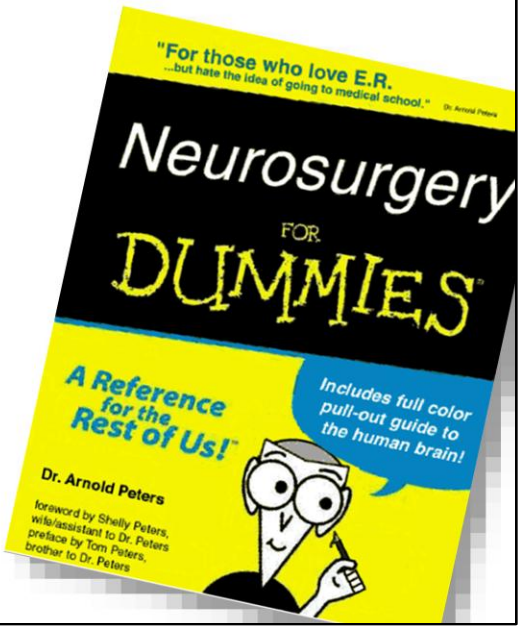
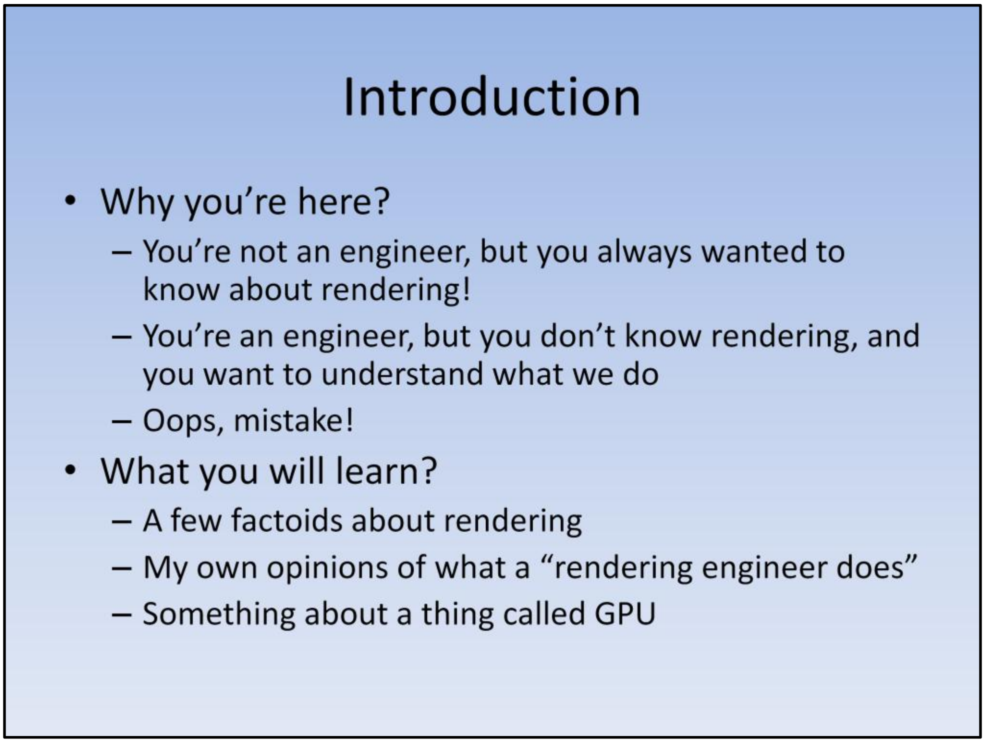
Rendering 101

For videogames


"For those who love E.R.
...but hate the idea of going to medical school."
Dr. Arnold Peters
Neurosurgery
FOR
DUMMIES
A Reference for the Rest of Us!
Includes full color pull-out guide to the human brain!
Dr. Arnold Peters
foreword by Shelly Peters, wife/assistant to Dr. Peters
preface by Tom Peters, brother to Dr. Peters

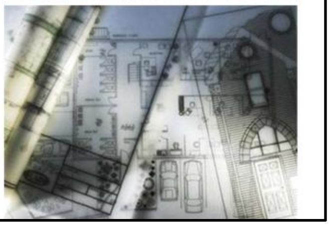


Introduction

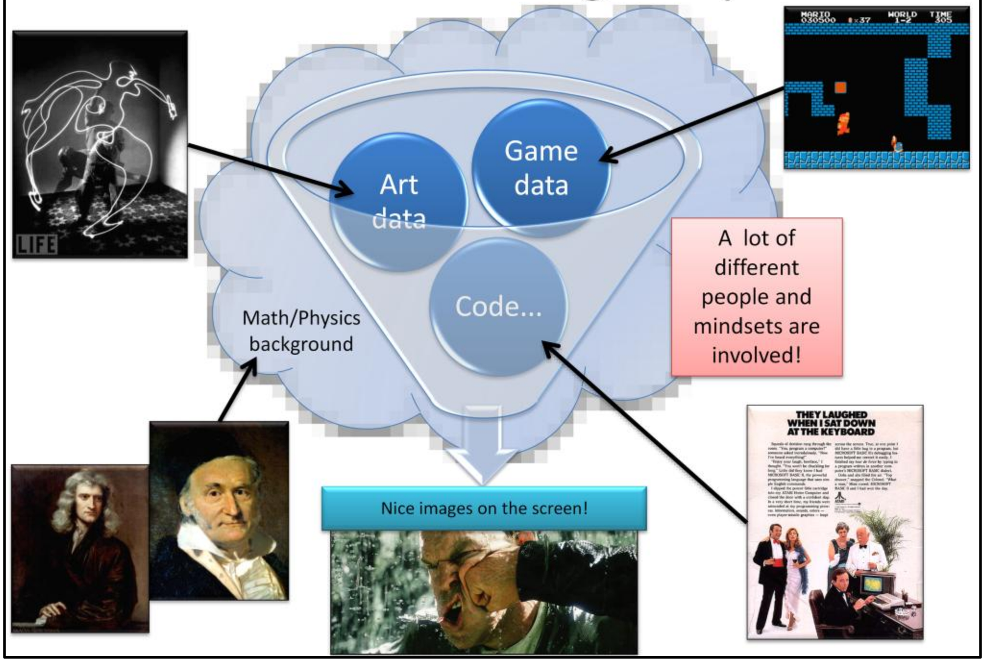
- Why you’re here?
  - You’re not an engineer, but you always wanted to know about rendering!
  - You’re an engineer, but you don’t know rendering, and you want to understand what we do
  - Oops, mistake!
- What you will learn?
  - A few factoids about rendering
  - My own opinions of what a “rendering engineer does”
  - Something about a thing called GPU

Why rendering?

- Artists can create images
  - A sequence of images creates an animation
- But in a game the player is in control
  - We go right, we have to show an image depicting what's on the right
- We need a method to create images "on the fly"
  - Artists define objects, colours, where lights are etc...
  - A scheme...
  - Rendering converts this into images



Game Rendering Recipe



Art data

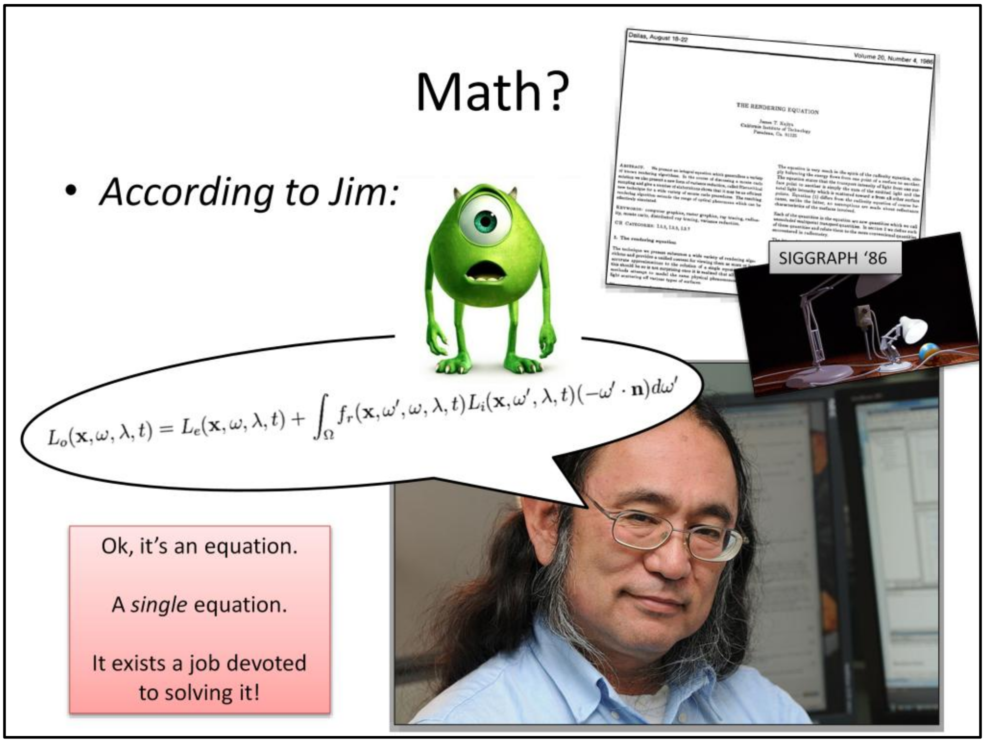
Game data

Code...

Math/Physics background

A lot of different people and mindsets are involved!

Nice images on the screen!



Math?

- *According to Jim:*

$$L_o(\mathbf{x},\omega,\lambda,t) = L_e(\mathbf{x},\omega,\lambda,t) + \int_\Omega f_r(\mathbf{x},\omega',\omega,\lambda,t)L_i(\mathbf{x},\omega',\lambda,t)(-\omega'\cdot\mathbf{n})d\omega'$$

Ok, it's an equation.

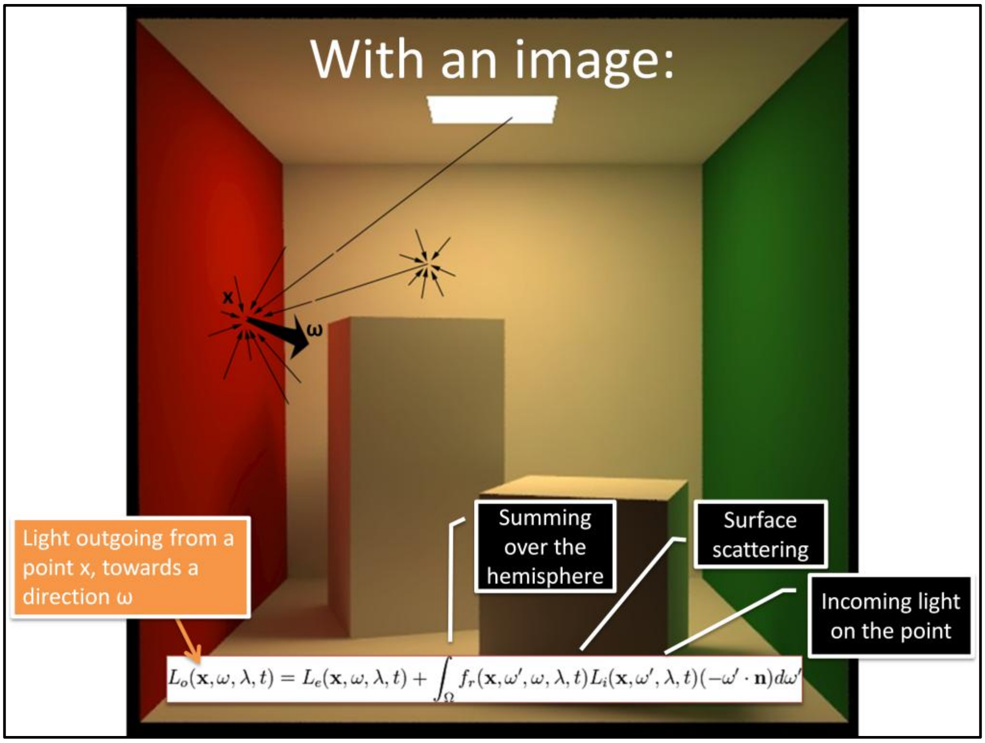
A *single* equation.

It exists a job devoted to solving it!

Jim Kajiya... The author of the rendering equation

Hawkins wrote in the preface of one of his books that his editor once told him that for each equation in his book, he would lose half of the audience. So he included only one, and that's what I'll do too...

The equation is in an ugly form, I took the image from Wikipedia



With an image:

Light outgoing from a point x, towards a direction ω

Summing over the hemisphere

Surface scattering

Incoming light on the point

$$L_o(\mathbf{x},\omega,\lambda,t) = L_e(\mathbf{x},\omega,\lambda,t) + \int_\Omega f_r(\mathbf{x},\omega',\omega,\lambda,t)L_i(\mathbf{x},\omega',\lambda,t)(-\omega'\cdot\mathbf{n})d\omega'$$

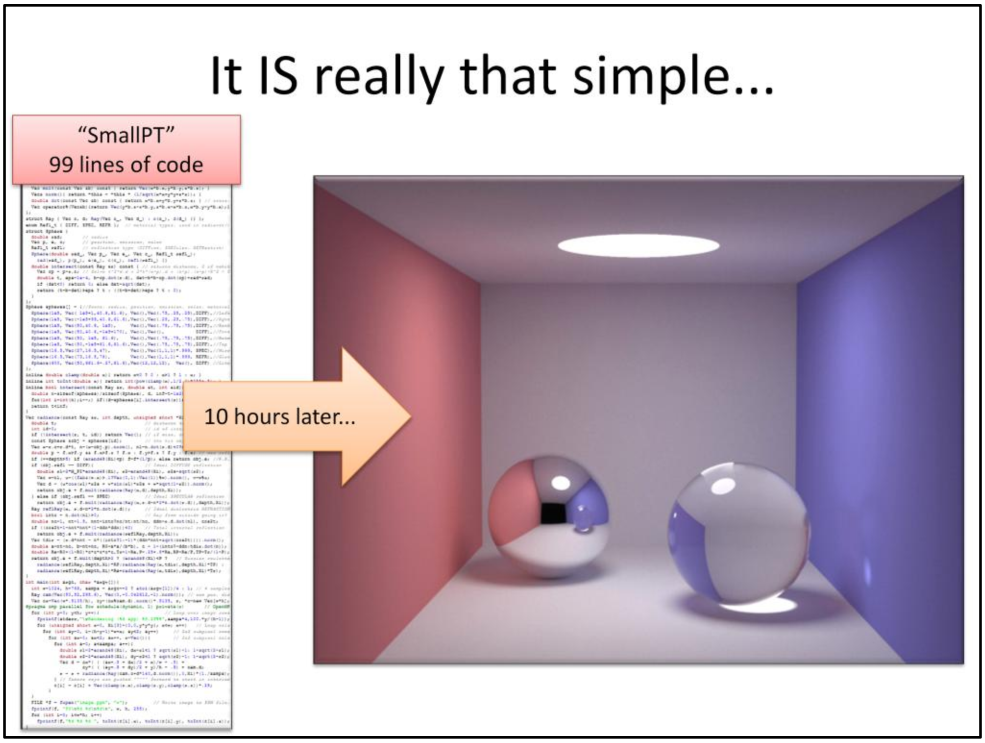
To generate an image, we want to know from every point in the scene the light that reaches our virtual eye.
The basic device is something that tells us how much light from a point goes to a given direction, this is what the lighting equation describes.

We want to compute the light outgoing (“Lo”) from the big black arrow (radiance), emitting from a point towards our sensor (camera/eye):

1) We need to know all the light that arrieves on that point. Only lights emit light (that is accounted by the term Le, that is non-zero if the point x is on a light), all the other points in the scene just reflect some energy that arrieves from somewhere else...
   In math, this is an integral (sum) over an hemisphere (a point can’t receive light from directions behind its surface) of the “Li” (incoming light)
   This quantity is called “irradiance”
2) Each ray of light that reaches our point is scattered by the surface material in some direction, with some intensity
   A function defines the properties of the material, in the equation is the “fr(...)” term.
   It’s called BRDF – bi-directional (as it takes two angles, the incoming one from Li and the outgoing one we’re computing Lo on) reflectance distribution function

The problem is that we don’t know the light incoming (Li) from all the directions on that point! To compute that, we need to solve another light-from-a-point-towards-a-direction kind of problem that is, our equation is recursive.

In general, even to compute effects as simple and fundamental as shadows, we have to consider the other objects in the scene to find the color of each other (thus we call these effects “global” illumination)



It IS really that simple...

"SmallPT"
99 lines of code

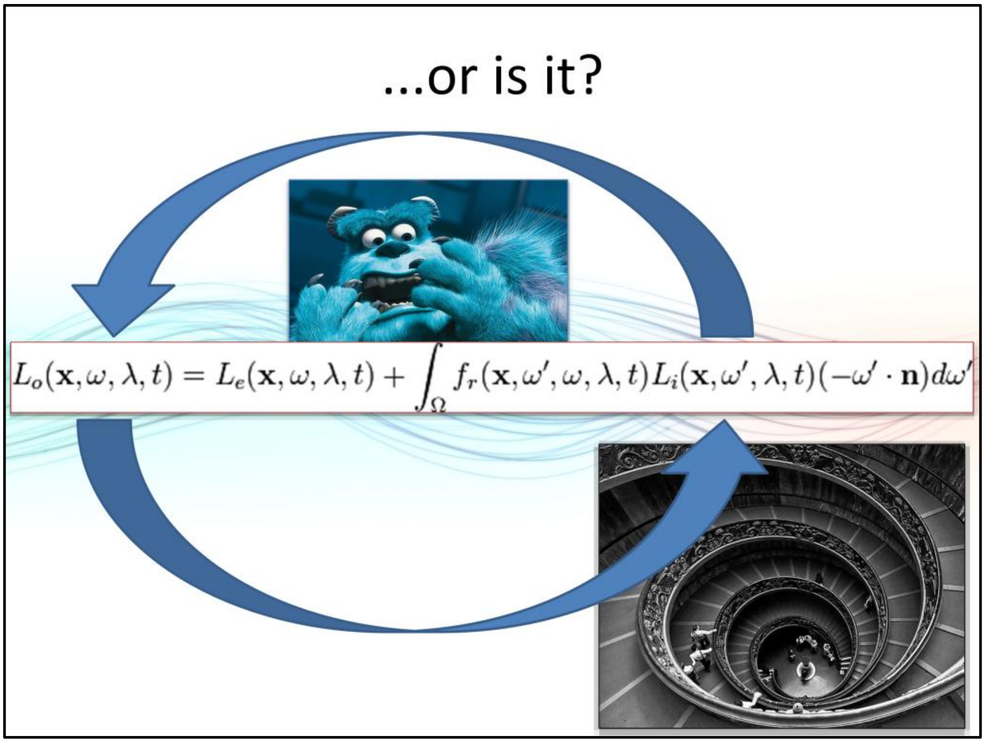
10 hours later...

The very core of rendering, is all in that equation... well, plus some signal/sampling theory as you're going to make an image out of pixels, some knowledge about colors and human perception and a sprinkle of geometry in order to have some math that is able to represent your scene... Et voilà! (at least for static images, in theory...)

SmallPT: path tracing in 99 lines of C++ code: http://www.kevinbeason.com/smallpt/

The image took 10 hours on a Intel Core 2 Quad, 2.4ghz, using 4 threads

The CUDA version is very interesting: http://code.google.com/p/tokaspt/

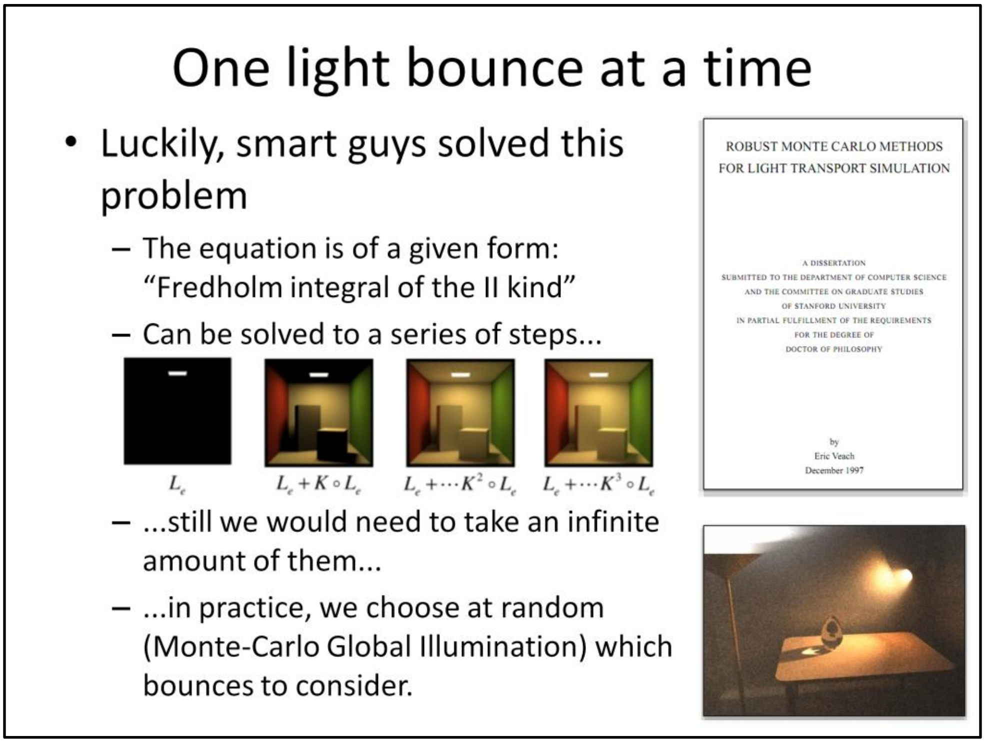


...or is it?

$$L_o(\mathbf{x},\omega,\lambda,t) = L_e(\mathbf{x},\omega,\lambda,t) + \int_\Omega f_r(\mathbf{x},\omega',\omega,\lambda,t)L_i(\mathbf{x},\omega',\lambda,t)(-\omega'\cdot\mathbf{n})d\omega'$$

Aaaaaarrrrghhh... We need to know the light in order to compute the light... Infinite!

Under some conditions though, it’s like a spiral, it converges to a point, which we can compute.



One light bounce at a time

- Luckily, smart guys solved this problem
  - The equation is of a given form: “Fredholm integral of the II kind”
  - Can be solved to a series of steps...

L_e $L_e + K \circ L_e$ $L_e + \cdots K^2 \circ L_e$ $L_e + \cdots K^3 \circ L_e$

  - ...still we would need to take an infinite amount of them...
  - ...in practice, we choose at random (Monte-Carlo Global Illumination) which bounces to consider.

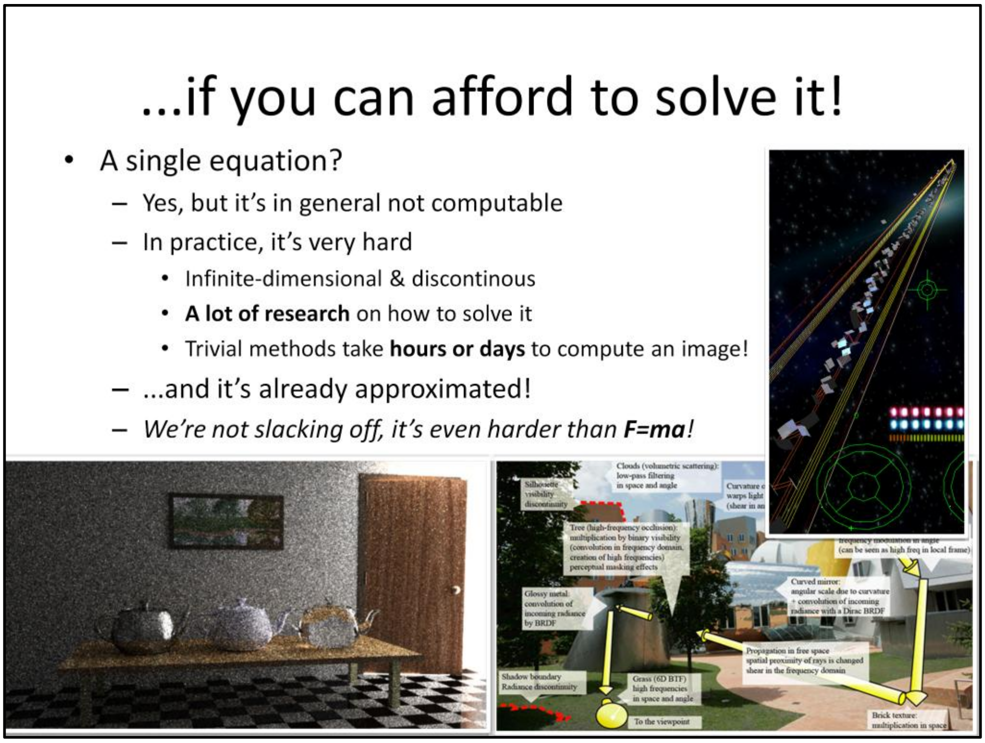
ROBUST MONTE CARLO METHODS
FOR LIGHT TRANSPORT SIMULATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

by
Eric Veach
December 1997

The series solution is called “Liouville-Neumann” series.

The best reference for this theory (even if it’s quite heavy in terms of math) is Eric Veach’s dissertation thesis.



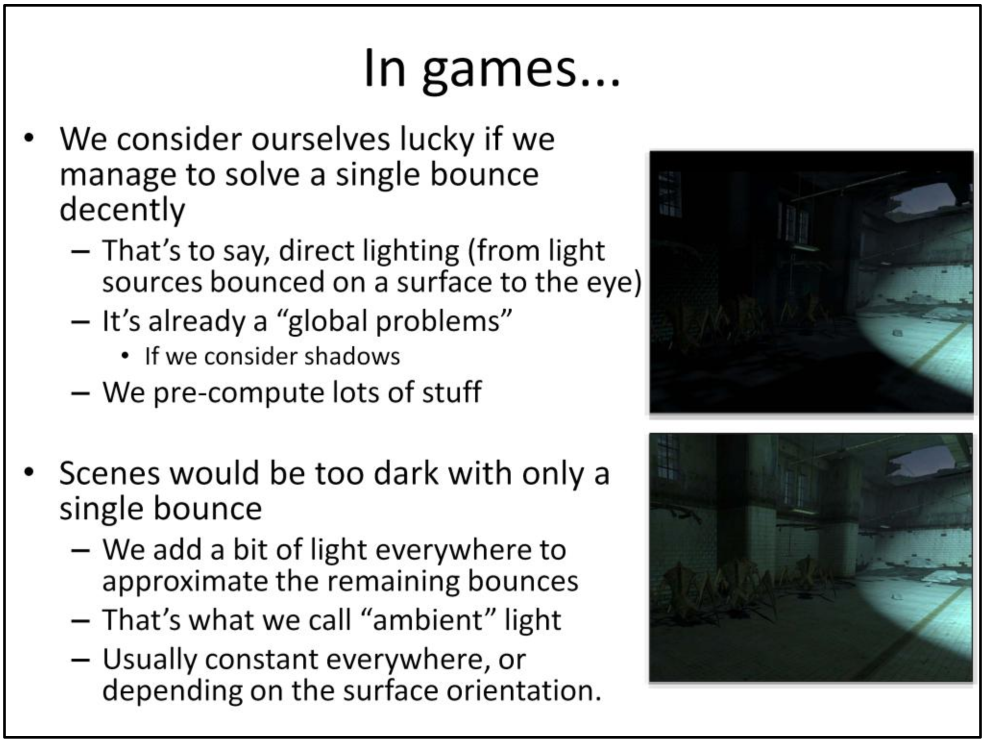
...if you can afford to solve it!

- A single equation?
  - Yes, but it’s in general not computable
  - In practice, it’s very hard
    - Infinite-dimensional & discontinous
    - **A lot of research** on how to solve it
    - Trivial methods take **hours or days** to compute an image!
  - ...and it’s already approximated!
  - *We’re not slacking off, it’s even harder than* ***F=ma****!*

Clouds (volumetric scattering):
low-pass filtering
in space and angle
Silhouette
visibility
discontinuity
Curvature o
warps light
(shear in an
Tree (high-frequency occlusion):
multiplication by binary visibility
(convolution in frequency domain,
creation of high frequencies)
perceptual masking effects
(can be seen as high freq in local frame)
Glossy metal:
convolution of
incoming radiance
by BRDF
Curved mirror:
angular scale due to curvature
+ convolution of incoming
radiance with a Dirac BRDF
Propagation in free space
spatial proximity of rays is changed
shear in the frequency domain
Shadow boundary
Radiance discontinuity
Grass (6D BTF)
high frequencies
in space and angle
To the viewpoint
Brick texture:
multiplication in space

Not computable: search for the “raytrix” at Stanford

Approximated: we consider only straight rays, the equation here does not consider that the light penetrates into some materials and exists at a different point (i.e. Wax, human skin), does not consider other minor effects like fluorescence and phosphorescence, diffraction... All these are conceptually easy but trivial solutions are intractably slow.



In games…

- We consider ourselves lucky if we manage to solve a single bounce decently
  - That's to say, direct lighting (from light sources bounced on a surface to the eye)
  - It's already a "global problems"
    - If we consider shadows
  - We pre-compute lots of stuff
- Scenes would be too dark with only a single bounce
  - We add a bit of light everywhere to approximate the remaining bounces
  - That's what we call "ambient" light
  - Usually constant everywhere, or depending on the surface orientation.

Images from:
http://www.valvesoftware.com/publications/2006/SIGGRAPH06_Course_ShadingInValvesSourceEngine_Slides.pdf

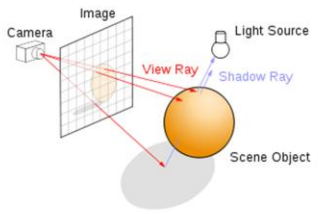
Another problem

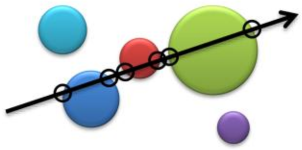
- We went a little bit too fast...
  - Light “from a point”
  - Ok but which point?
  - Which points are visible from our virtual eye?
  - Which points are visible from another point in the scene?

- Rendering =
  - Visibility (which surfaces do we see?)
  - Shading (what colour a given surface has?)

Visiblity: Raytracing

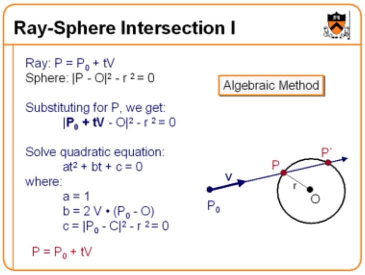
- Let’s assume that our scene is made of spheres
- Let’s draw lines from our virtual eye outwards toward the scene
- Can we find where these lines intersect the spheres?
- Can we find which one is the closest?





It turns out we can…

- Some basic math…



Ray-Sphere Intersection I

Ray: $P = P_0 + tV$
Sphere: $|P - O|^2 - r^2 = 0$

Algebraic Method

Substituting for P, we get:
$|\mathbf{P_0} + \mathbf{tV} - O|^2 - r^2 = 0$

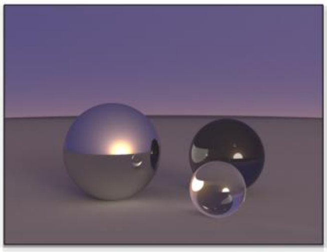
Solve quadratic equation:
$at^2 + bt + c = 0$
where:
$a = 1$
$b = 2V \cdot (P_0 - O)$
$c = |P_0 - C|^2 - r^2 = 0$

$P = P_0 + tV$

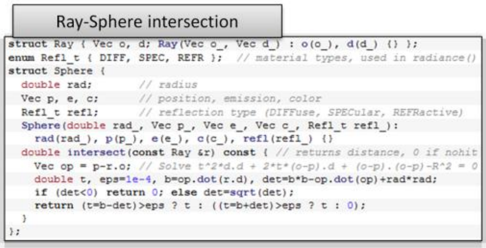
- Rinse and repeat:

- We can do the same for many other primitives
  - http://www.realtimerendering.com/intersections.html

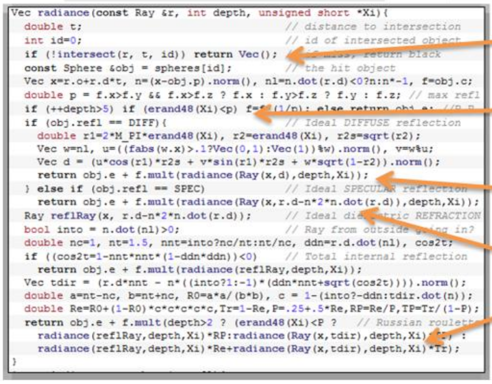
Back to SmallPT source code



Ray-Sphere intersection



```
struct Ray { Vec o, d; Ray(Vec o_, Vec d_) : o(o_), d(d_) {} };
enum Refl_t { DIFF, SPEC, REFR };  // material types, used in radiance()
struct Sphere {
  double rad;       // radius
  Vec p, e, c;      // position, emission, color
  Refl_t refl;      // reflection type (DIFFuse, SPECular, REFRactive)
  Sphere(double rad_, Vec p_, Vec e_, Vec c_, Refl_t refl_):
    rad(rad_), p(p_), e(e_), c(c_), refl(refl_) {}
  double intersect(const Ray &r) const { // returns distance, 0 if nohit
    Vec op = p-r.o; // Solve t^2*d.d + 2*t*(o-p).d + (o-p).(o-p)-R^2 = 0
    double t, eps=1e-4, b=op.dot(r.d), det=b*b-op.dot(op)+rad*rad;
    if (det<0) return 0; else det=sqrt(det);
    return (t=b-det)>eps ? t : ((t=b+det)>eps ? t : 0);
  }
};
```



```
Vec radiance(const Ray &r, int depth, unsigned short *Xi){
  double t;                               // distance to intersection
  int id=0;                               // id of intersected object
  if (!intersect(r, t, id)) return Vec(); // if miss, return black
  const Sphere &obj = spheres[id];        // the hit object
  Vec x=r.o+r.d*t, n=(x-obj.p).norm(), nl=n.dot(r.d)<0?n:n*-1, f=obj.c;
  double p = f.x>f.y && f.x>f.z ? f.x : f.y>f.z ? f.y : f.z; // max refl
  if (++depth>5) if (erand48(Xi)<p) f=f*(1/p); else return obj.e; //R.R
  if (obj.refl == DIFF){                  // Ideal DIFFUSE reflection
    double r1=2*M_PI*erand48(Xi), r2=erand48(Xi), r2s=sqrt(r2);
    Vec w=nl, u=((fabs(w.x)>.1?Vec(0,1):Vec(1))%w).norm(), v=w%u;
    Vec d = (u*cos(r1)*r2s + v*sin(r1)*r2s + w*sqrt(1-r2)).norm();
    return obj.e + f.mult(radiance(Ray(x,d),depth,Xi));
  } else if (obj.refl == SPEC)            // Ideal SPECULAR reflection
    return obj.e + f.mult(radiance(Ray(x,r.d-n*2*n.dot(r.d)),depth,Xi));
  Ray reflRay(x, r.d-n*2*n.dot(r.d));     // Ideal dielectric REFRACTION
  bool into = n.dot(nl)>0;                // Ray from outside going in?
  double nc=1, nt=1.5, nnt=into?nc/nt:nt/nc, ddn=r.d.dot(nl), cos2t;
  if ((cos2t=1-nnt*nnt*(1-ddn*ddn))<0)    // Total internal reflection
    return obj.e + f.mult(radiance(reflRay,depth,Xi));
  Vec tdir = (r.d*nnt - n*((into?1:-1)*(ddn*nnt+sqrt(cos2t)))).norm();
  double a=nt-nc, b=nt+nc, R0=a*a/(b*b), c = 1-(into?-ddn:tdir.dot(n));
  double Re=R0+(1-R0)*c*c*c*c*c,Tr=1-Re,P=.25+.5*Re,RP=Re/P,TP=Tr/(1-P);
  return obj.e + f.mult(depth>2 ? (erand48(Xi)<P ?   // Russian roulette
    radiance(reflRay,depth,Xi)*RP:radiance(Ray(x,tdir),depth,Xi)*TP) :
    radiance(reflRay,depth,Xi)*Re+radiance(Ray(x,tdir),depth,Xi)*Tr);
}
```

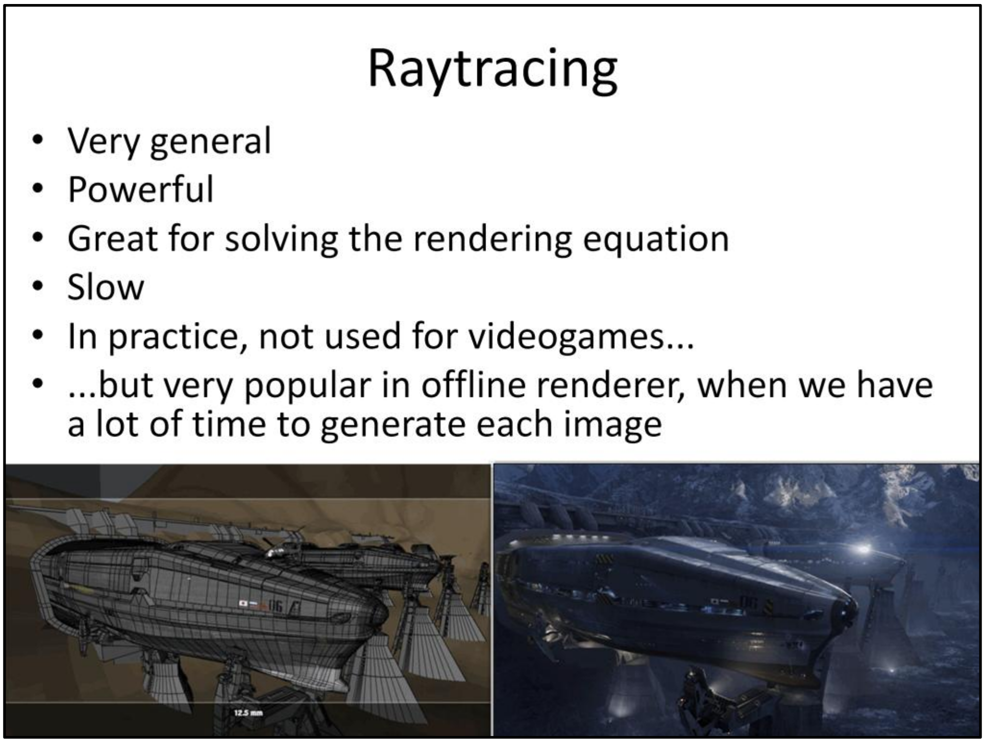
Ray-Scene Intersection

Russian-roulette, stop the recursion at a random depth

Materials: Diffuse (Lambertian) shoots a ray in a random direction around the hemisphere

Materials: Pure mirror shoots a ray directly towards the reflection vector

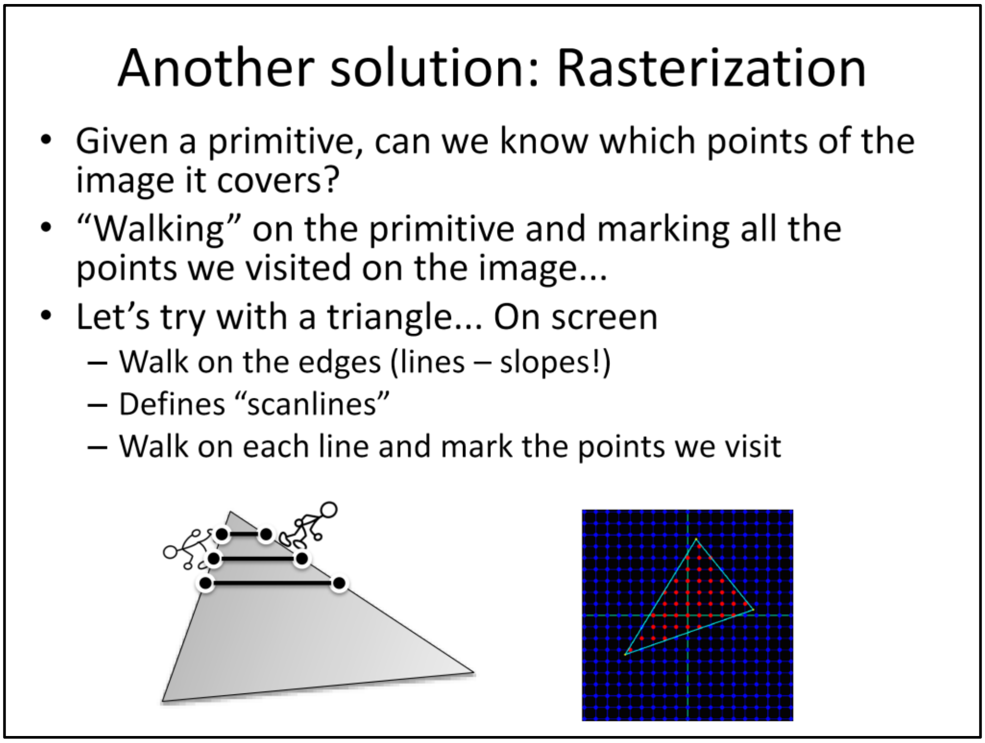
Materials: Transparent refraction



Raytracing

- Very general
- Powerful
- Great for solving the rendering equation
- Slow
- In practice, not used for videogames...
- ...but very popular in offline renderer, when we have a lot of time to generate each image

Images from http://www.awn.com/articles/article/2012-end-world-we-know-it/page/4,1



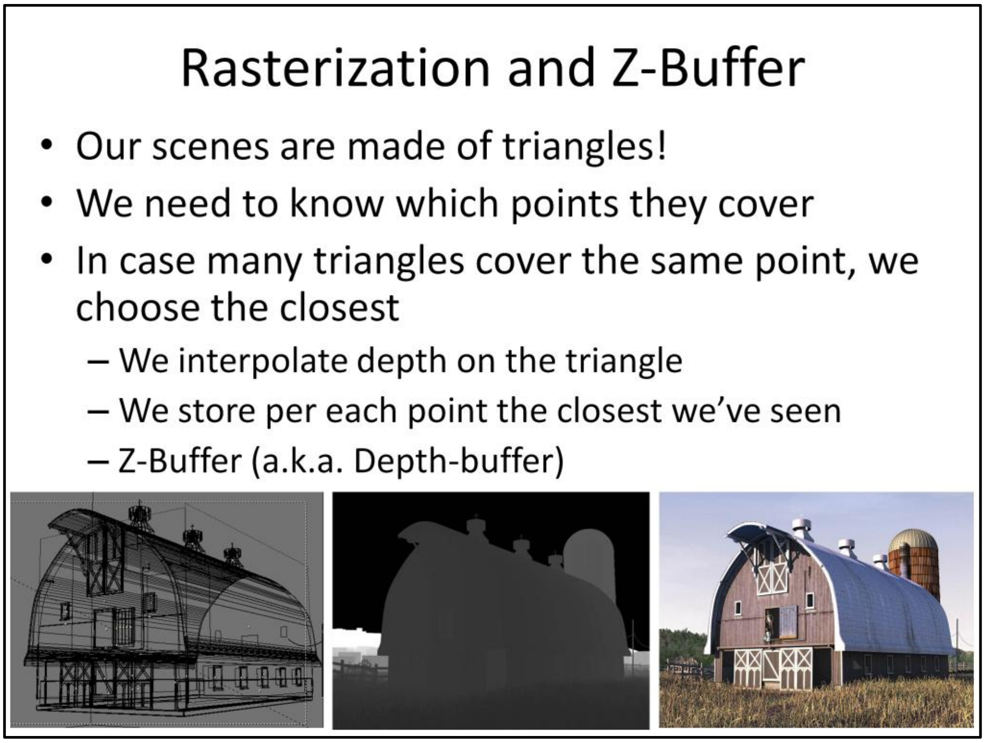
Another solution: Rasterization

- Given a primitive, can we know which points of the image it covers?
- “Walking” on the primitive and marking all the points we visited on the image...
- Let’s try with a triangle... On screen
  - Walk on the edges (lines – slopes!)
  - Defines “scanlines”
  - Walk on each line and mark the points we visit

Writing a software rasterizer is cool

Scanline:
http://www.flipcode.com/documents/fatmap.txt
http://chrishecker.com/Miscellaneous_Technical_Articles#Perspective_Texture_Mapping

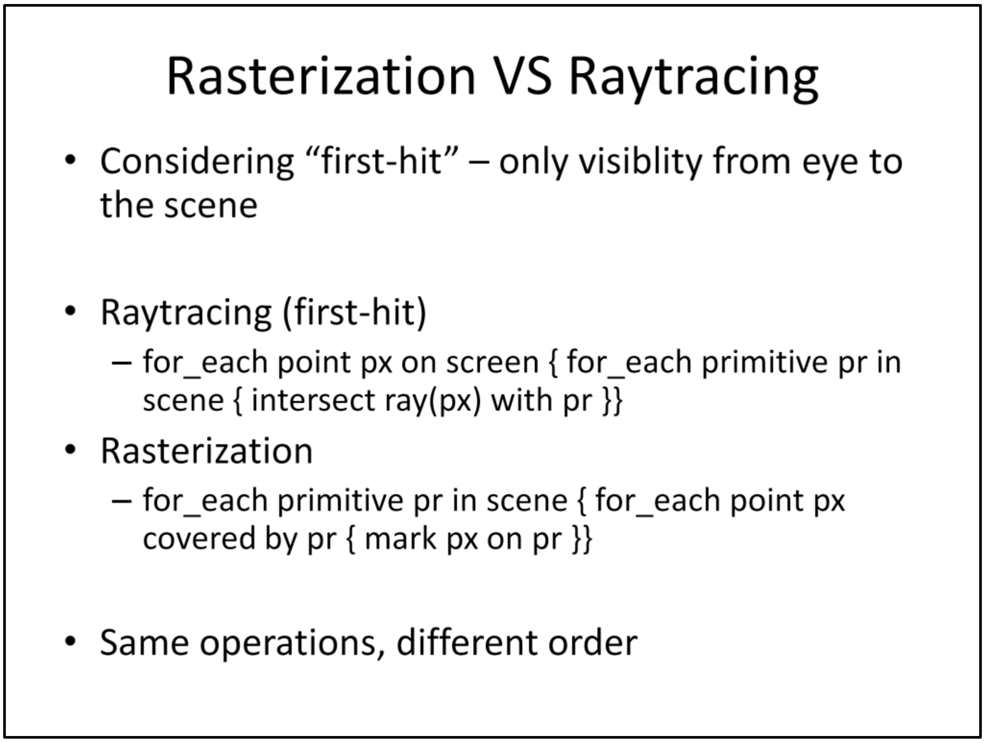
Half-plane:
http://www.devmaster.net/forums/showthread.php?t=1884



Rasterization and Z-Buffer

- Our scenes are made of triangles!
- We need to know which points they cover
- In case many triangles cover the same point, we choose the closest
  - We interpolate depth on the triangle
  - We store per each point the closest we've seen
  - Z-Buffer (a.k.a. Depth-buffer)

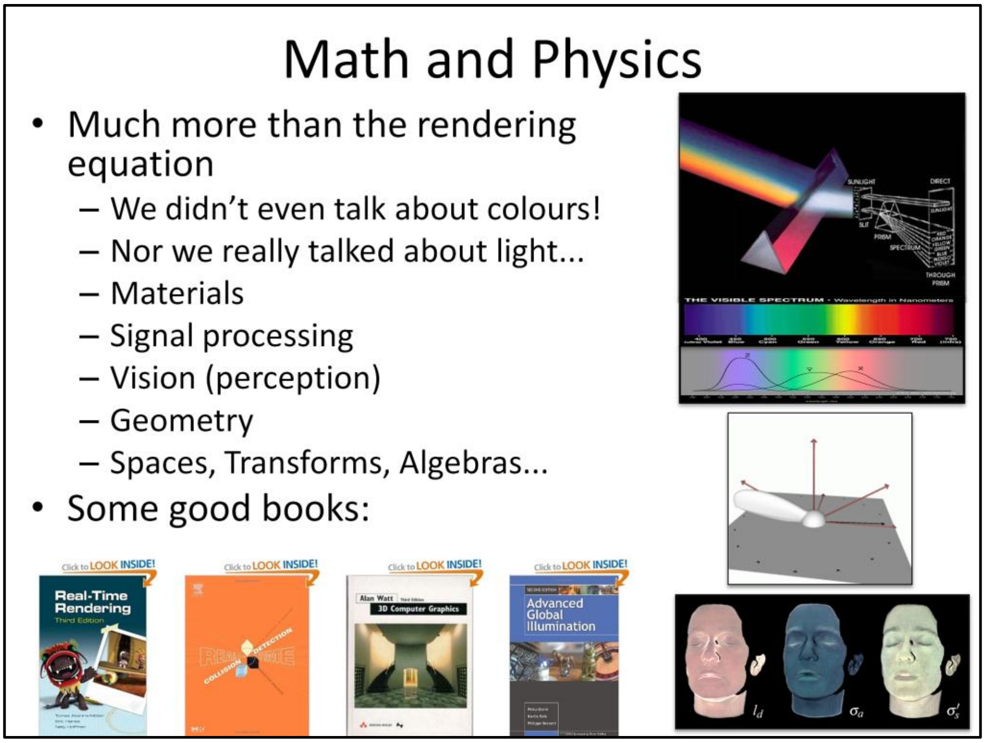
Images from: http://www.harkyman.com/2005/05/22/sears-kit-barn-tutorial/



Rasterization VS Raytracing

- Considering “first-hit” – only visiblity from eye to the scene
- Raytracing (first-hit)
  - for_each point px on screen { for_each primitive pr in scene { intersect ray(px) with pr }}
- Rasterization
  - for_each primitive pr in scene { for_each point px covered by pr { mark px on pr }}
- Same operations, different order

http://c0de517e.blogspot.com/2011/09/raytracing-myths.html



Math and Physics

- Much more than the rendering equation
  - We didn't even talk about colours!
  - Nor we really talked about light...
  - Materials
  - Signal processing
  - Vision (perception)
  - Geometry
  - Spaces, Transforms, Algebras...
- Some good books:

Amazon links:

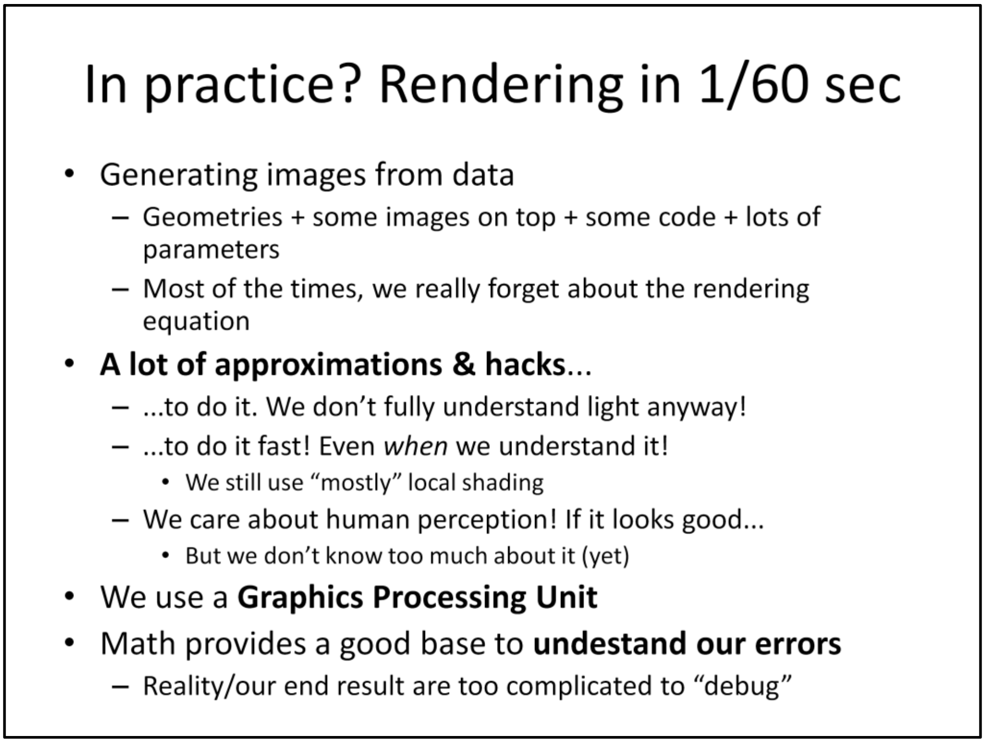
http://www.amazon.com/Real-Time-Rendering-Third-Tomas-Akenine-Moller/dp/1568814240/ref=sr_1_1?ie=UTF8&qid=1317419811&sr=8-1

http://www.amazon.com/Real-Time-Collision-Detection-Interactive-Technology/dp/1558607323/ref=sr_1_13?ie=UTF8&qid=1317419811&sr=8-13

http://www.amazon.com/Computer-Graphics-3rd-Alan-Watt/dp/0201398559/ref=pd_sim_b56

http://www.amazon.com/Advanced-Global-Illumination-Second-Philip/dp/1568813074/ref=pd_sim_b2

http://www.amazon.com/Realistic-Image-Synthesis-Photon-Mapping/dp/1568814623/ref=sr_1_1?s=books&ie=UTF8&qid=1317420066&sr=1-1

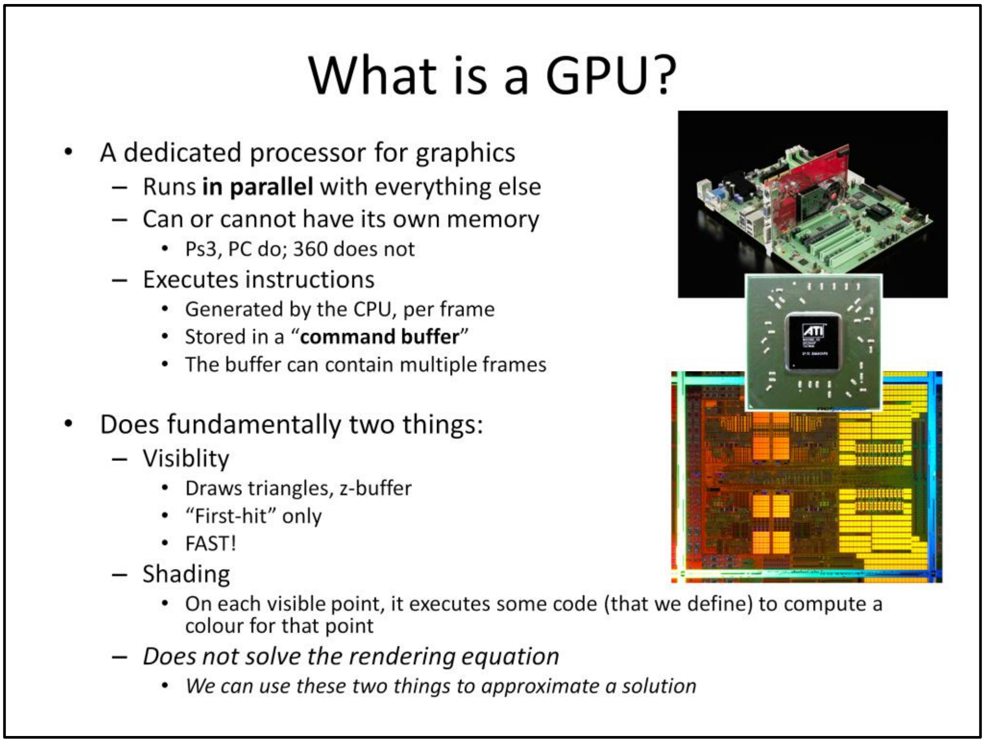


In practice? Rendering in 1/60 sec

- Generating images from data
  - Geometries + some images on top + some code + lots of parameters
  - Most of the times, we really forget about the rendering equation
- **A lot of approximations & hacks**...
  - ...to do it. We don’t fully understand light anyway!
  - ...to do it fast! Even *when* we understand it!
    - We still use “mostly” local shading
  - We care about human perception! If it looks good...
    - But we don’t know too much about it (yet)
- We use a **Graphics Processing Unit**
- Math provides a good base to **undestand our errors**
  - Reality/our end result are too complicated to “debug”

•“mostly” local... Commonly the shading (color) of a surface depends only on the surface and the lights, plus the shadows that are the only widespread global effect

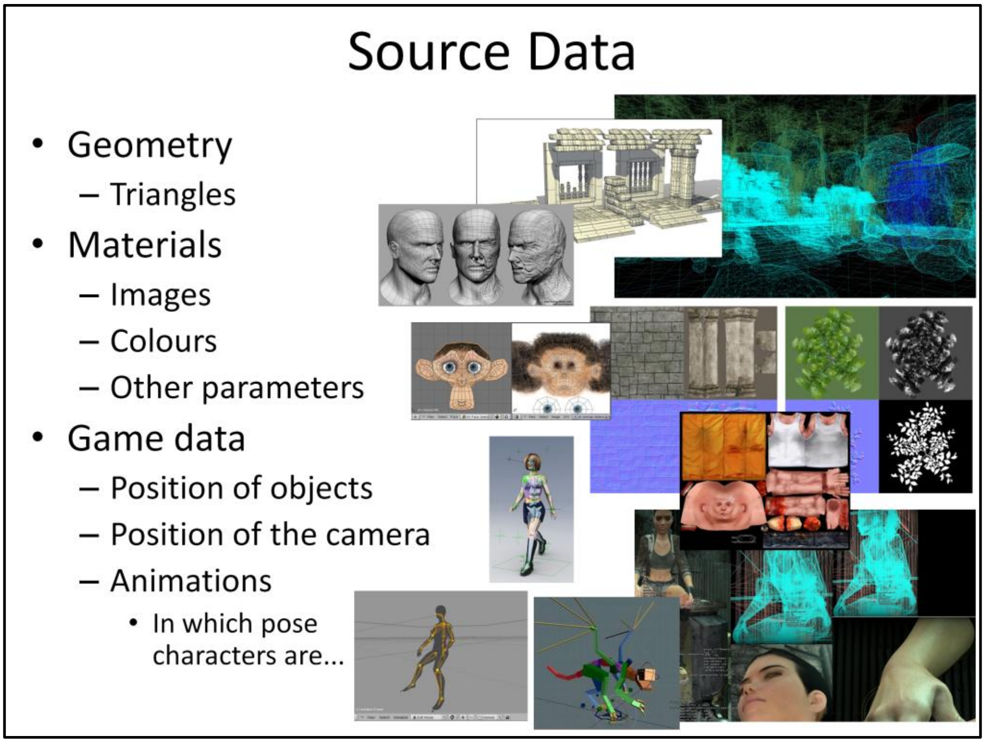
• We still don’t do our math right in a LOT of cases! Makes everything complicated, when your shading in practice depends on hundreds of material paramters (and hacks), different rendering passes/effects, a lot of source data (images, geometry...)



What is a GPU?

- A dedicated processor for graphics
  - Runs **in parallel** with everything else
  - Can or cannot have its own memory
    - Ps3, PC do; 360 does not
  - Executes instructions
    - Generated by the CPU, per frame
    - Stored in a "**command buffer**"
    - The buffer can contain multiple frames
- Does fundamentally two things:
  - Visiblity
    - Draws triangles, z-buffer
    - "First-hit" only
    - FAST!
  - Shading
    - On each visible point, it executes some code (that we define) to compute a colour for that point
  - *Does not solve the rendering equation*
    - *We can use these two things to approximate a solution*

In detail:
http://c0de517e.blogspot.com/2008/04/gpu-part-1.html
http://c0de517e.blogspot.com/2008/04/how-gpu-works-part-2.html
http://c0de517e.blogspot.com/2008/04/how-gpu-works-part-3.html
http://c0de517e.blogspot.com/2009/05/how-gpu-works-appendix.html
http://c0de517e.blogspot.com/2008/07/gpu-versus-cpu.html



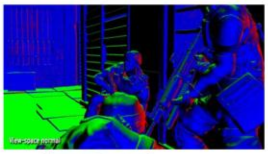
Source Data

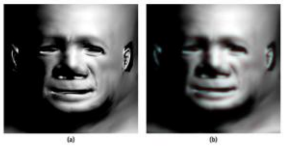
- Geometry
  - Triangles
- Materials
  - Images
  - Colours
  - Other parameters
- Game data
  - Position of objects
  - Position of the camera
  - Animations
    - In which pose characters are...

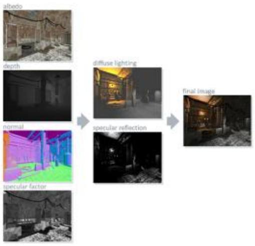
Most images from: http://minifloppy.it/works/1/udk-jungle-environment

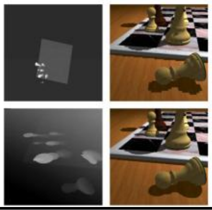
Algorithms

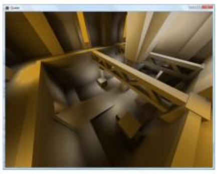
- Drawing triangles is easy
  - How we draw them is the key!
    - How many? Where? When? What color? Those are our tools...
  - How do we approximate the rendering equation by drawing triangles?
    - Many techniques
    - We won't see any of them here

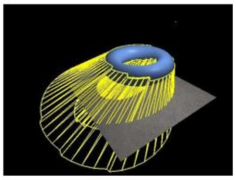


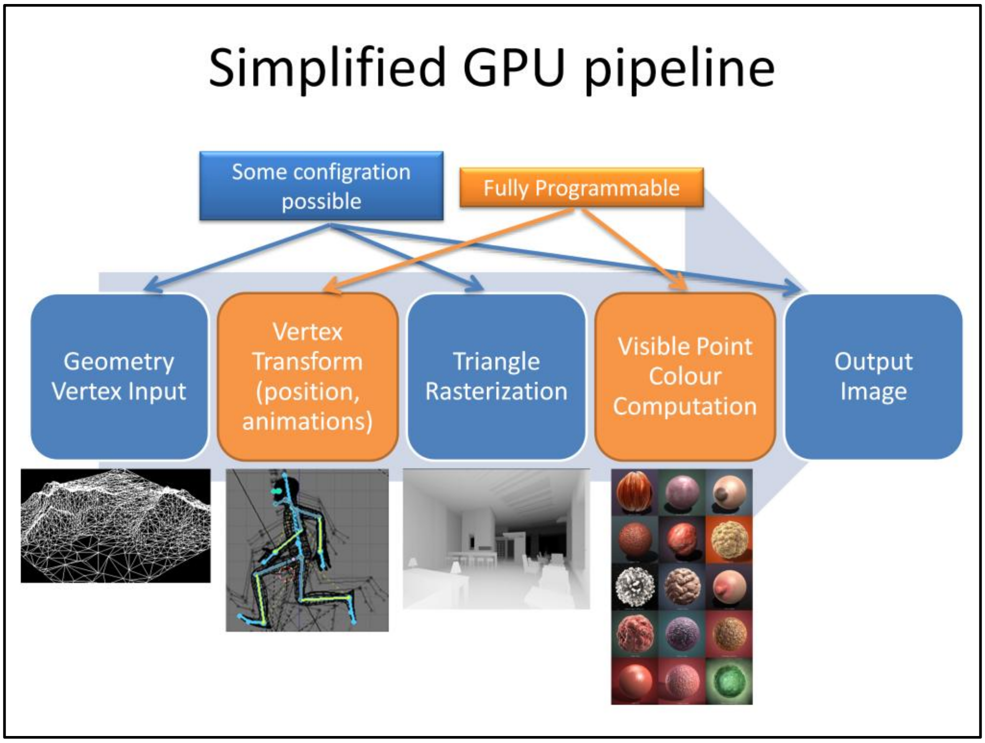












Simplified GPU pipeline

Some configration possible

Fully Programmable

Geometry Vertex Input

Vertex Transform (position, animations)

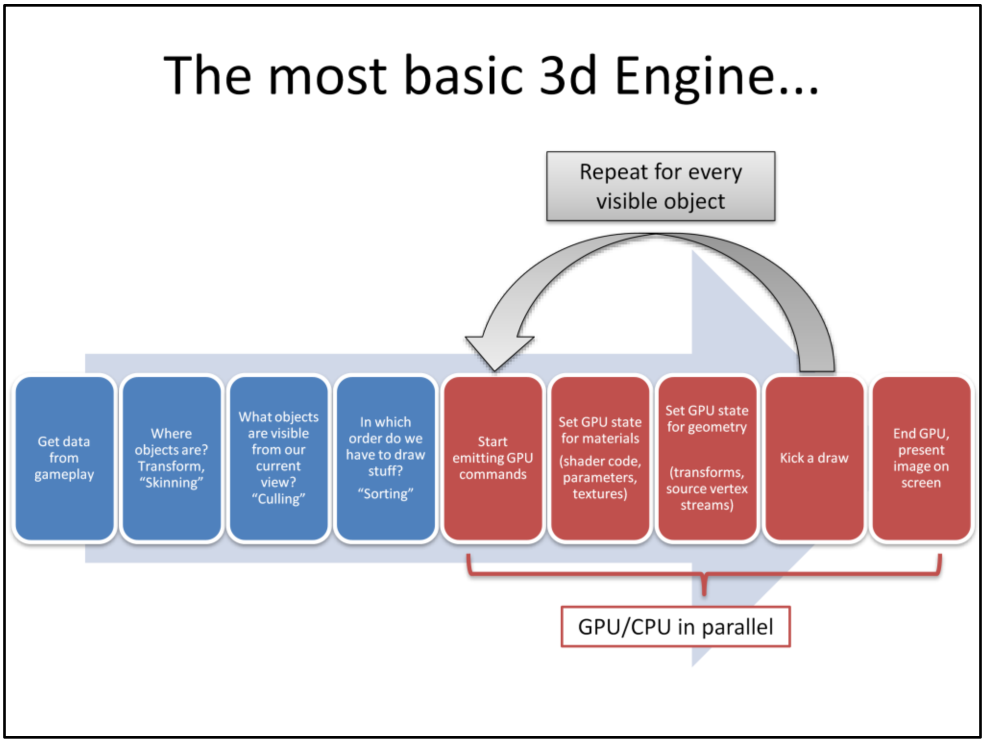
Triangle Rasterization

Visible Point Colour Computation

Output Image

GPU commands

- We can tell the GPU to:
- Set state
  - Turns internal switches on and off
  - Binds resources
    - Images (textures)
    - Vertex data (geometry streams)
    - Code for the programmable stages (shaders)
- Draw
  - Starts the engines!
  - Process a bunch of data with the current settings
  - Draws some triangles in some memory area...
- Present
  - Finish the current frame!
  - Display the contents of a given memory area on screen
- Plus something else here and there...
  - I.E. GPU <-> CPU syncronization primitives



The most basic 3d Engine...

Repeat for every visible object

Get data from gameplay

Where objects are? Transform, “Skinning”

What objects are visible from our current view? “Culling”

In which order do we have to draw stuff? “Sorting”

Start emitting GPU commands

Set GPU state for materials (shader code, parameters, textures)

Set GPU state for geometry (transforms, source vertex streams)

Kick a draw

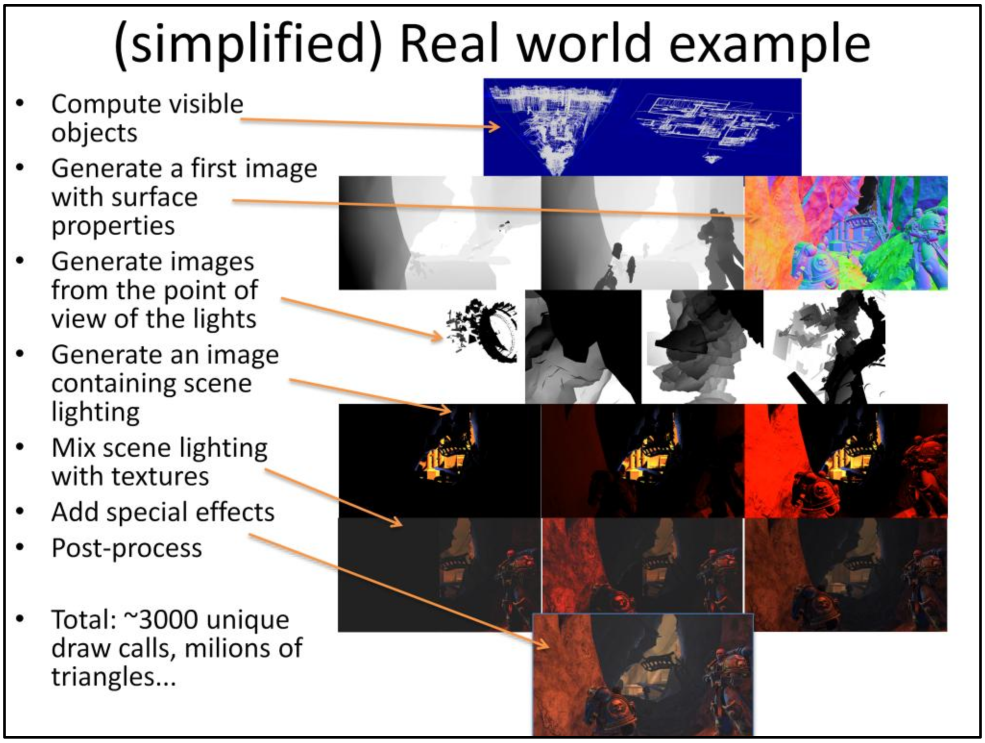
End GPU, present image on screen

GPU/CPU in parallel

Culling is needed because our scenes are often huge, we can’t send _everything_ to the GPU and hope it will generate something fast enough...

Many things can be overlapped between the GPU and CPU, here we highlight only that when we send commands to from the CPU to the GPU, the two chips are operating in parallel, and if they need to talk to each other it needs syncronization. In practice the overlap will be there during the whole execution, usually when the rendering is taking the data from the game the GPU is still busy rendering the previous frame for example.

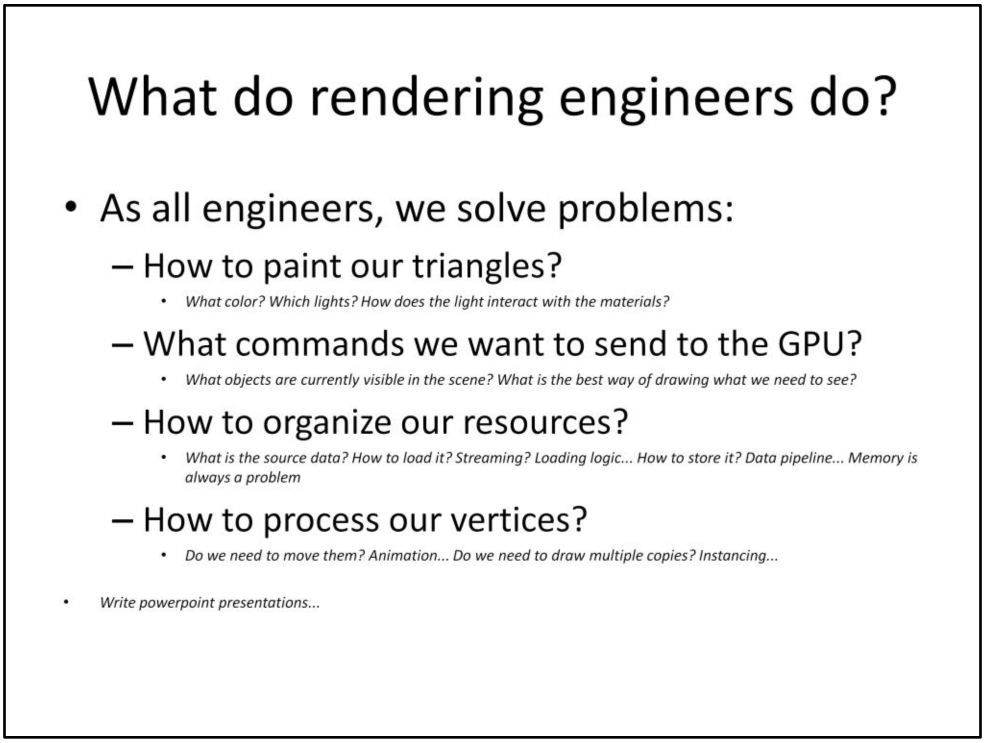
All real renderers also generate an image in many different passes, it might need to render the scene from the point of view of the lights (to cast shadows) and to generate many intermediate views and images, combine them and post-process them and use some as inputs for the rendering of others.



(simplified) Real world example

- Compute visible objects
- Generate a first image with surface properties
- Generate images from the point of view of the lights
- Generate an image containing scene lighting
- Mix scene lighting with textures
- Add special effects
- Post-process
- Total: ~3000 unique draw calls, milions of triangles...

These images were generated by instrumenting a retail PC copy of Space Marines (Relic/THQ): http://www.spacemarine.com/



What do rendering engineers do?

- As all engineers, we solve problems:
  - How to paint our triangles?
    - *What color? Which lights? How does the light interact with the materials?*
  - What commands we want to send to the GPU?
    - *What objects are currently visible in the scene? What is the best way of drawing what we need to see?*
  - How to organize our resources?
    - *What is the source data? How to load it? Streaming? Loading logic... How to store it? Data pipeline... Memory is always a problem*
  - How to process our vertices?
    - *Do we need to move them? Animation... Do we need to draw multiple copies? Instancing...*
- *Write powerpoint presentations...*