която пази дълбочината (разстоянието от камерата) до най-близкия обект, заемащ този пиксел до момента
  - Т.е., близките обекти попълват буфера с ниски стойности, далечните – с високи
  - При изчертаване на триъгълник, за всеки пиксел проверяваме в Z-buffer-а, дали не сме прекалено далеч (и следователно – скрити зад някой друг);

Z-buffer

- Z-buffer-ът решава няколко проблема:
  - Редът на изчертаване вече не е важен
  - Overdraw-ът може да бъде намален, като чертаем обектите в ред, обратен на художническия: голямата част от пикселите ще се отхвърлят заради Z-стойност
  - Лесно се проверява дали даден обект е видим.
  - Като цяло, нещата доста се опростяват



Недостатъци на Z-buffer

- Много ефекти няма как да бъдат симулирани
  - Отражения, пречупвания, полу-прозрачност...
  - ...“слънчеви зайчета“ (caustics), глобално осветление...
  - Има алгоритми за сенки, но са сложни (особено за „меки“ сенки)
  - Дълбочина на полето (DOF)
- При наистина много полигони, дори и най-бързите видеокарти се задъхват
  - ~1600 милиона триъгълника в секунда (GTX680, 2012г.)

Недостатъци на Z-buffer

- От гледна точка на програмиста:
  - Огромно разнообразие от потребителски хардуер
    - С различна поддръжка на стандарти, разширения,...
  - Скалируемост

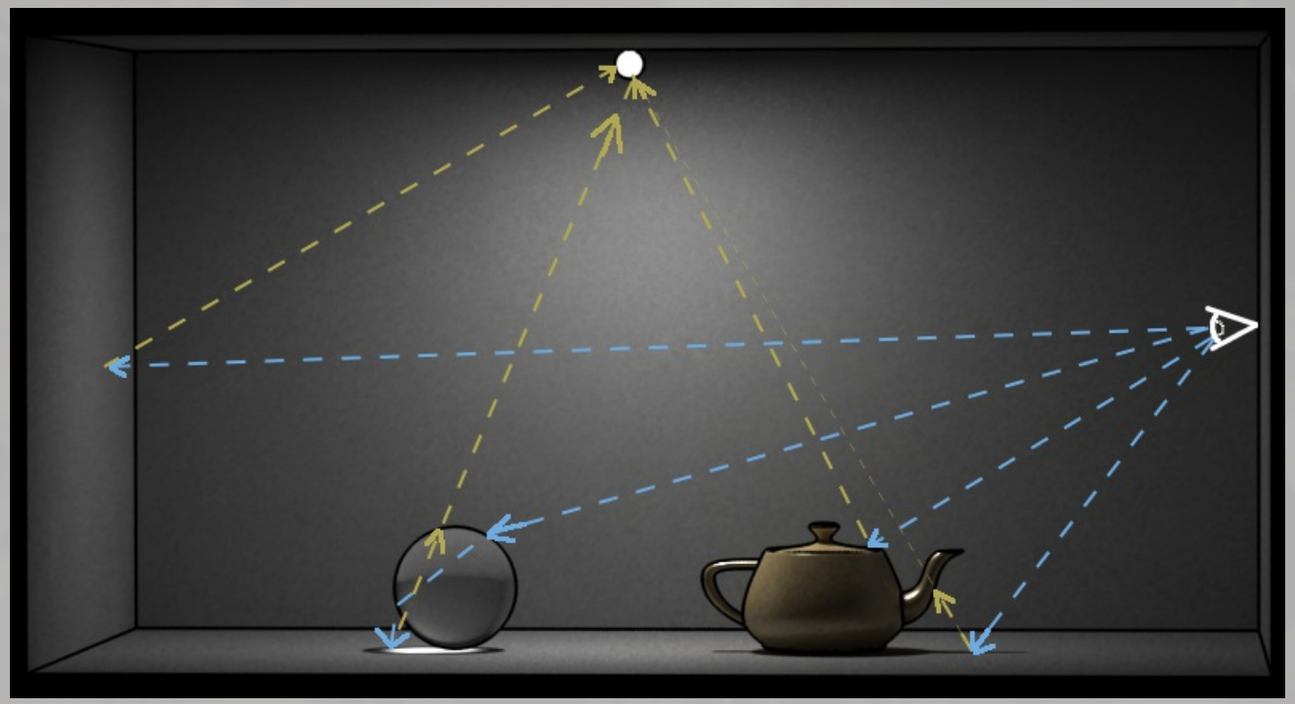
Сложност на Z-buffer алгоритъма

- При тежките сцени (с много триъгълници, често с размери близки до единичен пиксел), броят на триъгълниците доминира сложността
  - В този случай, тя е O(N), където N е броят на триъгълниците
- Разрешаващата способност също оказва влияние, макар и не толкова силно

Raytracing

- Методът на трасиращия лъч (raytracing) – основната идея е, че практически всички срещани 3D обекти могат да бъдат пресечени с тримерен лъч (и да бъде определена пресечната точка)
- Лъчите в raytracing-а се държат като фотоните в природата – движат се по права линия, отразяват се от повърхности, пречупват се през различните материи...
- ... само че тръгват от „окото“ и вървят по обратния път до светлинния източник

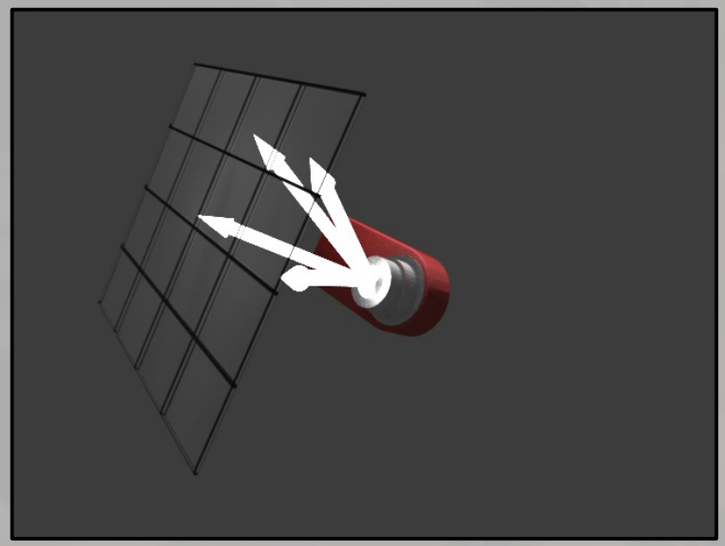
Raytracing



- За всяка повърхност, в която се ударят лъчите, се проверява има ли „път“ до лампата – ако няма, значи сме в сянка

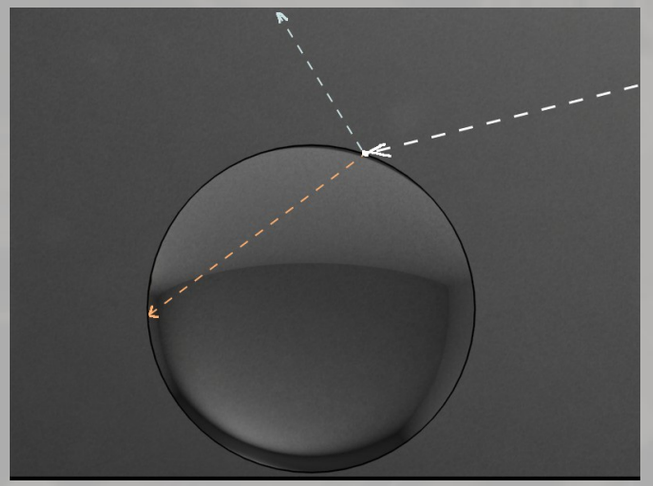
Raytracing

- Резултатното изображение получаваме, като за всеки пиксел от него пуснем (поне) един лъч
- За всеки лъч, проследяваме в каква повърхност се удря първо, и след което в зависимост от повърхността и осветеността, определяме цвета на пиксела
- Ако усредним повече от един лъч през пиксел, имаме Antialiasing



Raytracing

- При сложни материали (напр. стъкло), рейтрейсингът ни позволява да разцепим лъча на две части – отразен и пречупен (както става и в природата). Това увеличава експоненциално броя лъчи (комбинаторен взрив), но обуславя висок реализъм



- Можем да нагласяме колко отскока на лъча има нужда да трасираме, но и малък брой (напр., 8) е достатъчен в повечето случаи.

Raytracing

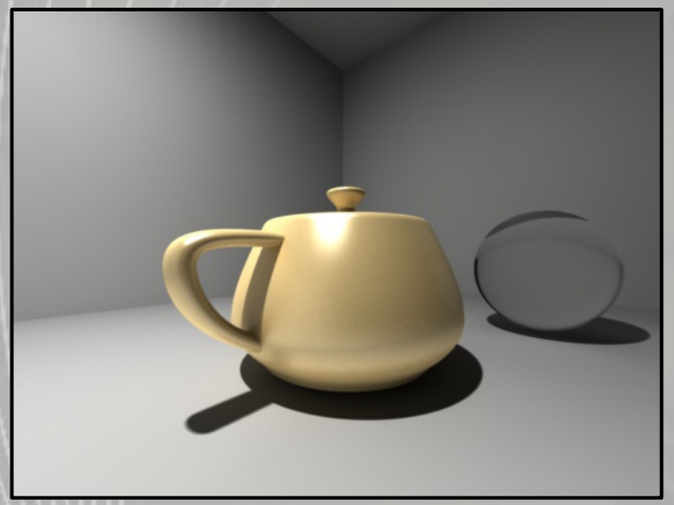
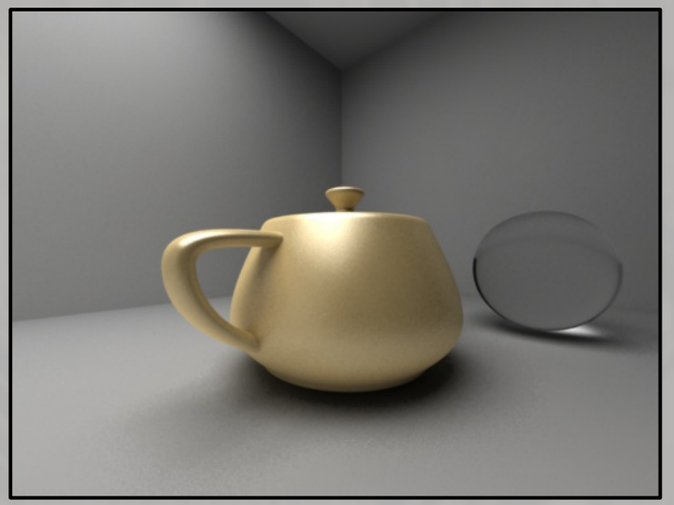
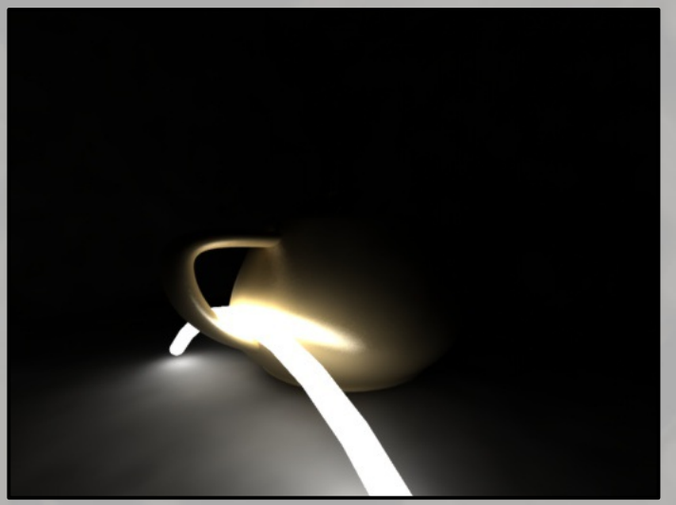
- Лъчите стават много: трябва да трасираме минимум един през всеки пиксел, а за проверките за сенки, броят нараства многократно
- Raytracing-ът е традиционно бавен алгоритъм и силата му е именно в offline рендерирането
  - Например за CG ефекти във филмите

Примитиви при raytracing

- За да може даден обект да се рендерира чрез raytracing, трябва да можем да го пресечем с лъч
- Това включва много широк клас от обекти:
  - Сфера, конус, цилиндър, тор, равнина, ...
  - Произволен многостен (т.е. триъгълна мрежа)
  - Ротационни повърхности, някои видове сплайн сегменти
  - Релефни карти, неявни повърхнини
  - 3D Фрактали
  - И други...

Осветление

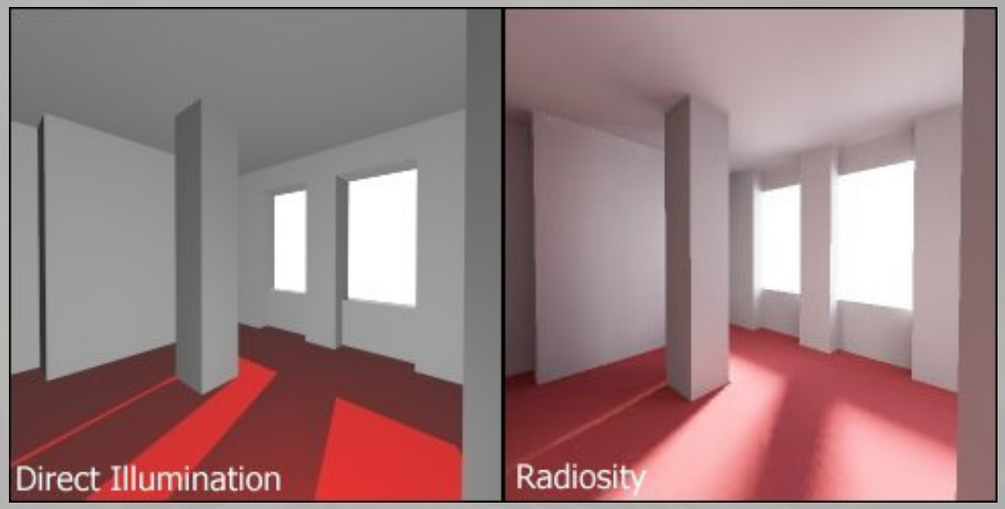
- С рейтрейсинга може да симулираме и разнообразие от светлинни източници
  - Точков и правоъгълен светлинен източник
  - Насочена светлина
  - Източник с произволна геометрия

Глобално осветление

- При рейтрейсинга, осветлението на дадена точка може да се изрази като сума приносите от видимите светлинни източници; това се нарича директно осветление
- Можем и да го изразим като сума от светлината, идваща от всички посоки, включително и други обекти, принципно неизлъчващи светлина; това се нарича глобално (индиректно) осветление и допринася много за фотореалистичността на картината

Глобално осветление



- Глобалното осветление не е специфично за рейтрейсинга; може да се приложи и при Z-buffer алгоритъма, но изисква преизчисляване, за което обикновено пак се ползва raytracing.

Право vs Обратно трасиране

- При raytracing, лъчите се движат в посока от окото към светлинните източници. Това е обратно на посоката в природата и се нарича „прав raytracing“
  - Реално посоката не представлява проблем, тъй като повечето взаимодействия между светлината и материята са симетрични
- Съществуват и алгоритми, базирани на обратен raytracing, но са доста неефективни и се използват за специфични цели
  - Една малка свещ излъчва около 10^{17} фотона за секунда!

Недостатъци на Raytracing-а

- Скорост
- Алгоритмите са тежки, лесно е да се объркат, а е трудно да се провери дали работят вярно
- Сцените с много обекти отнемат огромно количество памет

Сложност на raytracing алгоритмите

- N – брой обекти (елементарни примитиви)
  M – брой пиксели
- При пресичането на лъч със сцената, трябва да се обходят всички обекти, за да се провери коя пресечна точка е най-близка
- Сложността е $O(N \cdot M)$ при наивна имплементация
- Но съществуват дървета за бързо пресичане, чрез които търсенето става логаритмично; сложността при тях е $O(M \cdot log(N))$

Scanline rendering

- Scanline е третият алгоритъм, който е широко използван
  - Например, стандартния рендер на 3ds Max.
- Основната идея тук е да се прекара една виртуална равнина, която да отсича цялата сцена; равнината съвпада с един ред пиксели от екрана, т.е. обектите, които пресича, са тези, които ще бъдат изчертани на този ред от екрана

Scanline rendering

- Реално се поддържа един списък с активните обекти (които пресичат сканиращата линия). При преминаване на следващ ред, този списък трябва да се обнови (т.е. някои от обектите отпадат, а се появяват някои нови)
- Списъкът се пази сортиран по X координатите на ръбовете на обектите
- Рендерирането на единичен ред се състои от обхождане на списъка за определяне на видимите обхвати; те се изчертават с подходящите цветове, подобно на растеризацията при Z-buffer

Предимства

- Не е необходимо да се пазят всички обекти в паметта; достатъчно е да са сортирани по Y - алгоритъмът ги взима един по един, като никога не се връща назад.
- Поради същата причина, алгоритъмът разглежда даден връх само веднъж
- Няма overdraw – всеки пиксел се разглежда еднократно
- В случай, че трябва да се справяме с проблемите като при Painter's algorithm, можем да използваме Z-buffer и тук – само че ни е достатъчен само един ред

Недостатъци

- Ако обектите са много, временните структури (списъка с активните обекти) стават големи и достигат размера на пълен Z-buffer. В такъв случай Scanline губи преимуществото си пред Z-buffer
- Алгоритъмът е труден за паралелизиране и не се поддържа от видео хардуера
- Споделя повечето от недостатъците на Z-buffer

Сравнение

- Z-buffer е бърз, прост, с добра хардуерна поддръжка
- Времето за рендериране при Z-buffer расте линейно спрямо броя триъгълници. Разделителната способност не оказва такова голямо значение
- При добра имплементация, времето за рендериране при Raytracing расте логаритмично спрямо броя триъгълници, и линейно спрямо разделителната способност

Сравнение

- Т.е., Raytracing има капацитета да се справи с огромно количество триъгълници, и след даден момент, започва да се справя по-добре от Z-buffer
  - Например, от OpenRT дават пример за сцена с 10^9 триъгълника, която те рендерират с интерактивни скорости (5-6 fps). Това няма как да се постигне на съвременните видеокарти.
- Raytracing се паралелизира по-добре от другите два
  - Това обуславя ползването на рендер-ферми от студията

Сравнение

- Scanline все повече „губи позиции“ спрямо другите два
- С появата на API-та с общо предназначение за видеокартите (като CUDA и OpenCL), бариерите между raytracing алгоритъма и растеризиращия хардуер се размиват

Регистрирахте ли се?

Въпроси?