EDAN30 Photorealistic Computer Graphics

Ray tracing

Michael Doggett
Department of Computer Science
Lund university

Copyright © 2011 Michael Doggett
Outline

• Recursive ray tracing

• Whitted ray tracing,
  • “An improved illumination model for shaded display”, Turner Whitted, CACM June 1980
What is ray tracing?

• Another rendering algorithm
  – Fundamentally different from polygon rendering

• Rasterization (OpenGL, DirectX)
  – Renders one triangle at a time
  – Z-buffer stores nearest pixels
  – Local lighting (per-vertex or per-pixel)

• Ray tracing
  – Renders one pixel at a time
  – [Kind of] Sorts the geometry per pixel
  – Global lighting equation (reflections, shadows)
Why ray tracing?

• Simple concept
• Higher quality rendering
  – Global lighting equation
    ‣ Accurate shadows, reflections, refraction, etc
    ‣ Soft shadows, more realistic materials and lighting
  – All computations done per pixel
    ‣ No interpolation
• Base for many advanced algorithms
  – Global illumination, e.g., path tracing, photon mapping
• A disadvantage: can take longer to render images!
Original goal: Add reflections
Side by side comparison (1980’s)

Images courtesy of Eric Haines
Newer example

[Image courtesy of ARTVPS www.artvps.com]
#include <stdlib.h>   // card > aek.ppm
#include <stdio.h>
#include <math.h>

typedef int i;typedef float f;struct v{f x,y,z;v operator+(v r){return v(x+r.x,y+r.y,z+r.z);}v operator*(f r){return v(x*r,y*r,z*r);}f operator%(v r){return x*r.x+y*r.y+z*r.z;}v(){}v operator^(v r){return v(y*r.z-z*r.y,z*r.x-x*r.z,x*r.y-y*r.x);}v(f a,f b,f c){x=a;y=b;z=c;}v operator!(){return*this*(1/sqrt(*this%*this));}};
i G[7]={247570,280596,280600,249748,18578,18577,23118,16,16};f R(){return(f)rand()/RAND_MAX;}i T(v o,v d,f&t,v&n){t=1e9;i m=0;f p=-o.z/d.z;if(.01<p)t=p,n=v(0,0,1),m=1;for(i k=19;k--;)for(i j=9;j--;)if(G[j]&1<<k){v p=o+v(-k,0,-j-4);f b=p%d,c=p%p-1,q=b*b-c;if(q>0){f s=-b-sqrt(q);if(s<t&&s>.01)t=s,n=!(p+d*t),m=2;}}return m;}v S(v o,v d){f t;v n;i m=T(v o,v d,f&t,v&n);if(!m)return v(.7,.6,1)*pow(1-d.z,4);v h=o+d*t,l=!(v(9+R(),9+R(),16)+h*-1),r=d+n*(n%d*-2);f b=l%n;if(b<0||T(h,l,t,n))b=0;f p=pow((1&*1600),99);if(m&1){h=h*.2;return((i)(ceil(h.x)+ceil(h.y))&1?v(3,1,1):v(3,3,3))*(b*.2+.1);}return v(p,p,p)+S(h,r)*.5;}i main(){printf("P6 512 512 255 ");v g=!v(-6,-16,0),a=!(v(0,0,1)^g)*.002,b=!(g^a)*.002,c=(a+b)*-256+g;for(i y=512;y--;){v p(13,13,13);for(i r=64;r--;){v t=a*(R()-5)*99+b*(R()-5)*99;p=S(v(17,16,8)+t,!(t*-1+(a*(R()+x)+b*(y+R())+c)*16))*.5+t;printf("%c%c%c",(i)p.x,(i)p.y,(i)p.z));}printf("%c%c%c",(i)p.x,(i)p.y,(i)p.z));}
To be physically correct, follow photons from light source

- Not what we do for a simple ray tracer
  - Though this is almost what we do for more advanced techniques (photon mapping)

- Not effective, **not** many rays will arrive at the eye
Instead: follow "photons" backwards from the eye

- Rationale: find photons that arrive through each pixel
- How does one find the visible object at a pixel?
- With intersection testing
  - Ray, \( r(t) = o + td \), against geometrical objects
    - \( r \) - ray, \( o \) - origin, \( d \) - direction, \( t \) - parameter
  - Use object that is closest to camera!
  - Valid intersections have \( t > 0 \)
  - \( t \) is a signed distance

For fast intersection testing, use a spatial data structure!
trace() and directIllumination()

• We now know how to find the visible object at a pixel
• How about finding the color of the pixel?
• Basic ray tracing is essentially only two functions with recursive calls
  – trace() and directIllumination()
• trace(): finds the first intersection with an object
  – Calls directIllumination(), and then trace()
• directionIllumination(): computes the direct lighting at that intersection point
Whitted-style ray tracer: A very simple type of ray tracing

- First call `trace()` to find first intersection
- `directIllumination()` computes direct illumination from light sources
- `trace()` then calls `trace()` for reflection and refraction directions
- `directIllumination()` applies the BRDF of the surface, and so on...
trace() in detail

Color trace(Ray R)
{
    bool hit;
    Intersection is;
    Color col, col_tmp;
    hit = intersectScene(R, is);
    if(hit)
    {
        col = directIllumination(is);
        if(is indicates reflective object)
        {
            col_tmp = trace(Reflected ray);
            col = weightTogether(col, col_tmp);
        }
        if(is indicates transmissive object)
        {
            col_tmp = trace(Refracted ray);
            col = weightTogether(col, col_tmp);
        }
    }
    else col = background_color;
    return col;
}

[recursion should also be terminated.. could be done after fixed depth]
directIllumination() computes direct lighting

• For now, we will use the simple standard lighting equation that we used so far
  – Diffuse + Specular

• Could use other models for the BRDF (Bi-directional Reflection Distribution Function)
  – More realistic materials
directIllumination() in detail

Color directIllumination(const Intersection &is)
{
    Color col(0,0,0);
    for each light L
    {
        // create shadow ray; is "is" in shadow
        // use intersectScene for this...
        if(not inShadow(L,is))
        {
            col+=DiffuseAndSpecular(L,is);
        }
    }
    return col;
}
In `directIllumination()`, we need a function `inShadow()`

- Compute distance from intersection point, \( p \), to light source: \( t_{\text{max}} \)

- Then use intersection testing:
  - Point is in shadow if \( 0 < t < t_{\text{max}} \) is true for at least one object
Who calls trace() to begin with?

• Someone need to spawn rays
  – One or more per pixel
  – A simple routine, `computeImage()`, computes rays, and calls `trace()` for each ray.

• Use camera parameters to compute rays
  – Resolution, FOV, camera direction & position & up
When does recursion stop?

• Recurse until ray does not hit anything?
  – Does not work for closed models

• One solution is to allow for max N levels of recursion
  – N=3 is often sufficient (sometimes 10 is sufficient)

• Another is to look at material parameters
  – E.g., if specular material color is (0,0,0), then the object is not reflective, and we don’t need to spawn a reflection ray
  – More systematic: send a weight, w, with recursion
    – Initially w=1, and after each bounce, w*=O.specular_color(); and so on.
    – Will give faster rendering, if we terminate recursion when weight is too small (say <0.01)

• We will return to this issue in future lectures...
Quick recap: Reflection

\[ r = 2(n \cdot l)n - l \]

Image courtesy of Illuminate Labs
Refraction

[Image courtesy of slartybartfast at ompf.org]
Refraction

[“Glasses” by Gilles Tran 2006]
Refraction: need a transmission vector, t

- n, i, t are unit vectors
- η₁ & η₂ are refraction indices
- c₁=cos(θ₁)=-n⋅i
- Decompose i into:
  - i₀=−c₁n, i_perp=i+c₁n
- t=sin(θ₂)m - cos(θ₂)n, where
- m=i_perp/‖i_perp‖=(i+c₁n)/sin(θ₁)
- Use Snell’s law: sin(θ₂)/sin(θ₁)=η₁/η₂ = η
- t = ηi + (ηc₁ - c₂) n, where c₂=cos(θ₂)
- Simplify: c₂=sqrt[ 1 – η²(1-c₁²) ]
Some refraction indices, $\eta$

- Measured with respect to vacuum
  - Air: 1.0003
  - Water: 1.33
  - Glass: around 1.45 – 1.65
  - Diamond: 2.42
  - Salt: 1.54

- Note 1: the refraction index varies with wavelength, but we often only use one index for all three color channels, RGB
  - Or use 3 slightly different \( \rightarrow \) diffraction!

- Note 2: can get Total Internal Reflection (TIR)
  - Means no transmission, only reflection
  - Occurs when \( c_2 \) is imaginary (see previous slide)
Diffraction
Next Lecture

• Tomorrow - Sampling and Object intersection

• Reading
  • Textbook : Chapter 1: Getting started, Chapter 3.1 & 3.3 : A Simple Ray Tracer