Ray Tracing III

Ulf Assarsson and
Tomas Akenine-Möller
Department of Computer Engineering
Chalmers University of Technology

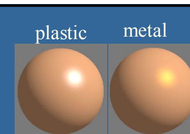
Today's lecture

- Types of material, and how light interacts
  - Glass, plastic... (dielectrics)
  - Metal (conductive)
- An overview of what else we can do with ray tracing
  - Good for your project…

Smooth Metal
(slät metall)

- Often used material, and well-understood in computer graphics
- We'll present a good approximation here
- Metals obey three "laws":
  - The highlight often has the same color as the diffuse
  - Law of reflection
  - The Fresnel equations:
    How much is reflected and how much is absorbed
  - Though, Fresnel effect for metals is subtle
  - Higher for dieletric materials

Smooth metals (2)



Types of highlights

- Highlight
- The law of reflection
- Since the metal is smooth, we can say that it reflects perfectly in the reflection direction
- Fresnel equations depend on
  - Incident angle
  - Index of refraction (chromium oxide: 2.7)
- Can compute polarized, and unpolarized values for the light (in CG, we ignore polarization, often)

Smooth metals (3)
Fresnel

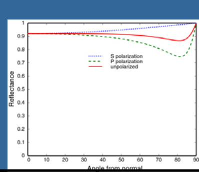
- F is describes the reflectance at a surface at various angles (n=index of refraction)

$$F = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left(1+\frac{[c(g+c)-1]^2}{[c(g-c)+1]^2}\right)$$

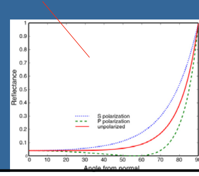
$$g=\sqrt{n^2+c^2-1}$$
$$c=\mathbf{v}\cdot\mathbf{h}$$

- Set refraction index=1.5, then you get:

Metals



Images courtesy of Steve Westin



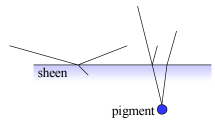
Dielectrics

An approximation to Fresnel
(by Schlick)

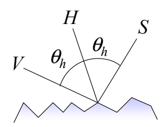
$$F \approx R_0 + (1-R_0)(1-\mathbf{v}\cdot\mathbf{h})^5$$

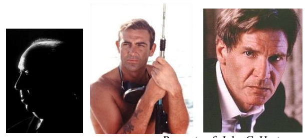
- $\mathbf{v}$ is the vector from the eye to the point on the surface
- $\mathbf{h}$ is the half vector, as usual. $\mathbf{h}=(\mathbf{l}+\mathbf{v})/||\mathbf{l}+\mathbf{v}||$
- R_0 is the reflectance when $\mathbf{v}\cdot\mathbf{h}=1$
- Works well in practice
- Use F for your reflection rays in shading:
  - F*`trace`(reflection_vector)

Fresnel Effect



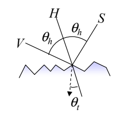
- % of light reflected increases as “halfway incidence angle” θ_h increases
  - Normal *N* of rough surface not well defined, so need *H* instead
  - S and V nearly constant across surface
  - Fresnel effect independent of N
  - Nearly constant across surface
- Affects color of reflected light
  - Normal incidence→surface color
  - Tangent incidence→light color
- Photographic trick: place bright light behind subject to create a Fresnel outline of features/profile





Property of John C. Hart

Fresnel Term



- Fresnel term *F* controls portion of light reflected v. portion refracted $(1 - F)$
- Physically accurate Fresnel function
  - Parameterized by index of refraction η

$$F_\eta(S,V) = \frac{1}{2}\left(\frac{\sin^2(\theta_h - \theta_t)}{\sin^2(\theta_h + \theta_t)} + \frac{\tan^2(\theta_h - \theta_t)}{\tan^2(\theta_h + \theta_t)}\right)$$

$$= \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left(1 + \frac{(c(g+c)-1)^2}{(c(g-c)+1)^2}\right)$$

  - where $c = S \cdot H$ and $g^2 = \eta^2 + c^2 - 1$
- Schlick’s hack

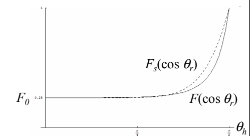
$$F_s(S,V) = F_0 + (1 - F_0)(1 - S \cdot H)^5$$

  - Parameter η implicit in F_0
- F_0 = Specular reflectance at normal incidence $(S \cdot H = 1)$

$\cos\theta_h = S \cdot H = V \cdot H$

But how do we find η?

$$F_0 = \left(\frac{\eta - 1}{\eta + 1}\right)^2 \qquad \eta = \frac{1 + \sqrt{F_0}}{1 - \sqrt{F_0}}$$



Property of John C. Hart

Smooth dielectric materials

- A dielectric is a transparent material
- Refracts light
- Filters light (due to impurities in material)
- Examples:
  - Glass
  - Diamond
  - Water
  - Air

Tomas Akenine-Möller © 2003

Smooth dielectric materials (2) Homegeneous impurities

- Light is attenuated with Beer’s law
- Loses intensity with: $dI = -C\,I\,dx$
- $I(s) = I(0)e^{-\ln(a)s}$
- Compute once for each RGB
- Also, use the Fresnel equations for these materials

Tomas Akenine-Möller © 2003

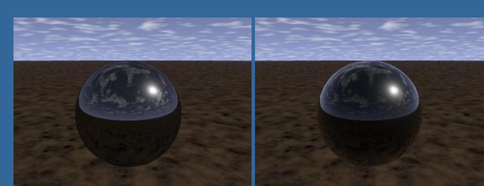
Beer’s law

The taller the glass, the darker the brew,
The less the amount of light that comes through

Tomas Akenine-Möller © 2003

Fresnel, example

- What does it look like
- A black dielectric sphere (glass or plastic)
  - in computer graphics, glass can be black
- Which is which?



Images courtesy of Steve Westin, Cornell University

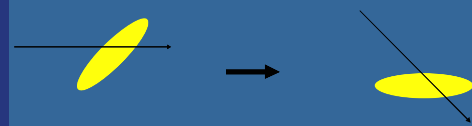
Tomas Akenine-Möller © 2003

What more can we do with ray tracing?

- A million things…
- Geometrical objects, etc.
- Optical effects
- Speed-up techniques
- Animation

Geometry

- Perfect for object-oriented programming
  - Makes it simple to add new geometrical objects
- Add a transform to each object
- The standard trick is not to apply the transform matrix to the object, but instead inverse-transform the ray

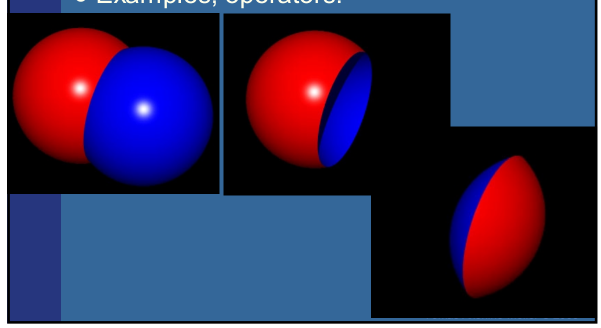


Geometry: Constructive Solid Geometry (CSG)

- Boolean operations on objects
  - Union
  - Subtraction
  - Xor
  - And
- Simple to implement
- Must find *all* intersections with a ray and an object
- Then do this for involved objects, and apply operators to found interval

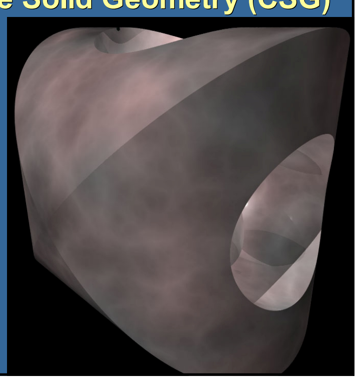
Geometry: Constructive Solid Geometry (CSG)

- Examples, operators:



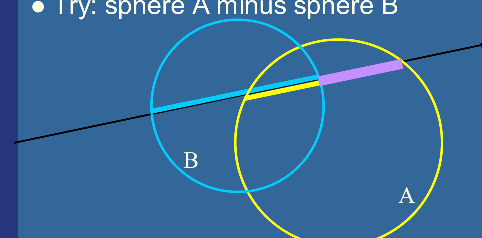
Geometry: Constructive Solid Geometry (CSG)

- Another example
- Done with 4 cylinders



Constructive Solid Geometry (CSG) How to implement

- Try: sphere A minus sphere B

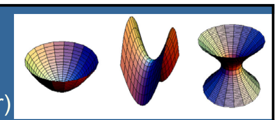


- In summary: find both entry and exit points on both spheres. Such two points on a sphere is an interval (1D). Apply the operator on these intervals

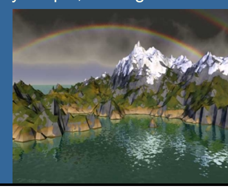
CSG

- Works on any geometrical object, as long as you can find all intersection point along a line
- Be careful with optimizations…
- And objects should be closed
  - Example: put caps on cylinder.

Geometry

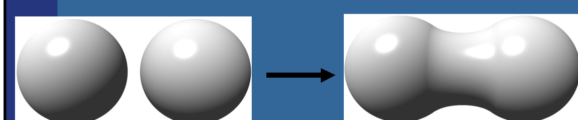


- Quadrics (2:a-gradsytor)
  - Cone, cylinder, paraboloids hyperboloids, etc.
- Higher order polynomial surfaces
  - Example: torus (badring), 4th degree
- Fractal landscapes
  - Pretty simple, fast algorithm exist

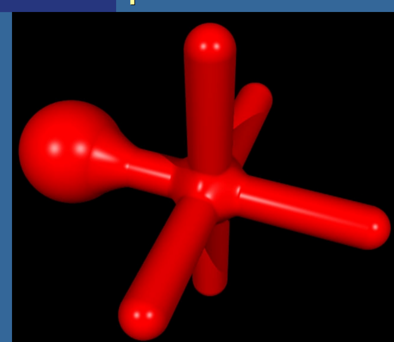


Geometry: Blobs

- A method for blending implicit surfaces (e.g., spheres, $x^2+y^2+z^2=1$)
- After blend, you get a higher order surface
- Need a numerical root finder

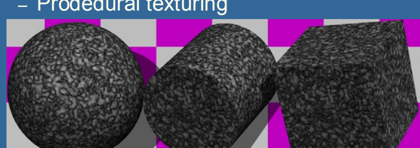


Blob example



Optics and more…

- You can add
  - Fog
  - Light fall off : $1/d^2$
  - Fresnel equations
  - Texture mapping
  - Prodedural texturing

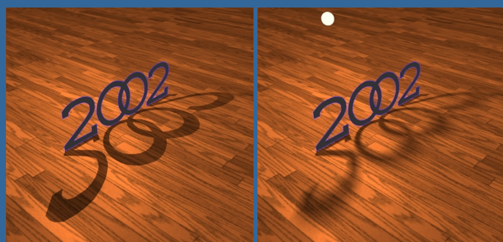


Optics

- Depth-of-field
  - Add more samples on a virtual camera lens

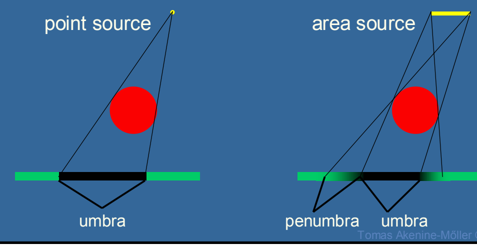
Soft shadows

- Higher level of realism
- Examples

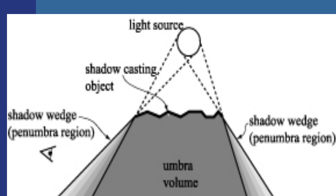


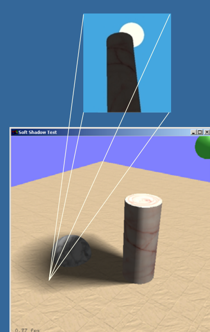
Soft shadows

- Why do they appear?
- Because light sources have an area or volume (seldom point lights)



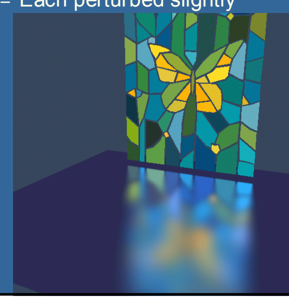
Example





Glossy (blurry) reflections

- Trace many reflection directions
  - Each perturbed slightly

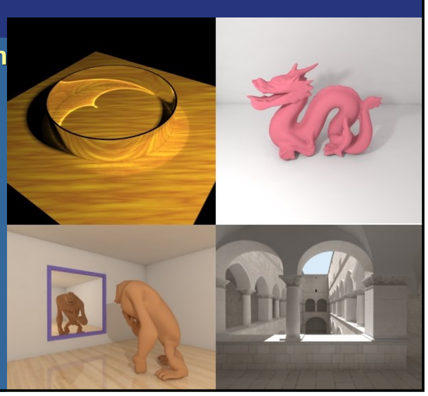


Do the same with transmission

Caustics

Photorealism

etc



Speed-up techniques

- Adaptive supersampling
- Try to exploit coherence
  - Frame-to-frame coherence (animation)
  - Pixel-by-pixel coherence (shadow cache)
  - Object-to-object (use spatial data structures)
  - Shading coherence?
  - And more…

Speed-up techniques

- For eye rays:
  - Render scene with OpenGL
  - Let each triangle or object have a unique color
  - Then read back color buffer
  - For each pixel, the color identifies the object
  - You need fast rendering in order to gain from this
- For eye rays (another technique):
  - Create a plane through a scanline of pixels
  - Objects that does not intersect plane can be skipped for this scanline

Animation

- Develop a little animation system, where you can save images
- Transforms are dealt with as treated previously
- Make transform a function of time: $\mathbf{M}(t)$
- Add motion blur (exposure time)

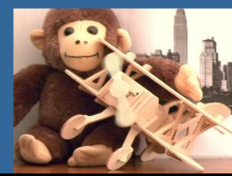


Image courtesy Brostow and Essa

Programmable shading

- Used successfully by Pixar (in RenderMan) and others for a long time now
- Instead computing lighting, etc by a fixed function (controlled by a few parameters)…
- …let a small program do it
- Gives the user incredible control and freedom!
- The small program is often called a ”shader”

Programmable shading

- A shader can modify
  - Lighting
  - Geometry (vertices)
  - Texturing
  - Lights
  - And whatever the programming interface allows
- Often done in a C-like language

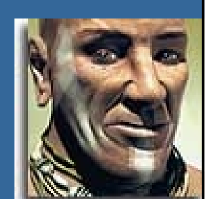
Ray tracing on GPU

See e.g.:

Ray tracing on Programmable Graphics Hardware

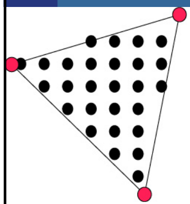
Tim Purcell, Buck, Mark Hanrahan, ACM Transactions on Graphics (SIGGRAPH), 2002

Ray Tracing with Graphics Cards using Fragment and Vertex Shaders



- The Graphics Processor Unit (GPU)
  - Performance ~doubles each 6 months
  - Optimized for highly parallel vertex- and fragment (pixel) shading
  - Efficient and fairly easy to use more transistors to increase performance/parallelism
- The CPU
  - Primary task – high speed for sequential code (well…)
  - Performance doubles each 18 months (Moore’s law)
  - A general processor that should be fast for everything – databases to signal processing…

What is vertex and fragment (pixel) shaders?



- Memory: Texture memory (read + write) typically 64-256 Mb
- Program size: 64-1024 instructions
- Instructions: mul, rcp, mov,dp, rsq, exp, log, cmp, jnz…
- 32 bits processors, usually SIMD

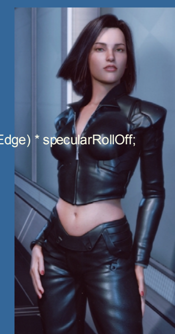
For each vertex, a vertex program (vertex shader) is executed

For each fragment (pixel) a fragment program (fragment shader) is executed

Fragment shader example

Leather

```
...
float specRollOffEdge =
      pow( (1.0 - VdH), 3.0);
float rollOff = specRollOffEdge + (1.0 - specRollOffEdge) * specularRollOff;
color C = LightColor * Ecc * Gain * rollOff; …
```



Hardware today:

- Geforce 6800 Ultra:
  - 6 vertex shaders
  - 16 Fragment shaders
  - 0.13µ, 400 MHz core (550MHz mem), 35.2 GByte/s, 3.6 Gpixels/s, 222 M transistors
- GeforceFX 5900–specs
  - 4 vertex shaders à 4 SIMD proc. Program size: Max 64 Kb
  - 8 Fragment shaders: Program size 1024 instr (no branches)
  - 0.13µ, 450 MHz core, 27.2 GByte/s, 3.6 Gpixels/s, 64 bits flops,
  - 130 M transistors
- ATI Radeon 9800 XT
  - 4 Vertex shaders: Program size 64 KB instr
  - 8 Pixel shaders: Program size "unlimited", 64 chuncks
  - Bandwidth 22.4 GB/s, 412 MHz core, 3.3 Gpixels/s
  - 110 M transistors, 0.15µ
- 3DLabs Wildcat VP (P10) – now old…
  - 16 vertex shaders à 1 processor
  - 64+64 SIMD processors for the fragment shaders
  - 75 M transistors, 20GB/s, 312 MHz DDR, 15µ
  - 200 SIMD proc, 200 Gflops,

GPU…

- The GPU is fast due to very deep pipelines
  - (up to 800 steps – Geforce3)

  And inherent parallelism at several layers:
  - Triangles – independent of each other
  - Vector instructions (SIMD) – 4 floats simultaneously
- Can be used for such versatile tasks such as ray tracing…

As will be shown next. Probably suitable for much more applications.

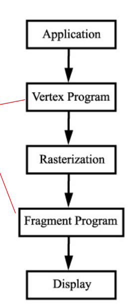
GPU…

- To keep GPU fast in future it should probably be a streaming processor rather than general-purpose
- 3 reasons:
  - Elemens independent -> several pipelines in parallel
  - High arithmetic intensity. Computation to memory bandwidh ratio is high
  - Prefetching can hide memory latency (FIFO-queues used).

Programmable Graphics Hardware

- The Current Programmable Graphics Pipeline

Used to be fixed function stages – now programmable



Limitations

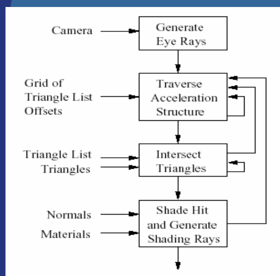
vertex and fragment program have simple, incomplete instruction set

Limited number of instructions

Number of textures is limited

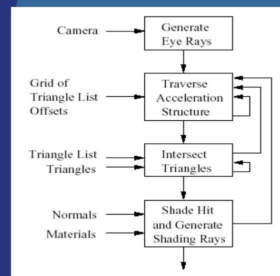
Fragment shaders: max 256 loop iterations, limited branching

Streaming Ray Tracer

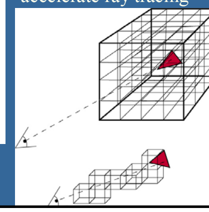


- Assume all scene geometry is represented as triangles
- Use a uniform grid to accelerate ray tracing

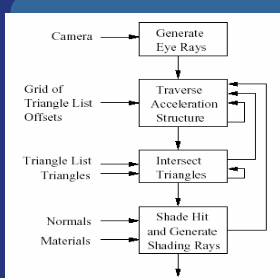
Streaming Ray Tracer



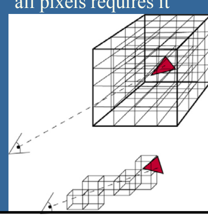
- Assume all scene geometry is represented as triangles
- Use a uniform grid to accelerate ray tracing



Streaming Ray Tracer

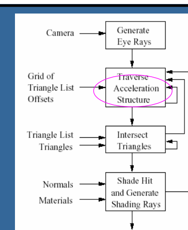


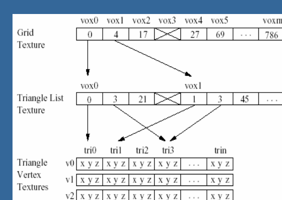
- Switch to intersection phase when >=20% of rays require it.
- Switch to shading when all pixels requires it

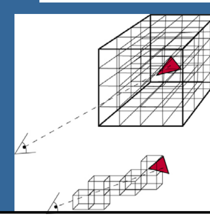


Ray tracing kernels

- Traverser
  - Output – ray-voxel pairs







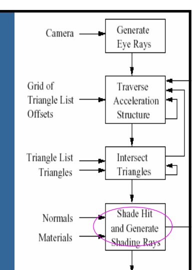
Ray trac

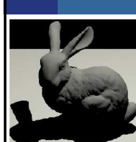
- Intersector
  - Using ray-v
  - Output – h
    - Point, nor

```
float4 IntersectTriangle( float3 ro, float3 rd, int list_pos, float4 h ){
    float tri_id = texture( list_pos, trilist );
    float3 v0 = texture( tri_id, v0 );
    float3 v1 = texture( tri_id, v1 );
    float3 v2 = texture( tri_id, v2 );
    float3 edge1 = v1 - v0;
    float3 edge2 = v2 - v0;
    float3 pvec = Cross( rd, edge2 );
    float det = Dot( edge1, pvec );
    float inv_det = 1/det;
    float3 tvec = ro - v0;
    float u = Dot( tvec, pvec ) * inv_det;
    float3 qvec = Cross( tvec, edge1 );
    float v = Dot( rd, qvec ) * inv_det;
    float t = Dot( edge2, qvec ) * inv_det;
    bool validhit = select( u >= 0.0f, true, false );
    validhit = select( v >= 0, validhit, false );
    validhit = select( u+v <= 1, validhit, false );
    validhit = select( t < h[0], validhit, false );
    validhit = select( t >= 0, validhit, false );
    t = select( validhit, t, h[0] );
    u = select( validhit, u, h[1] );
    v = select( validhit, v, h[2] );
    float id = select( validhit, tri_id, h[3] );
    return float4( {t, u, v, id} );
}
```

Ray tracing kernels

- Shader
  - Receive hit information from intersector.




Shadow Caster


Ray Caster


Whitted Ray Tracer


Path Tracer

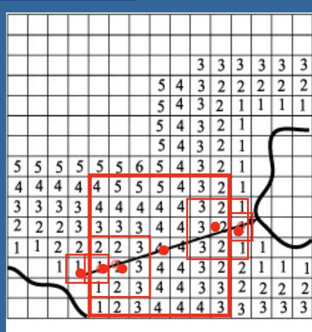
Results

- http://www.ce.chalmers.se/edu/proj/raygpu/

The future – expectations:

- Longer programs for fragment shaders
  - From 1024 instructions to 64 Kb(?) as for vertex shaders
- Faster shaders
  - More MHz, Higher bandwidth…
- More shaders simultaneously (higher parallelism)
- Full 64 bits accuracy
  - Currently just reached 32 bits with Radeon 9700
  - Geforce5 has (some) 64 bits flops
- More memory reads
  - Currently typically limited to 8-16 texture lookups/pass
- GPU = General programmable stream processor
- Non-graphical applications

Faster Grid Traversal using Proximity Clouds/Distance Fields



"Proximity Clouds – An Acceleration Technique for 3D Grid Traversal", Daniel Cohen and Zvi Sheffer

Demo

- Proximity clouds and SSE

End of ray tracing for a while
Let's talk about the project

- This is where you'll learn most of the interesting stuff
  - Dig into the references, come ask me, search internet
- 3 categories:
  - **A**: 3D game
  - **B**: Ray tracing (render a realistic image)
  - **C**: Anything of your choice (ask me if OK)
- See website -- lots of info

Project

- There are minimal requirements for A & B
- These are for one-person projects
- With more persons in project, the requirement becomes higher
- Per extra person: one more advanced algorithm
- Examples:
  - occlusion culling in game
  - Bump mapping in ray tracing

Project

- For the grade: I don't care about the models or textures. You can use only teapots for all I care
- I will grade the technical level of the implementation and how advanced it is

For those who want to…

- A competition in category A & B
  - The best game
  - The most realistic image (or movie)
- A jury will decide who wins
- Prize:
  - Honory
  - Glory
  - Diploma
  - Game
- Has nothing to do with grade – just for fun!