

# Ray Tracing II



Image: Nvidia CUDA ray tracer

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

# Lab 7

- YOU MUST START NOW
  - Or you will not be finished in time!

# Now, we have some basics in place

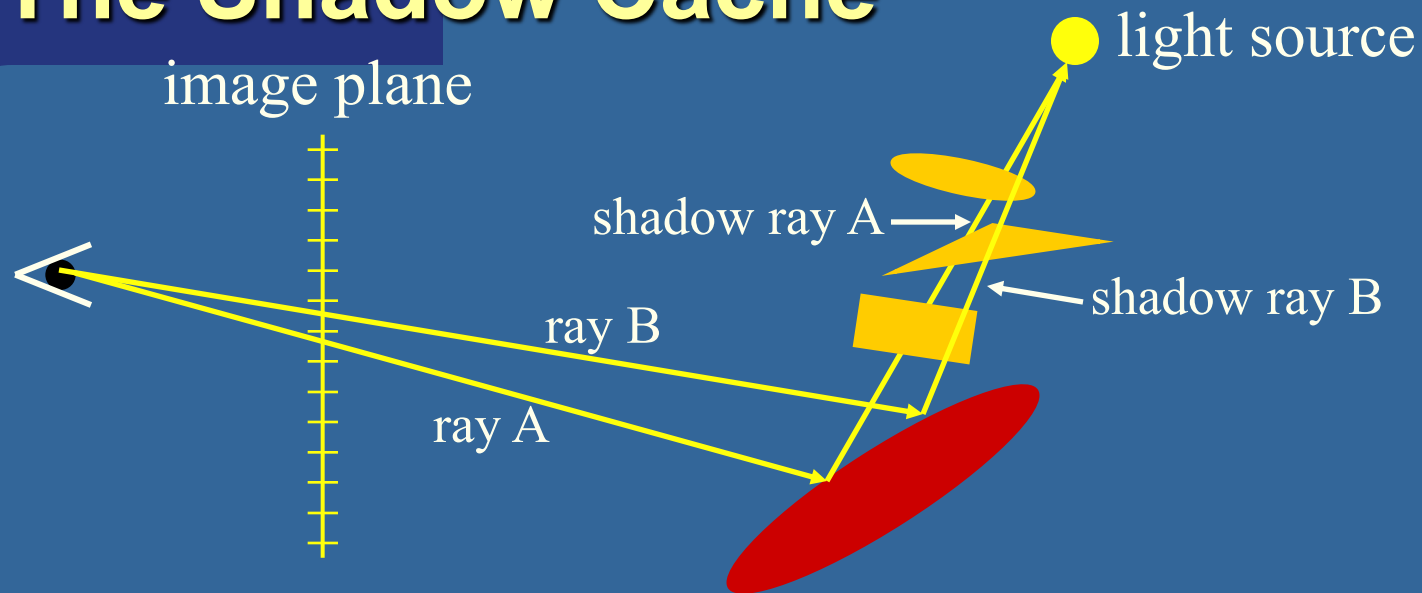
## Make ray tracing run faster!

- First, describe shadow cache
  - Simple idea
  - typically gives great speedup for shadow rays.
- Second, spatial data structures
  - Huge speed improvements
- Third, CPU/GPU-cache-aware coding and using parallelism:
  - Packet tracing,
    - You may want to check out Ingo Wald's interactive ray tracing work on the web
    - Intel's *Embree*
  - Shoot rays according to a Hilbert Curve
  - GPU ray tracing...

# Overview

- Shadow Cache
- Spatial data structures and ray traversal
  - Bounding volume hierarchies (BVHs)
  - BSP trees
  - Grids
- Materials
  - Fresnell Effect
  - Beer's Law
- Additional ray tracing techniques

# The Shadow Cache



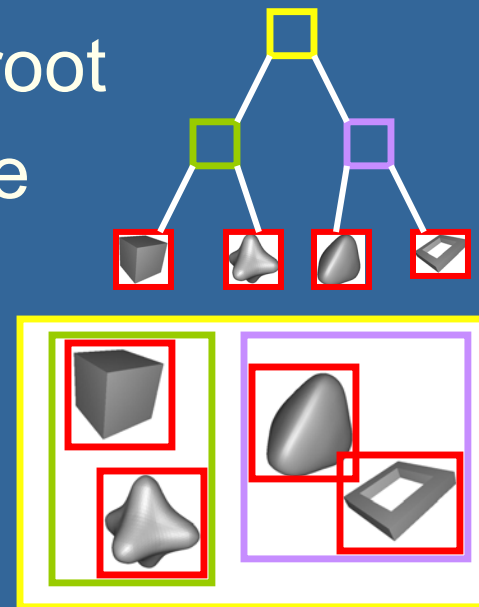
- It does not matter which object between the red ellipse and the light is detected
  - The point is in shadow if we find **one** object between
- Assume shadow ray A hits the triangle
  - store triangle in shadow cache
- For next ray B, start with testing the triangle in the shadow cache
- If high coherence, then we'll get many hits in cache

# Spatial data structures and Ray Tracing

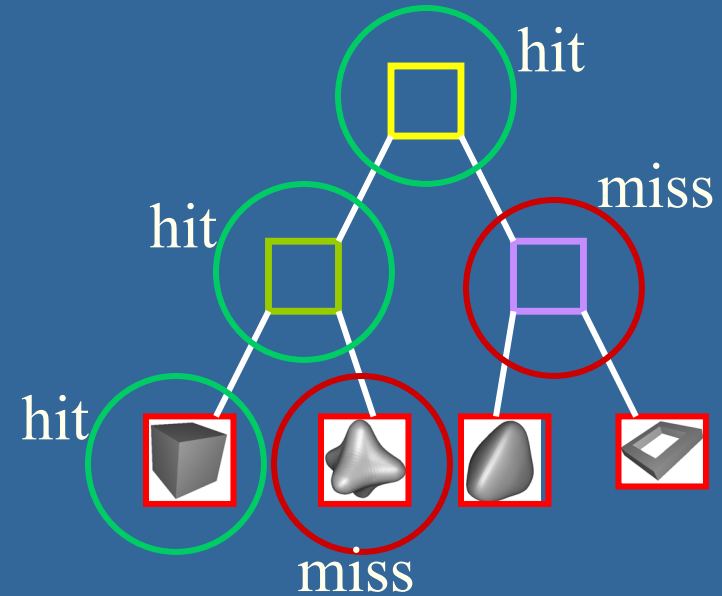
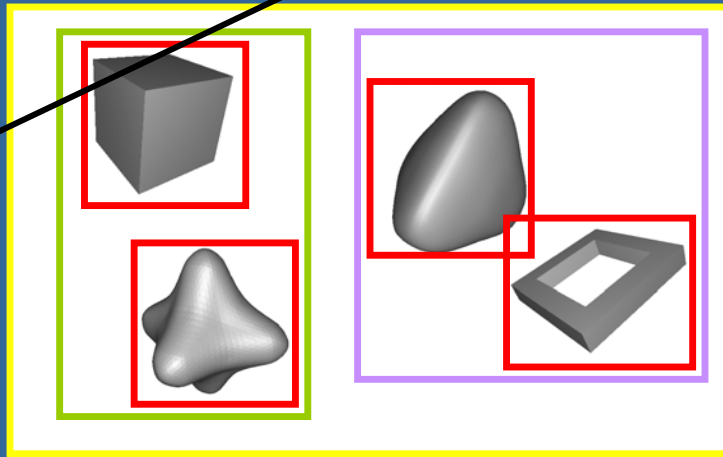
- Use these to get faster rendering
  - Ray tracing is often slow
  - Not useful for very few primitives (e.g., 2 spheres)
- The point:
  - You don't need to test for intersection between the ray and all objects in the scene
  - Rather, you test a small subset
  - Typically,  $O(\log n)$  instead of  $O(n)$  for each ray
- We will look at
  - Bounding volume hierarchies (BVHs)
  - BSP trees
  - Grids

# Bounding Volume Hierarchy (BVH)

- We'll use axis-aligned bounding boxes (AABBs) here
- The goal: find closest (positive) intersection between ray and all objects in the scene
- Simple: traverse the tree from the root
- If the ray intersects the AABB of the root, then continue to traverse the children
- If ray misses a child, stop traversal in that subtree



# Example: ray against BVH



- Without BVH, we would test each triangle of every object against the ray
- With BVH:
  - Only test the triangles of the cube against the ray
  - Plus some AABBs, but these are cheap



# Optimizations

- Always make a reference implementation
  - And save it for benchmarking!
- Benchmarking is key here:
  - Not all "optimizations" yield better performance
  - However, this definitely depends on what kind of scene you try to render
- Preprocessing is good
  - Use when possible

# BVH traversal optimizations

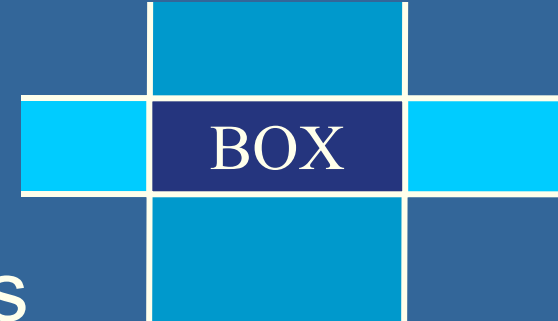
1. Use current closest intersection as an upper bound on how far the ray may "travel"
  - Example, if a ray hits a polygon at distance  $t_{\text{box}}$ , then we don't need to traverse a BV which is located farther than  $t_{\text{box}}$ .
2. Can also sort the BVs with respect to distance, and only open up if necessary
3. Shadow cache can be used for shadow rays
4. Shading cache: store what object the reflection and refraction ray hit last time, and for next reflection ray, test this object first → if hit, then a good upper bound on distance, and so we can discard everything beyond this distance

## A comment on previous slide

- Smits (ref to left) has noted that some of the presented optimizations were not useful any longer
- Reason: architectural changes in computers
- Bottleneck is often in memory accesses, not in floating point operations
- Lesson: use benchmarks to test if your optimizations gives better performance for a couple of scenes

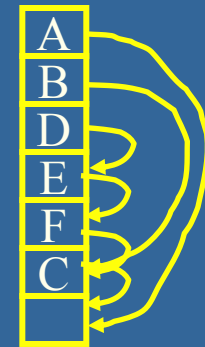
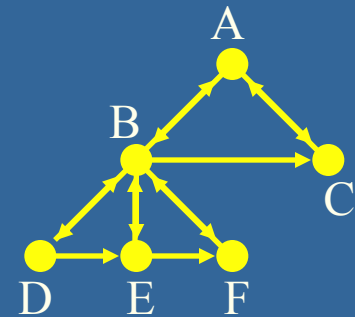
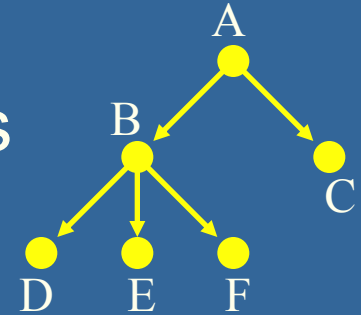
# AABB hierarchy optimization

- An AABB is the intersection of three slabs (2 in 2D)
- Observation: all boxes' slabs share the same plane normals
- Exploit this for faster AABB/ray intersection!
- AABB/ray needs to compute one division per  $x$ ,  $y$ , and  $z$ 
  - Precompute these once per ray, and use for entire AABB hierarchy



# BVH traversal... skip-pointer trees

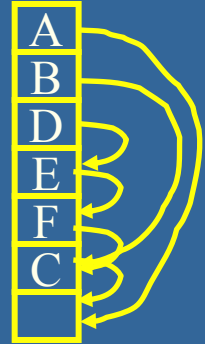
- Standard (depth-first) traversal is slow:
  - Involves recursion
  - And memory may be allocated once per node
- Left-child, right-sibling, parent pointers avoids recursion
  - Instead follow pointers
- Store these in a clever way, with skip pointers
  - Store nodes in depth-first order
  - A skip pointer points to the place where traversal shall continue given a miss



Good for single-threaded (non-parallel) code.

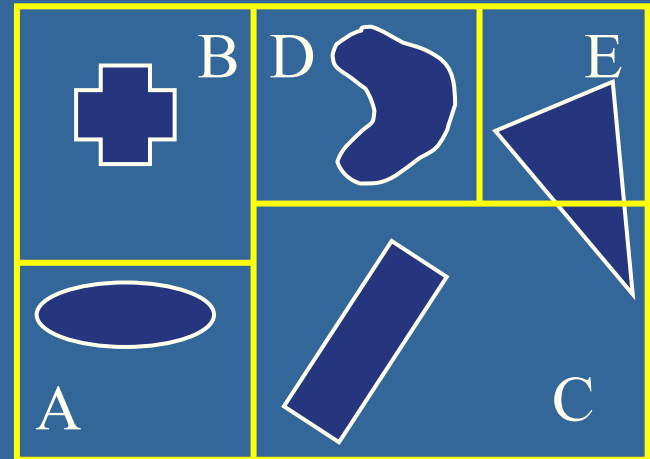
# BVH traversal... skip-pointer trees

- If no miss, continue in depth first order
- If nodes are allocated linear in memory, then we can expect many cache hits
- However, a node's children cannot be accessed in any order (child  $n$  can only be reached via child  $0..n-1$ ). Often not a big problem for AABB-hierarchies.
  - Because we typically need to test all children anyways.
  - However, for modern parallel CPUs/GPUs, you often want all of a node's children to be located adjacently in memory, so they can be efficiently fetched for testing in parallel.



# Axis-Aligned BSP trees

- (Not that different from BVHs with AABBs)
- An advantage here is that we traverse the space in a rough sorted order along the ray
- Pretty simple code as we will see



If we have a fixed order for the splitting dimension (e.g.  $x, y, z, x, y, z, \dots$  or  $z, x, y, z, x, y, \dots$  etc) this is called a kD-tree.

# Axis-aligned BSP tree against ray

```

RayTreeIntersect(Ray, Node, min, max)
{
    if(node==NULL) return no_intersection;
    if(node is leaf)
    {
        test all primitives in leaf, discard if not between min and max;
        return closest intersection point if any;
    }
    dist = signed distance along Ray to cutting plane of Node;
    near = child of Node that contains ray origin;
    far = child of Node that does not contain ray origin;
    if(dist>0 and dist<max) // interval intersects plane of Node
    {
        hit=RayTreeIntersect(Ray,near,min,dist); // test near side
        if(hit) return hit;
        return RayTreeIntersect(Ray,far,dist,max); // test far side
    }
    else if(dist>max or dist<0) // whole interval is on near side
        return RayTreeIntersect(Ray,near,min,max);
    else // whole interval is on far side
        return RayTreeIntersect(Ray,far,min,max);
}

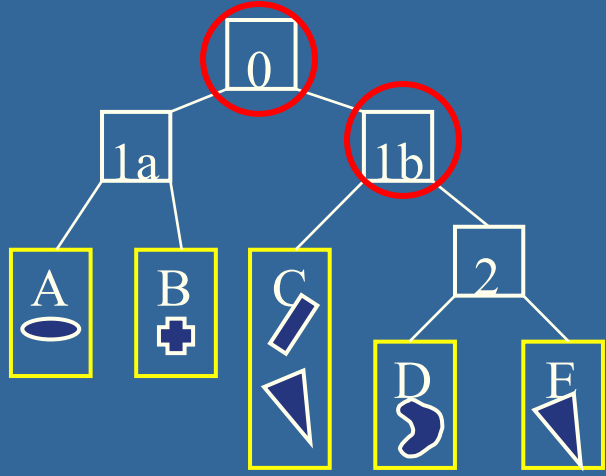
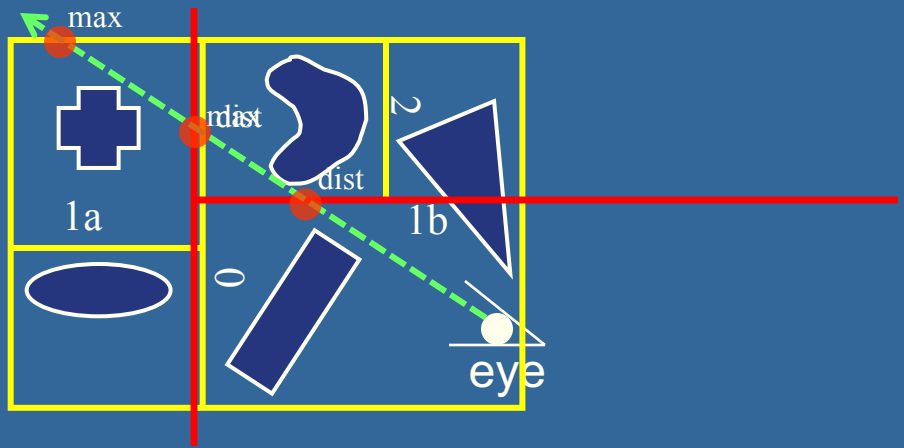
```



# AA-BSP Tree Traversal

- Test the planes against the ray
- Test recursively from root
- Continue on the "hither" side first

Kelvin Sung and Peter Shirley, "Ray Tracing with the BSP Tree", Graphics Gems III, pp. 271—274, 1992.



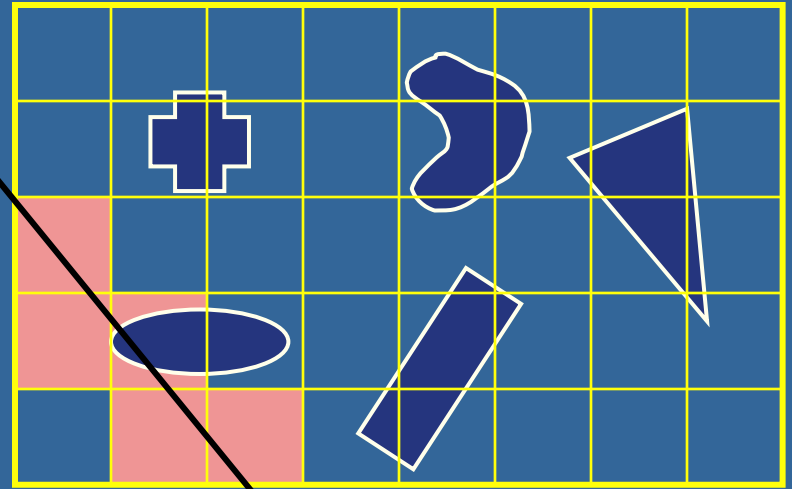
```

RayTreeIntersect(Ray, Node, min, max){
  if(node==NULL) return no_intersection;
  if(node is leaf)
    test all primitives in leaf, discard if not between min and max;
    return closest intersection point if any;
  dist = signed distance along Ray to cutting plane of Node;
  near = child of Node that contains ray origin;
  far = child of Node that does not contain ray origin;
  if(dist>0 and dist<max) // interval intersects plane of Node
    hit=RayTreeIntersect(Ray,near,min,dist); // test near side
    if(hit) return hit;
    return RayTreeIntersect(Ray,far,dist,max); // test far side
  else if(dist>max or dist<0) // whole interval is on near side
    return RayTreeIntersect(Ray,near,min,max);
  else return RayTreeIntersect(Ray,far,min,max); // whole interval is on far side
}

```

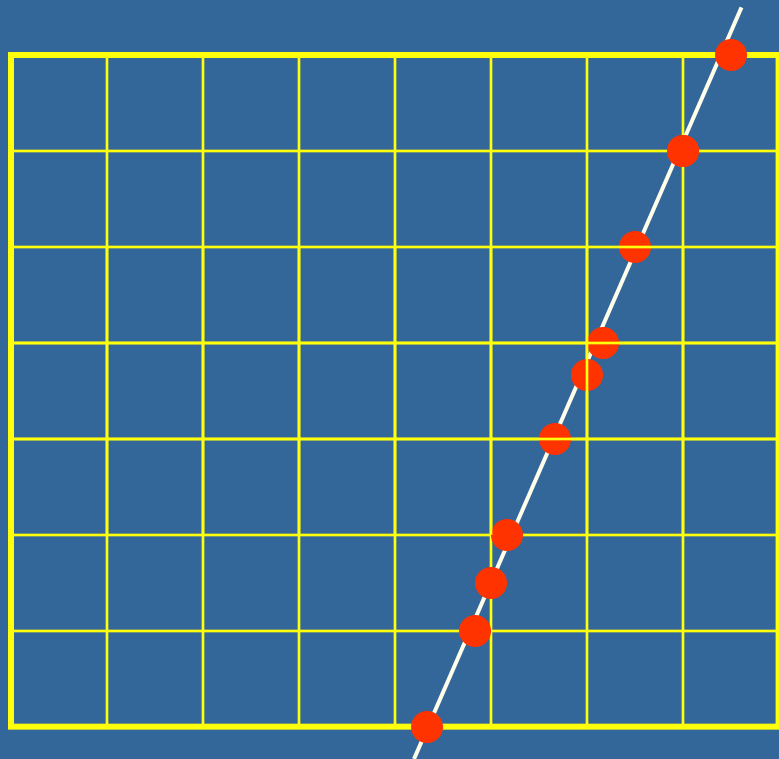
# Grids

- A large box is divided into a number of equally-sized boxes
- Each grid box stores pointers to all objects that are inside it
- During traversal, only the boxes that the ray intersect are visited, and objects inside these boxes are tested



# Grid Traversal Algorithm

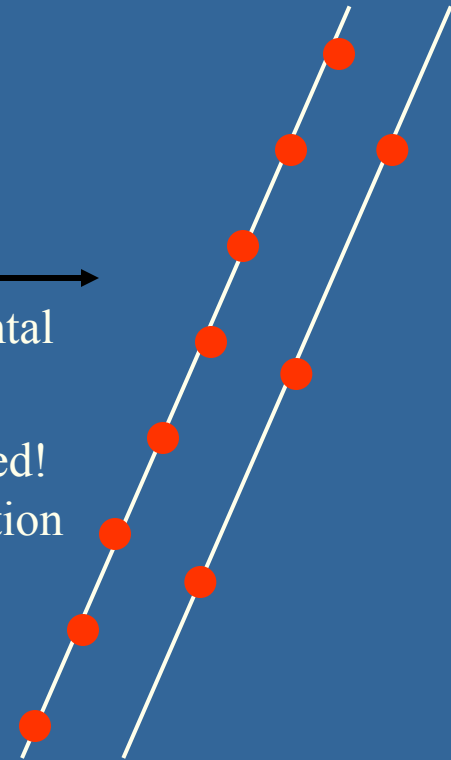
- A modified line generating algorithm can be used
  - Bresenham or DDA
- But easier to think in geometrical terms
  - Red circles mark where ray goes from one grid box to the next



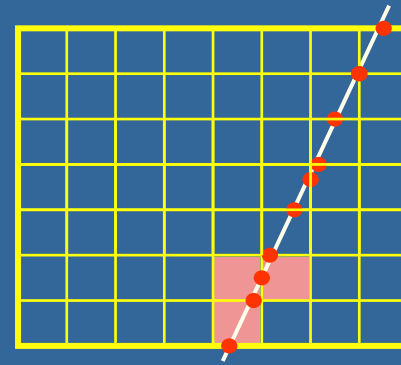
← Intersection points appear with irregular spacing

But, look first at only → intersection with horizontal lines, then vertical

These are regular spaced!  
Use that in implementation



# Traversal example

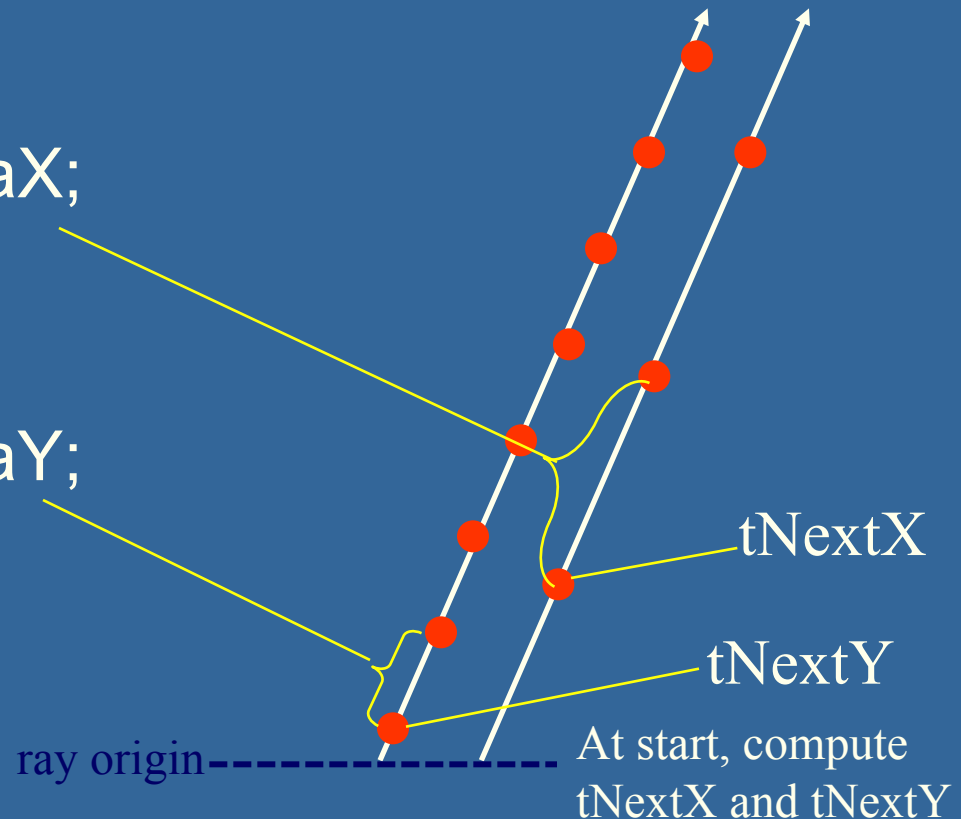


loop

```
if(tNextX < tNextY)
    tNextX += tDeltaX;
    X = X + stepX;
```

else

```
tNextY += tDeltaY;
Y = Y + stepY;
VisitVoxel(X, Y);
```



$tNextX$  = t-value at next x increment of cell index

$tNextY$  = t-value at next y increment of cell index

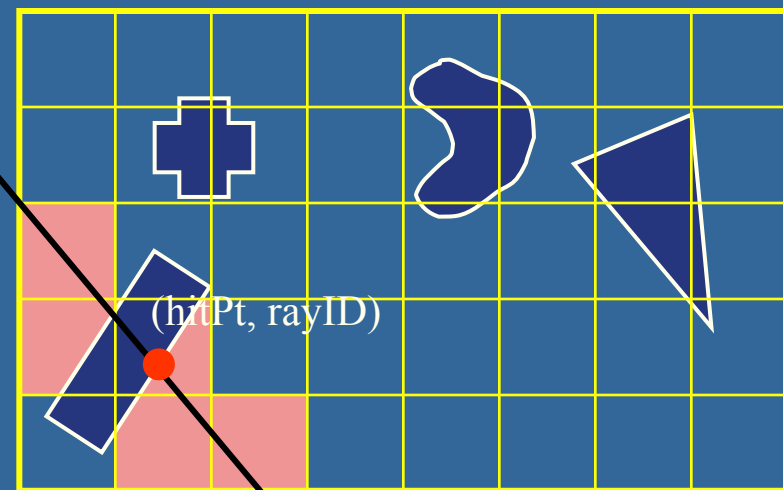
$stepX/Y = \pm 1$

## Grid Traversal (2)

- Easy to code up,
- Check out the following paper (for those who want to implement in their path tracer):
  - Amantides and Woo, "A Fast Voxel Traversal Algorithm for Ray Tracing", Proc. Eurographics '87, Amsterdam, The Netherlands, August 1987, pp 1-10.
- Available on course website

# Testing the same object more than once in grids

- If an object intersects more than one grid box, and a ray traverses these, then you may test the same object twice (waste of perf.)
- Solution: assign a unique rayID to each ray. For each tested object, store the {hitPt, rayID} with the object.
- If rayID of ray and object are the same, then we have already tested the object.



This is called  
mailboxing

So then just fetch the hitpoint, stored with the object

# Choose a good grid resolution

- Assume  $n$  objects in scene,  $g$  is grid resolution

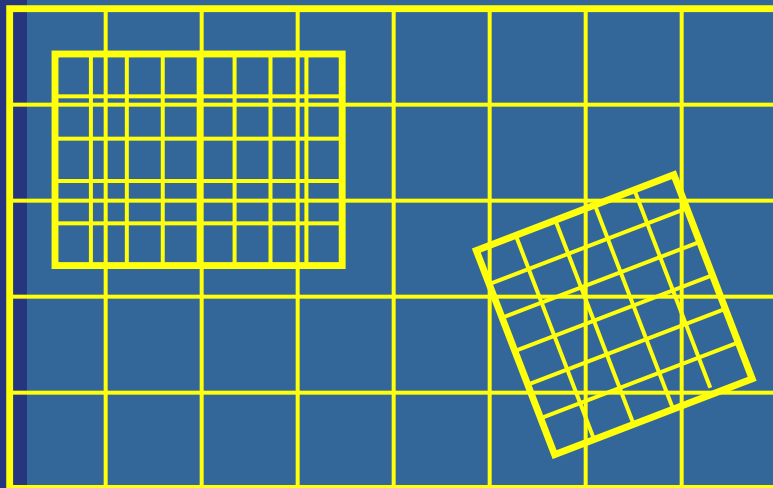
$$g = \sqrt[3]{n}$$

- Only good for cubes!

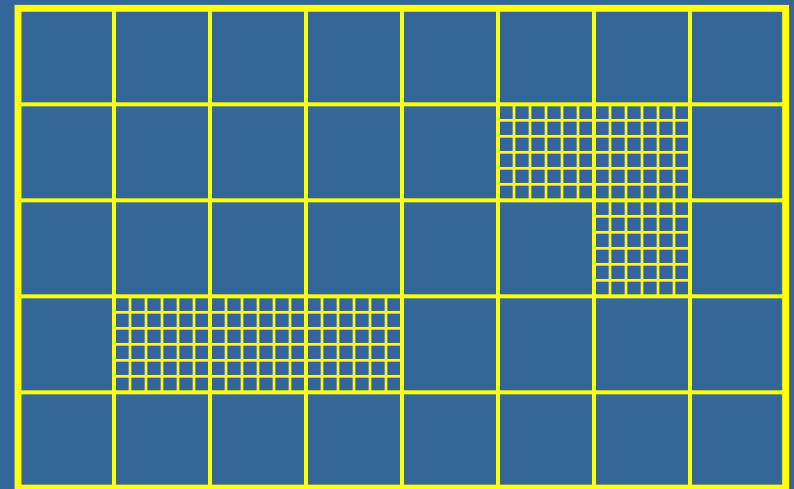
- Better to have different number of grid boxes per side
- Let the number of grid boxes per side be proportional to the length of the box side
- See Klimaszewski and Sederberg, in IEEE Computer Graphics & Applications, Jan-Feb, 1997, pp. 42—51.

# Hierarchical and Recursive Grids

- We often use hierarchies in CG, so we can do that now as well
- When a grid box (voxel) contains many primitives, introduce a smaller grid in that grid box



Hierarchical grid



Recursive grid

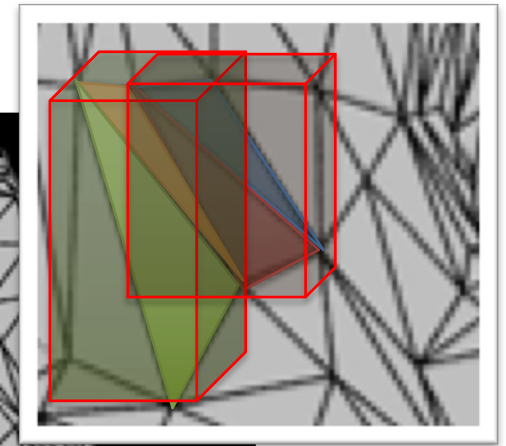
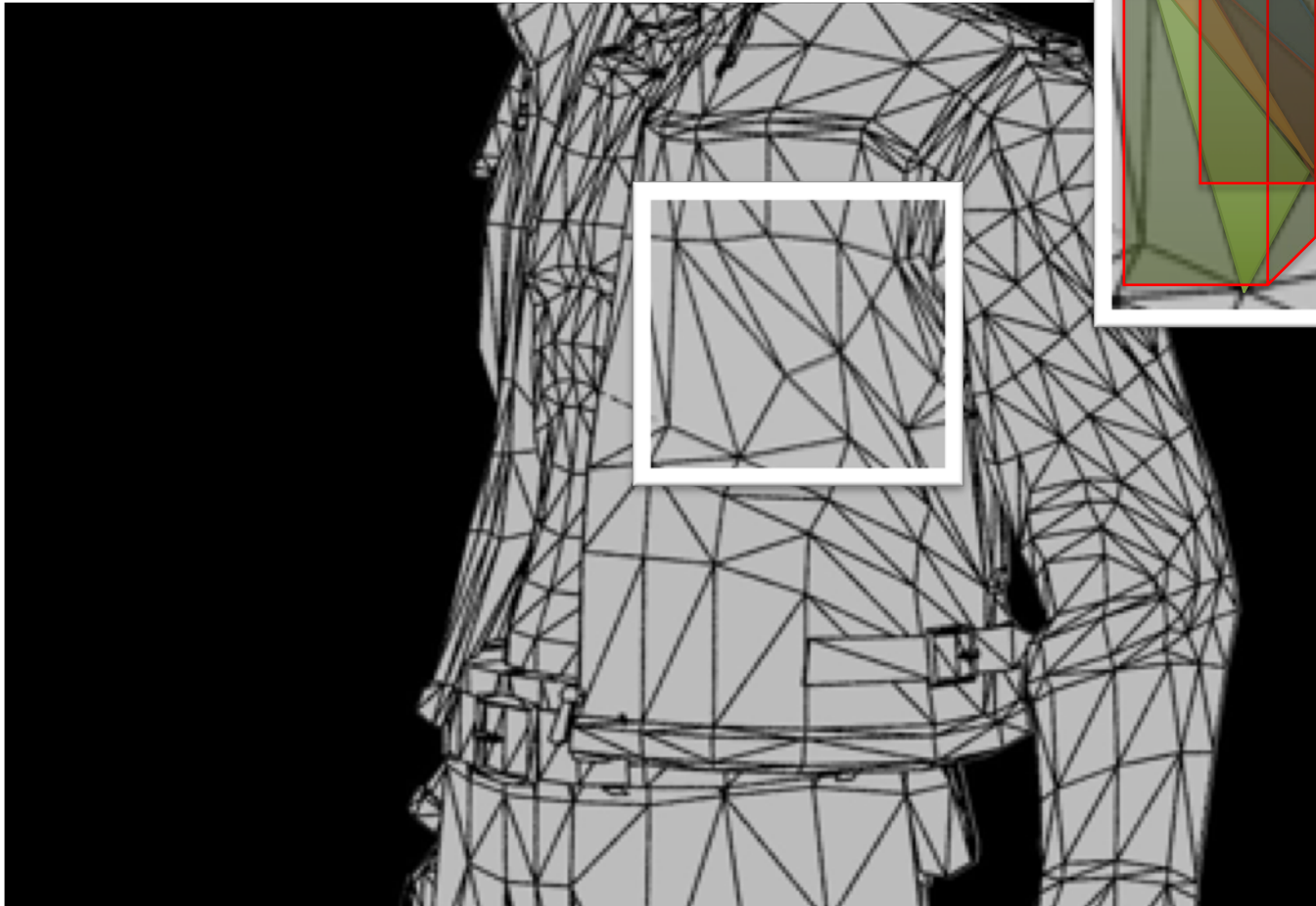


# Which spatial data structure is best?

- Depends on implementation, the type of scene, how complex shading, etc, etc.
- Make timings on several different algorithms and choose what works best for you
- Kd-trees:
  - Fastest to traverse, little memory, slow to build
- AABB-hierarchies:
  - Fast to build, slower to traverse (not automatically in order along ray. Fast to update for moving rigid objects.
  - Split-BVHs are currently very popular.
- Grids
  - Fast to build, middle fast to traverse, typically needs to be hierarchial/recursive
  - Hierarchical grids can be fast to update for moving rigid objects.

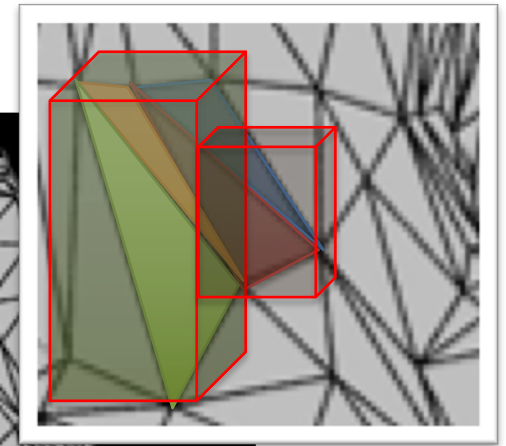
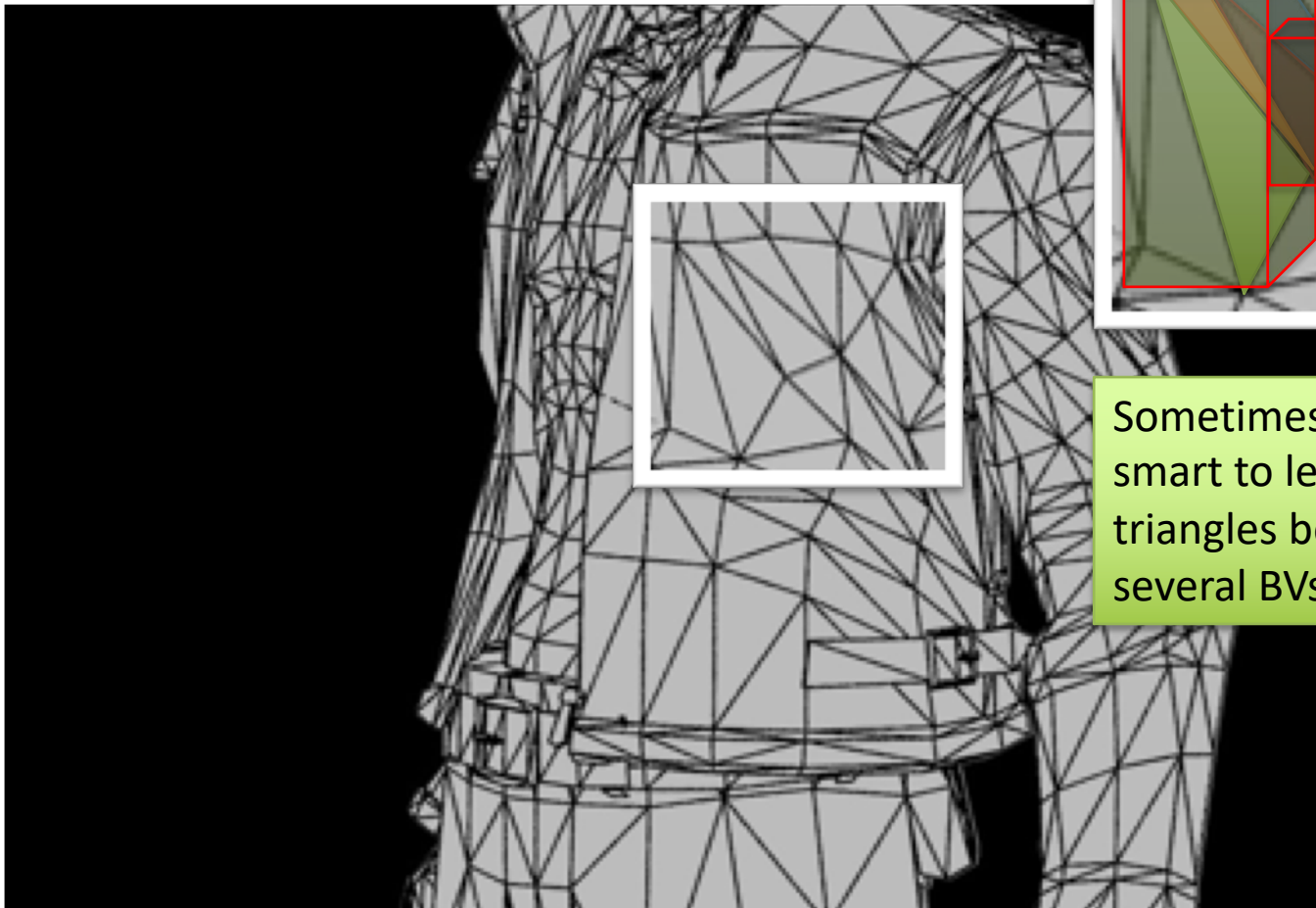
# Split Bounding Volume Hierarchies

- SBVHs



# Split Bounding Volume Hierarchies

- SBVHs

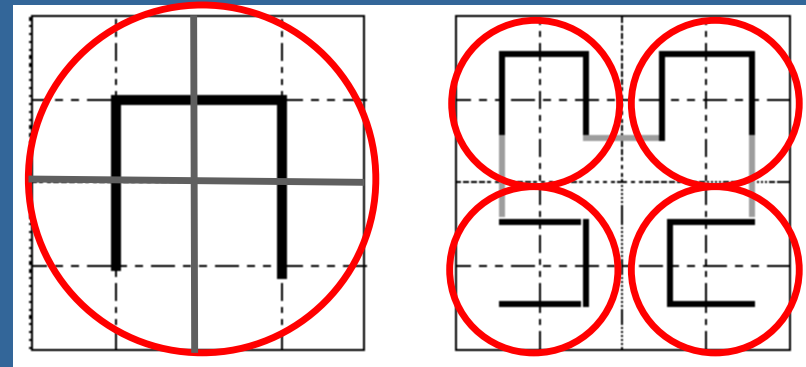


Sometimes it is smart to let triangles be part of several BVs.

# Cache awareness

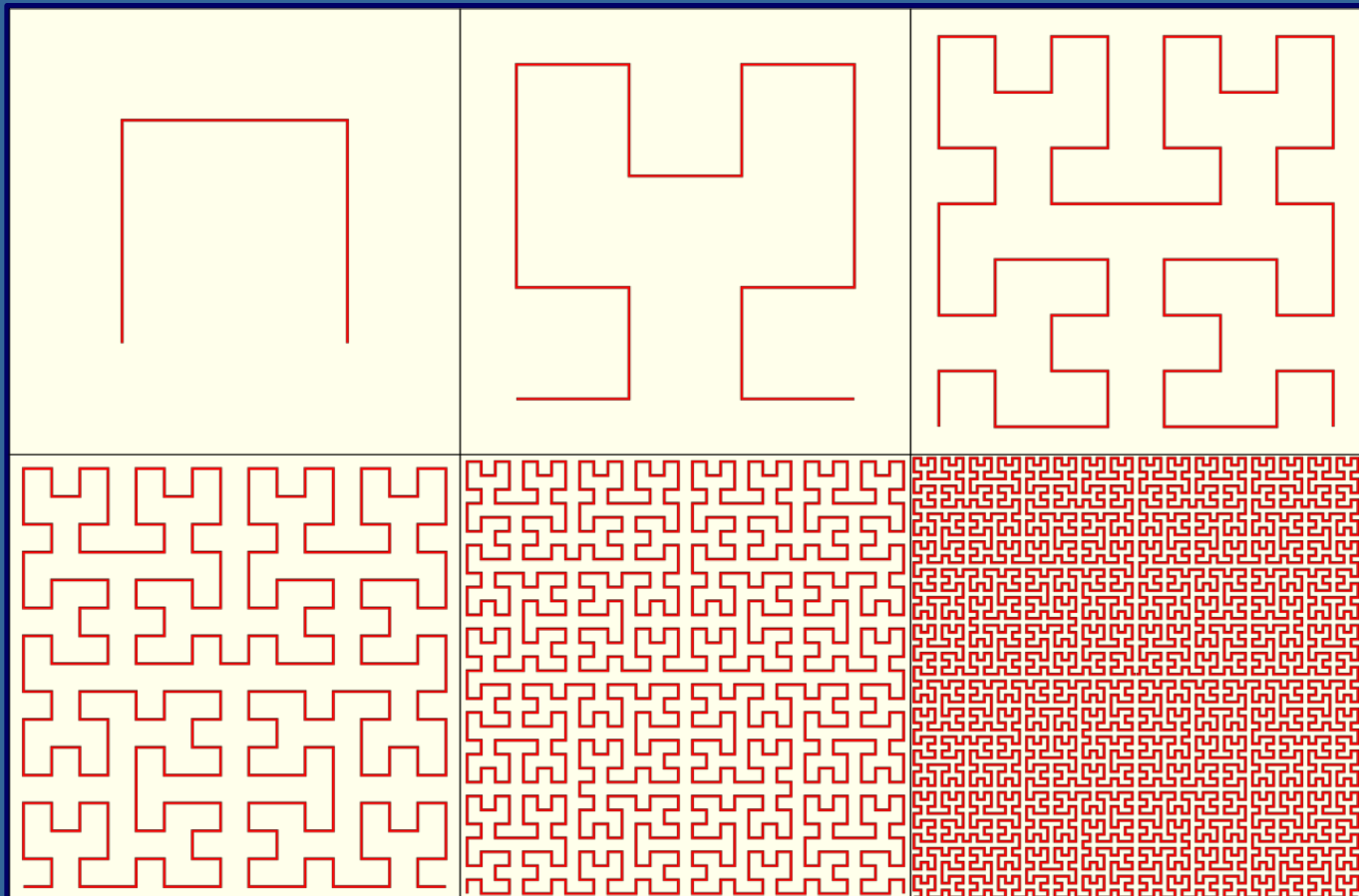
- To maximize cache locality, you can utilize that the next ray likely will access roughly the same memory locations since it will traverse roughly the same part of the tree and geometrical objects.
  - To maximize spatial locality, shoot the rays according to a Hilbert curve, instead of sequentially scanline by scanline.

# Hilbert Curve



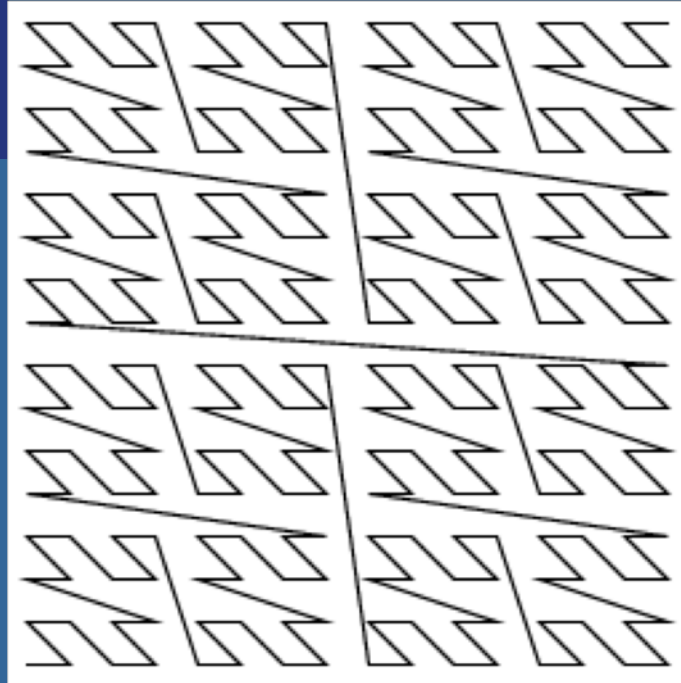
2x2 pixels

4x4 pixels



# Z-curve

or “Morton order”



For primary ray  $n$ :

- the screen-x coord is every 2<sup>nd</sup> bit of  $n$ , starting with bit 0.
- the screen-y coord is every 2<sup>nd</sup> bit of  $n$ , starting with bit 1.

Shoot rays  $r = 0..w*h$

Assume ray is the  $n$ :th ray, and  $n$ 's binary value is:

$$n = \dots y_3 x_3 y_2 x_2 y_1 x_1 y_0 x_0$$

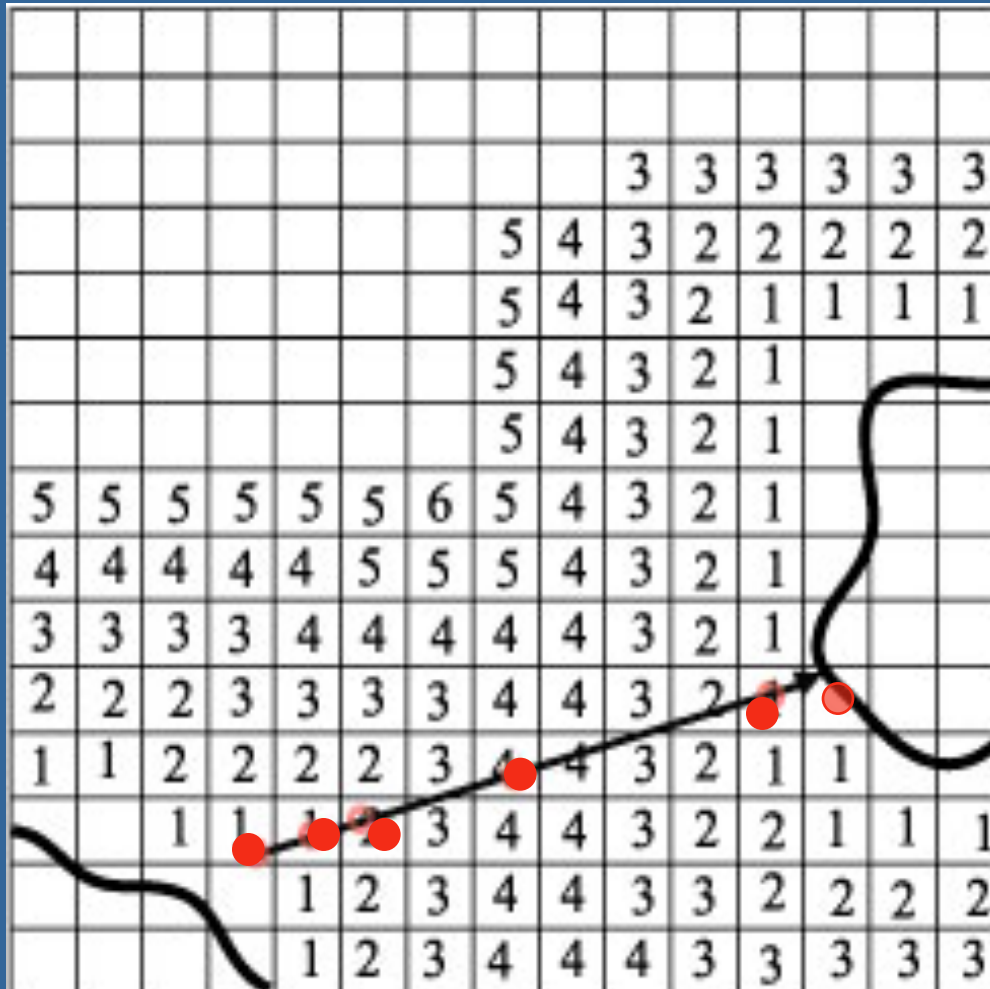
$$\text{e.g., } n = \quad 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 = 214$$

Then, the ray's  $x$  and  $y$  coordinates are:

$$x\_coord = \dots x_3 x_2 x_1 x_0 = 1 \ 1 \ 1 \ 0 = 14$$

$$y\_coord = \dots y_3 y_2 y_1 y_0 = 1 \ 0 \ 0 \ 1 = 9$$

# Faster Grid Traversal using Proximity Clouds/Distance Fields



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

**Demo**  
using SSE

# MATERIALS

- Types of material, and how light interacts
  - Glass, plastic... (dielectrics)
  - Metal (conductive)



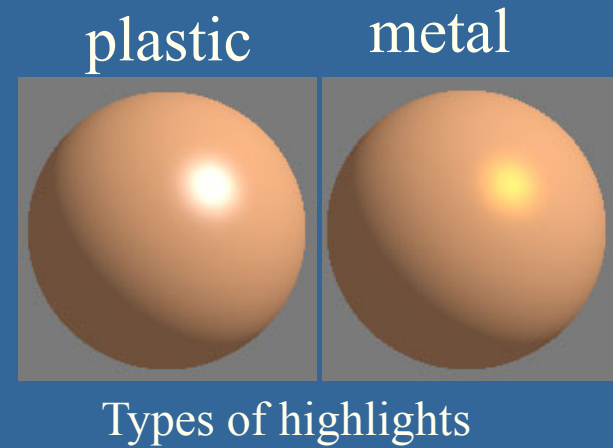
# Smooth Metal

(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 dielectric materials

## Smooth metals (2)

- Highlight
- The law of reflection
- If 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)





- At some places, the reflection is saturated (almost white), but mostly, it is clearly modulated by the copper color
  - Plastic adds the pure reflection color
  - Metal adds a modulated reflection color

# Fresnel

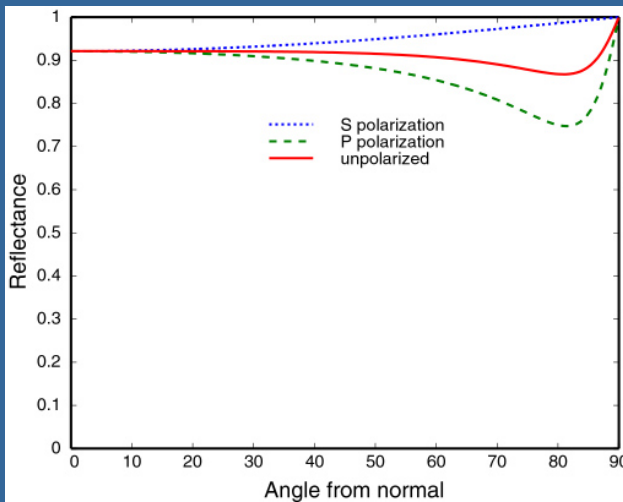
- $F$  describes the reflectance at a surface at various angles ( $n$ =index of refraction)

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

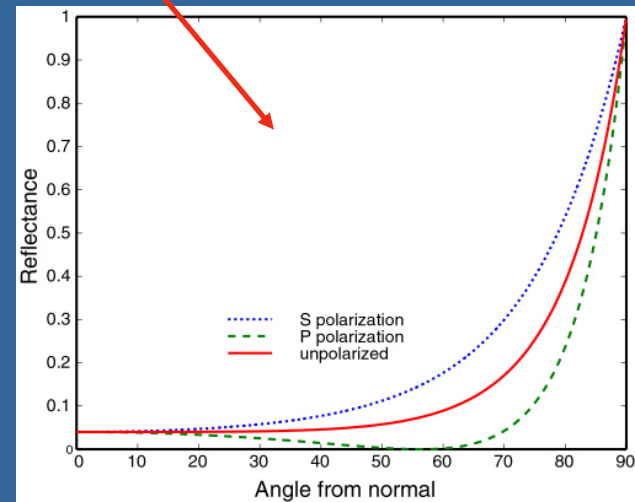
$$g = \sqrt{n^2 + c^2 - 1}$$

$$c = \mathbf{v} \cdot \mathbf{n} = \text{angle.}$$

- Set refraction index  $n=1.5$ , then you get:



Images courtesy  
of Steve Westin



Dielectric,  $n=1.5$

# An approximation to Fresnel

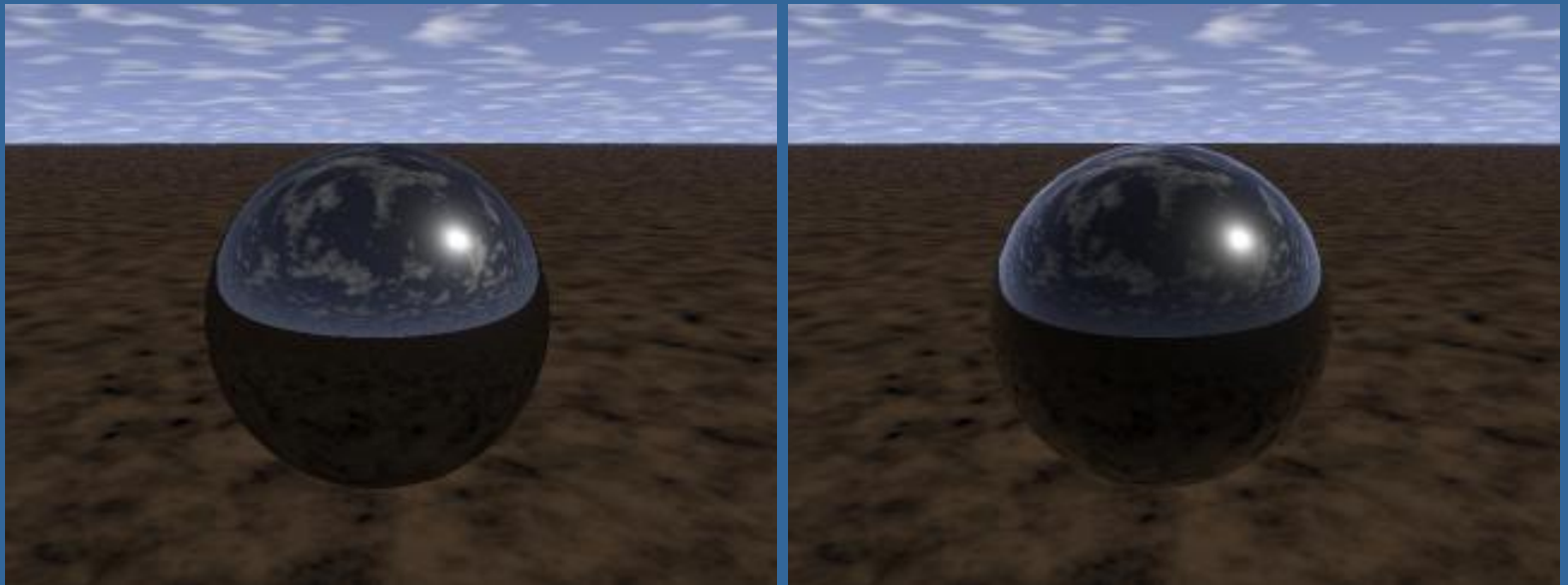
(by Schlick)

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

- $\mathbf{v}$  is the vector from the eye to the point on the surface
- $\mathbf{n}$  is the surface normal
- $R_0$  is the reflectance when  $\mathbf{v} \cdot \mathbf{n} = 1$
- Works well in practice
  
- Use  $F$  for your reflection rays in shading:
  - $F * \text{trace}(\text{reflection\_vector})$
  - Can be used for rasterization too (e.g. when applying result from cubemaps)

# Fresnel, example

- What does it look like
- A black dielectric sphere (glass or plastic)
  - in computer graphics, glass can be black
- Which has the Fresnel effect?



Images courtesy of Steve Westin, Cornell University

# Smooth dielectric materials

- A dielectric is a transparent material
- Refracts light
- Filters light (due to impurities in material)
- Examples (index of refraction):
  - Glass = 1.5
  - Plastic =  $\sim 1.5$
  - Diamond = 2.4
  - Water = 1.33
  - Air = 1.0

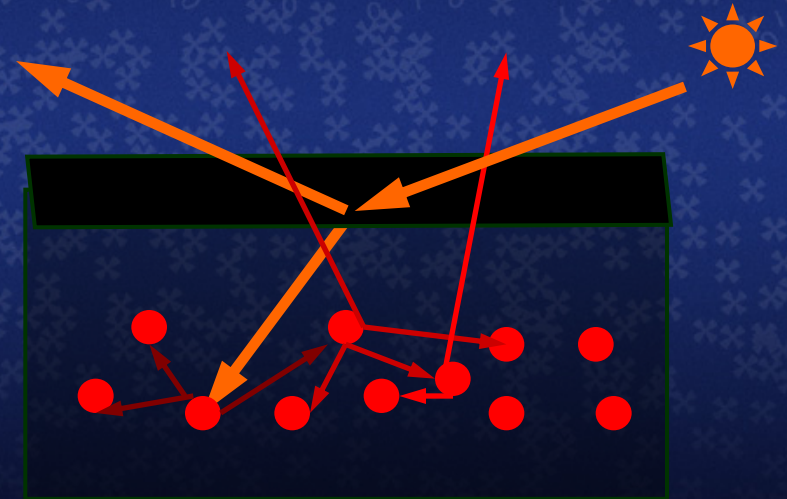
**The following 17 slides are from an excellent presentation at Microsoft Meltdown 2004,  
"Advanced Real-Time Reflectance",  
by Dan Baker, Naty Hoffman and Peter-Pike Sloan**

The corresponding paper can be found here:  
[www.gdconf.com/archives/2004/hoffman\\_naty.pdf](http://www.gdconf.com/archives/2004/hoffman_naty.pdf)



# Smooth Dielectric

- Low reflectance (water, glass, plastic, etc. ~5%)
- Refracted light continues inside the material, being scattered by impurities until it is absorbed or re-exits the surface



From Advanced Real-Time Reflectance, by Dan Baker, Naty Hoffman & Peter-Pike Sloan

# Diffuse / Specular Tradeoff



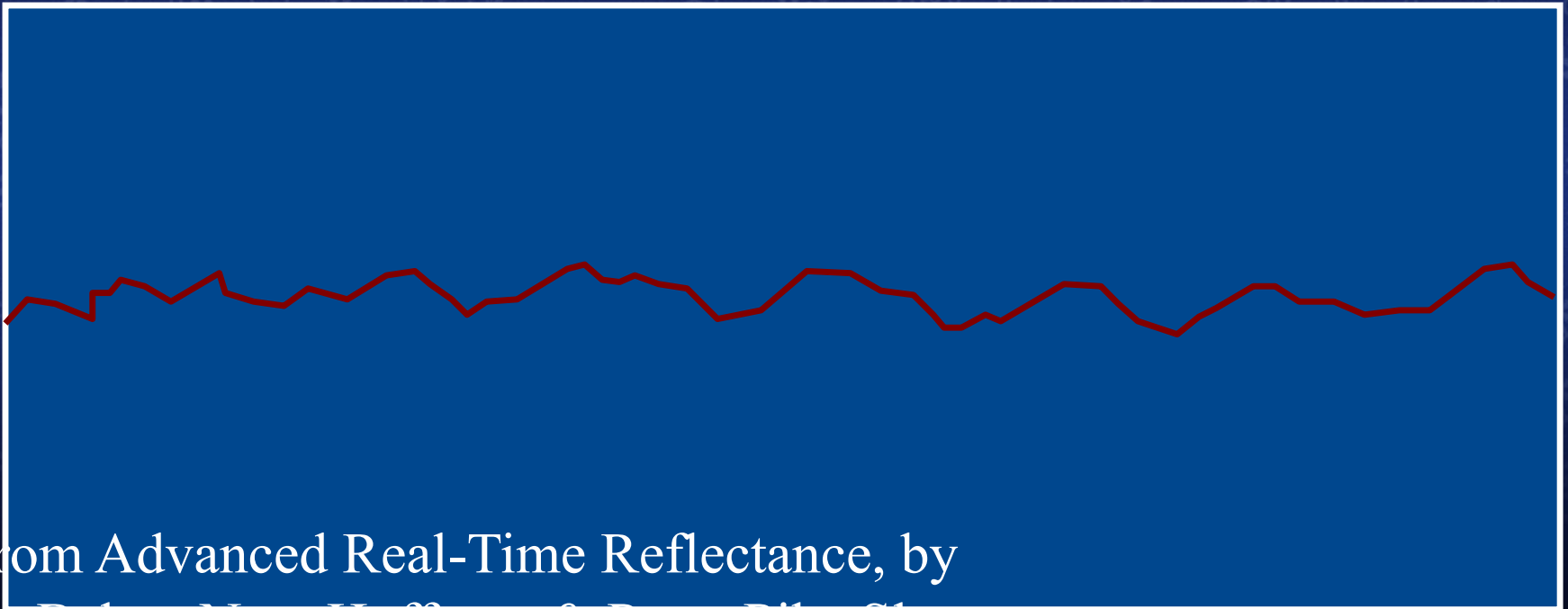
From Advanced Real-Time Reflectance, by  
Dan Baker, Naty Hoffman & Peter-Pike Sloan

# Semi-Rough (Glossy)

- **Most surfaces are not flat at all scales**

# Semi-Rough (Glossy)

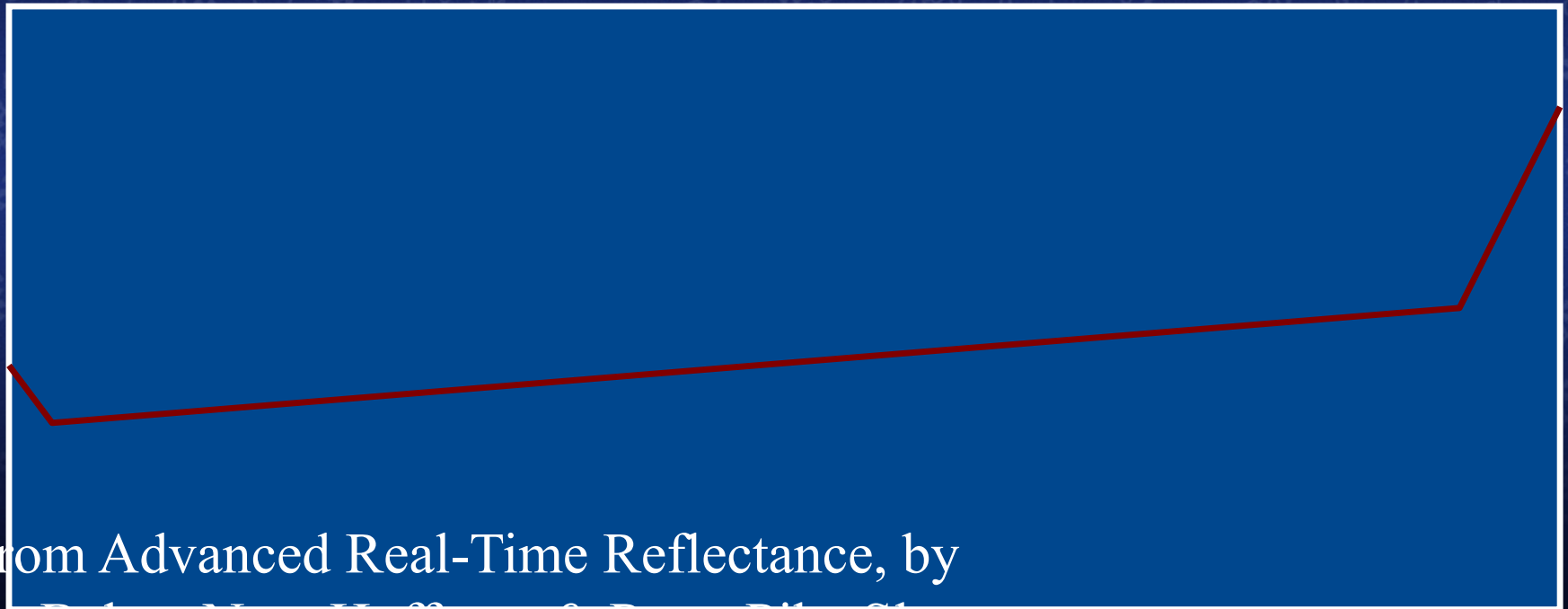
- **Most surfaces are not flat at all scales**
- **Many surfaces which appear flat at visible scales have complex *microscale* structure**



From Advanced Real-Time Reflectance, by  
Dan Baker, Naty Hoffman & Peter-Pike Sloan

# Semi-Rough (Glossy)

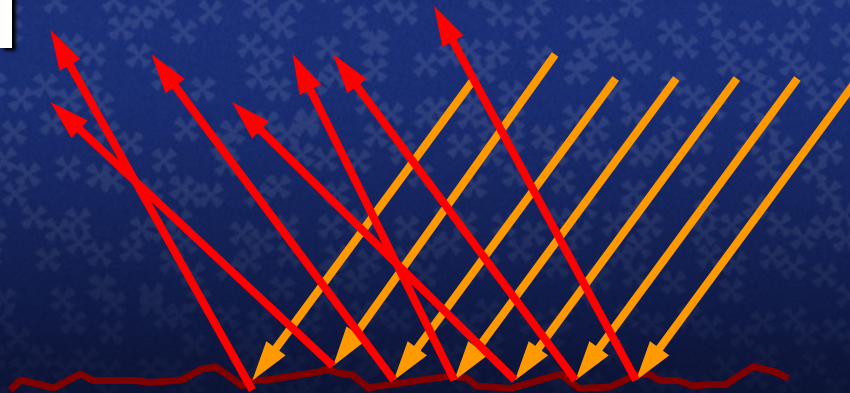
- **Most surfaces are not flat at all scales**
  - Many surfaces which appear flat at visible scales have complex *microscale* structure
  - At the smallest scale we can often treat the surface as flat again



From Advanced Real-Time Reflectance, by  
Dan Baker, Naty Hoffman & Peter-Pike Sloan

# Semi-Rough (Glossy)

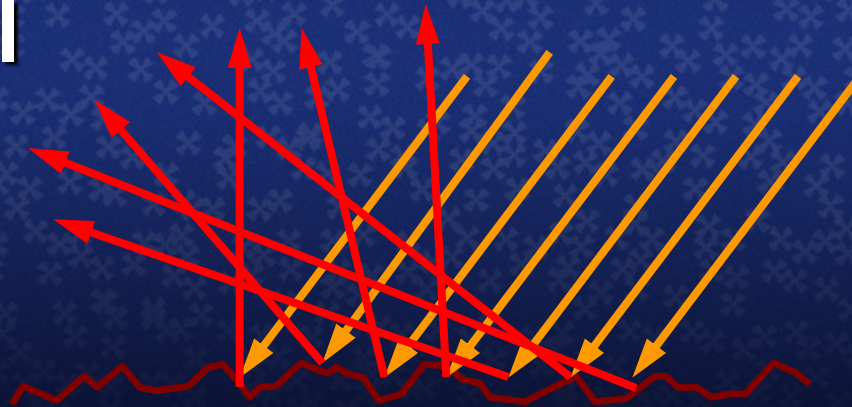
- A surface patch contains micro-facets with continuously distributed normals
- Light reflects off facets, 'spreads out'
- In 'semi-rough' surfaces distribution of micro-normals biased to macro-normal



From Advanced Real-Time Reflectance, by  
Dan Baker, Naty Hoffman & Peter-Pike Sloan

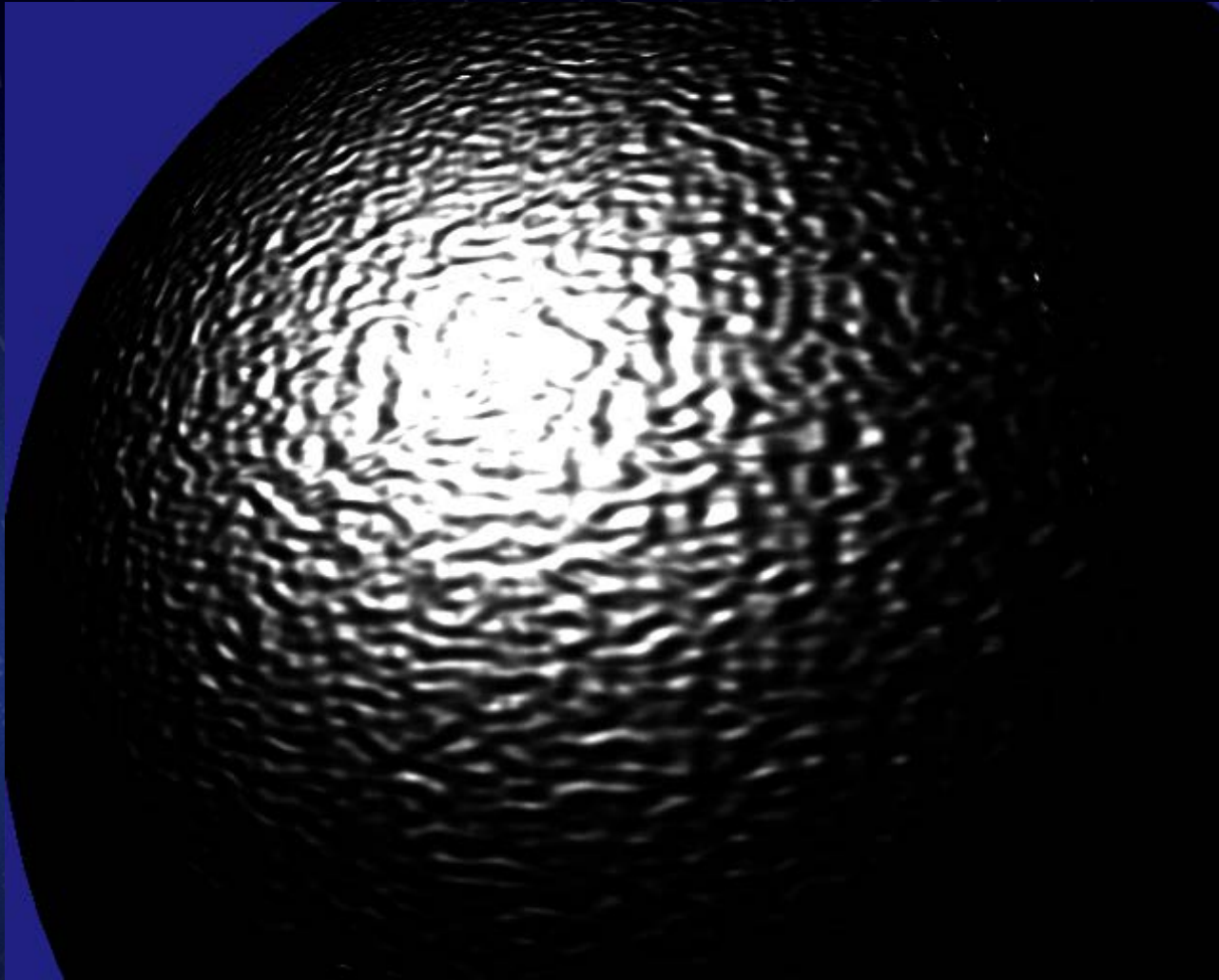
# Semi-Rough (Glossy)

- A surface patch contains micro-facets with continuously distributed normals
- Light reflects off facets, 'spreads out'
- In 'semi-rough' surfaces distribution of micro-normals biased to macro-normal



From Advanced Real-Time Reflectance, by  
Dan Baker, Naty Hoffman & Peter-Pike Sloan

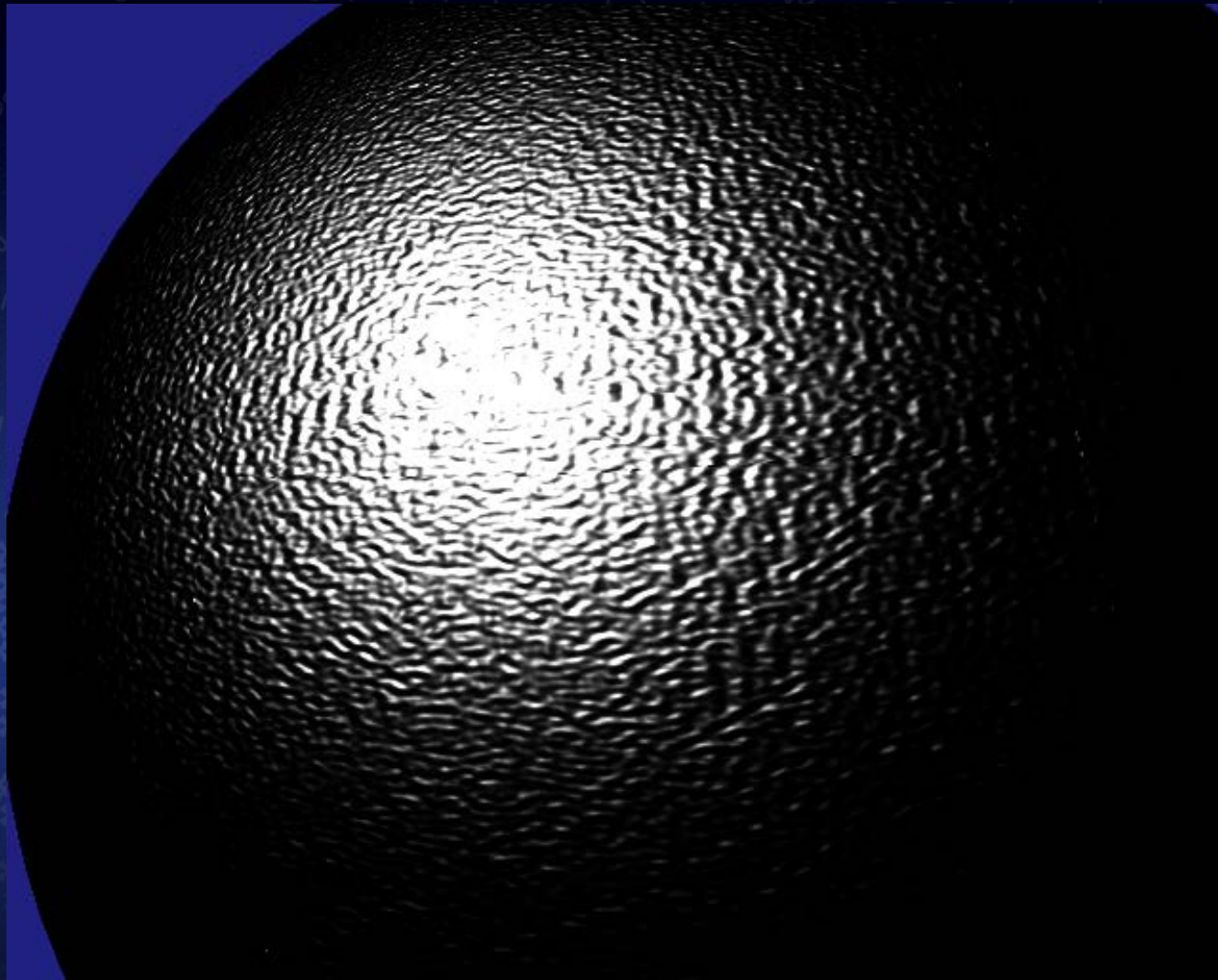
# Semi-Rough (Glossy)



From Advanced Real-Time Reflectance, by  
Dan Baker, Naty Hoffman & Peter-Pike Sloan

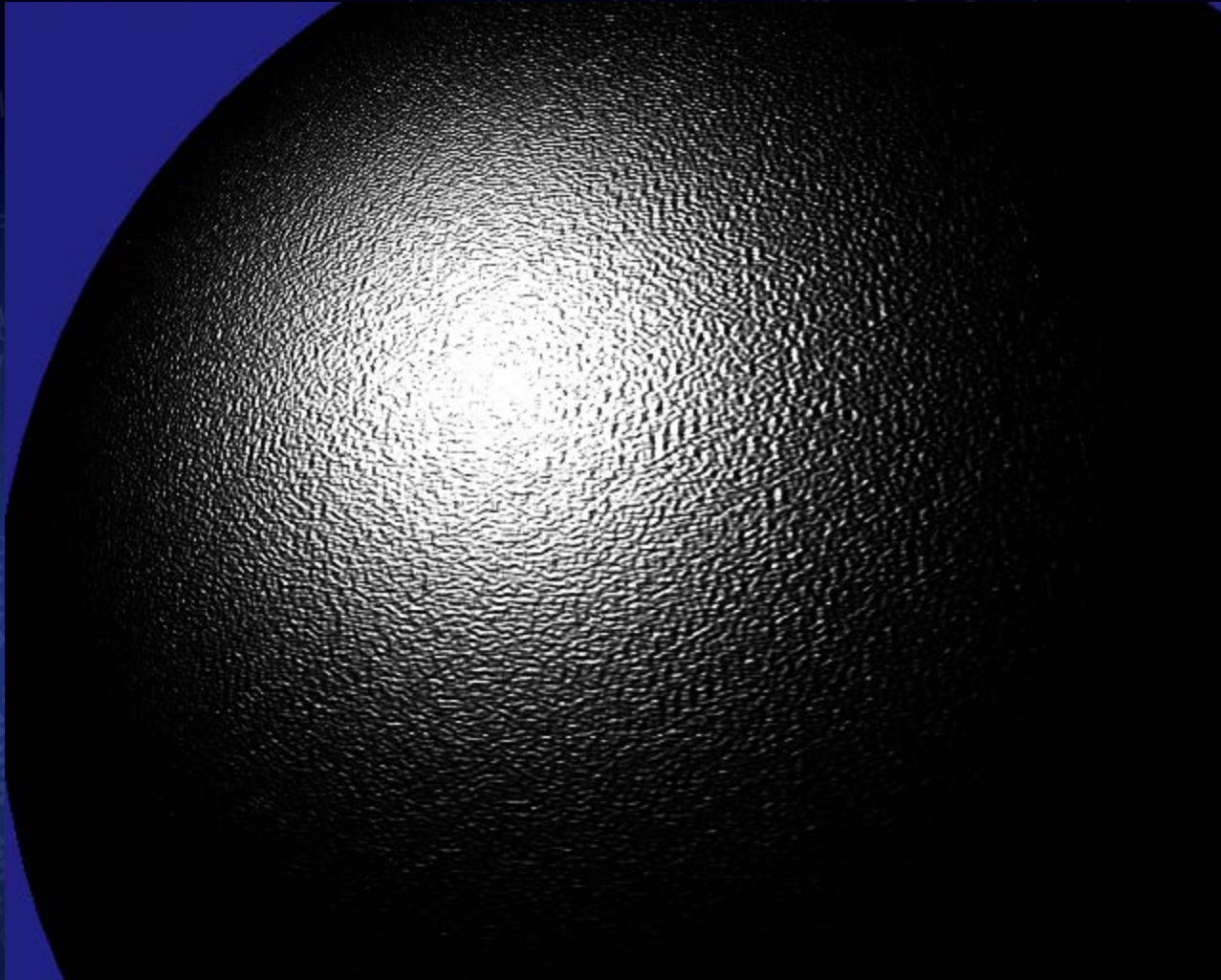


# Semi-Rough (Glossy)



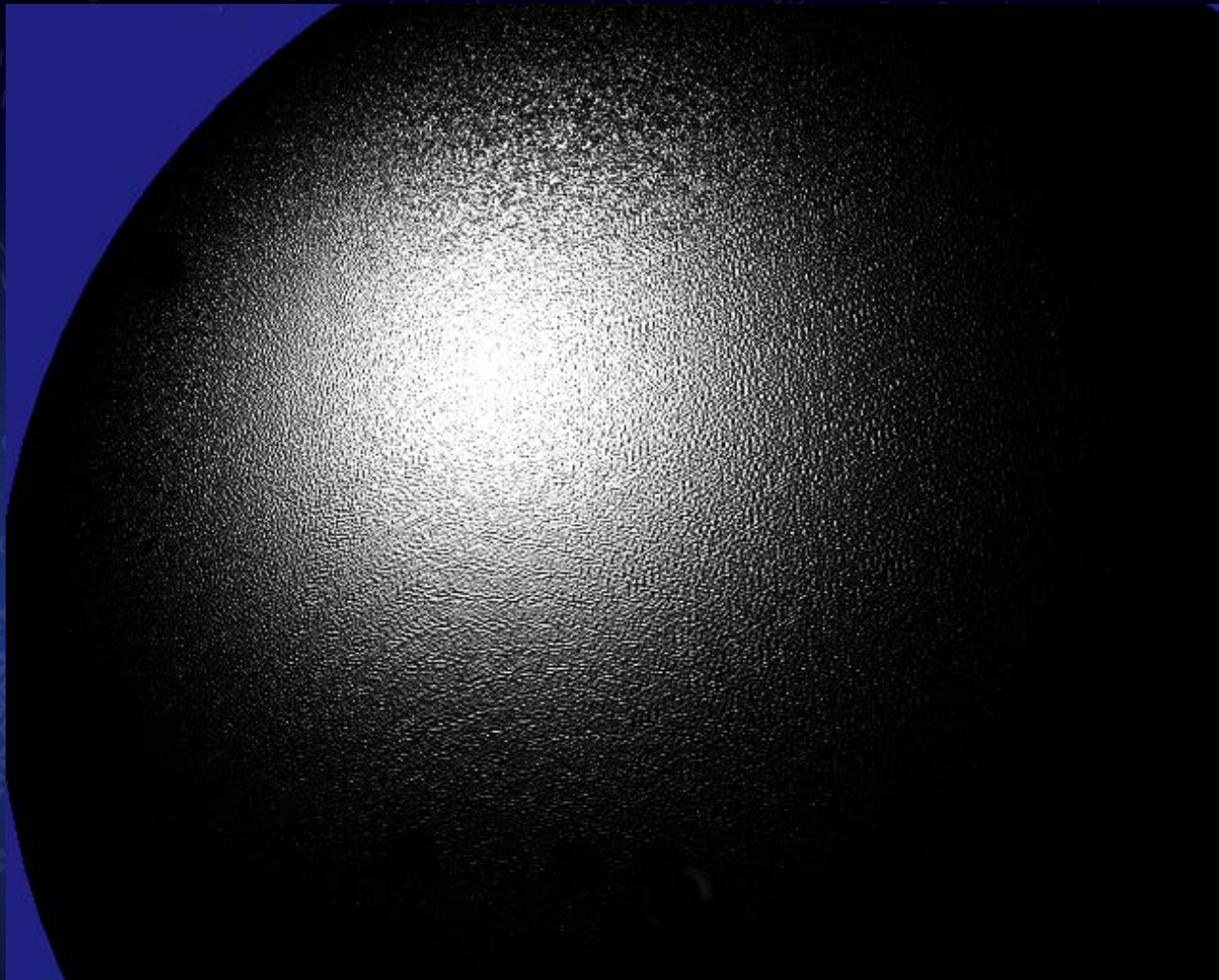
From Advanced Real-Time Reflectance, by  
Dan Baker, Naty Hoffman & Peter-Pike Sloan

# Semi-Rough (Glossy)



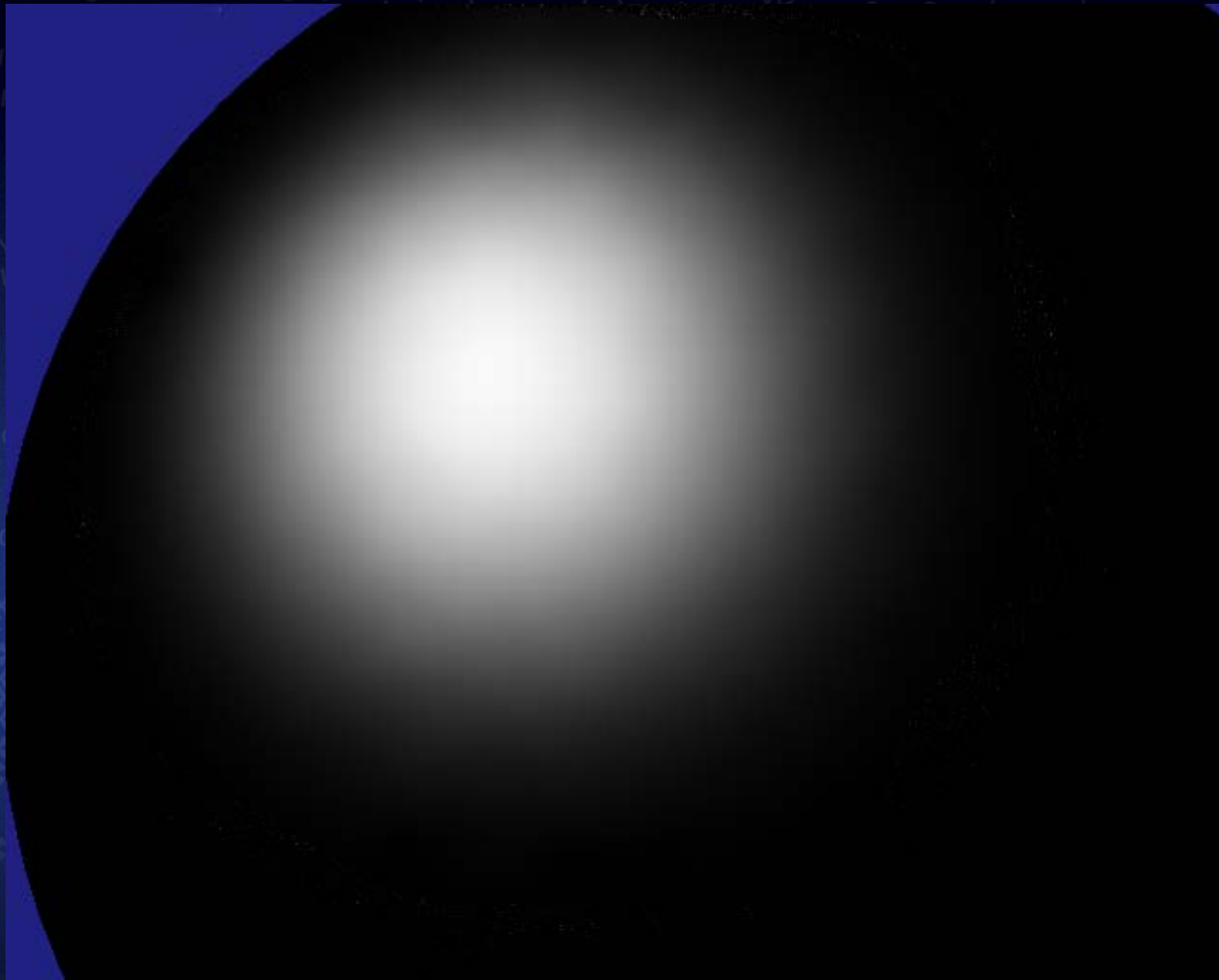
From Advanced Real-Time Reflectance, by  
Dan Baker, Naty Hoffman & Peter-Pike Sloan

# Semi-Rough (Glossy)



From Advanced Real-Time Reflectance, by  
Dan Baker, Naty Hoffman & Peter-Pike Sloan

# Semi-Rough (Glossy)



From Advanced Real-Time Reflectance, by  
Dan Baker, Naty Hoffman & Peter-Pike Sloan

# Semi-Rough (Glossy)

- Lights reflect as highlights
- Blurry reflection of environment



From Advanced Real-Time Reflectance, by  
Dan Baker, Naty Hoffman & Peter-Pike Sloan

# Semi-Rough Metal

- The reflectance of a metallic rough surface is dominated by the highlight
- Multiple-bounce reflections create an additional, more diffuse reflection
  - More strongly colored since each bounce increases the colors saturation

From Advanced Real-Time Reflectance, by Dan Baker, Naty Hoffman & Peter-Pike

# Semi-Rough Dielectric

- **Highlight tend to be weaker in dielectric surfaces due to lower reflectance**
  - **For the same reason, multiple-bounce reflections are less noticeable**
  - **Diffuse mostly due to subsurface scattering**

# Rough Dielectric

- Normal distribution is extremely random
- Almost uniformly diffuse with some retroreflection



From Advanced Real-Time Reflectance, by  
Dan Baker, Naty Hoffman & Peter-Pike Sloan



## Smooth dielectric materials (2)

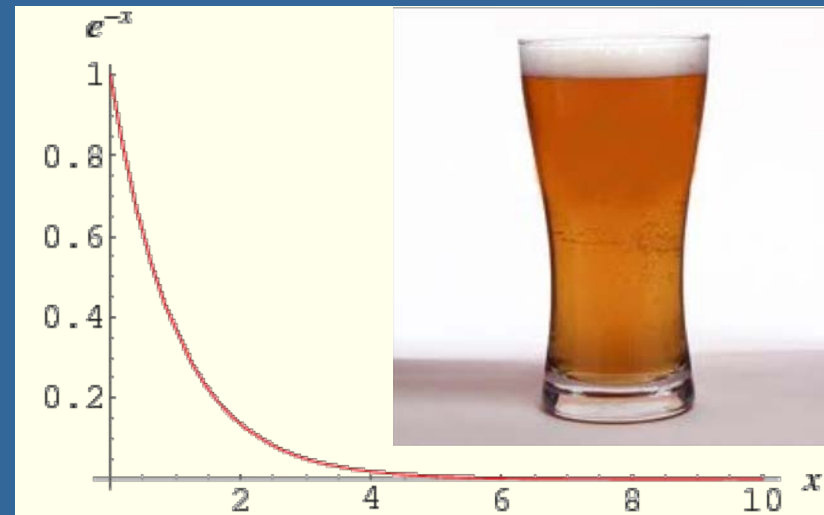
# Homogeneous impurities

E.g. Water, transparent plastic, glass...

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

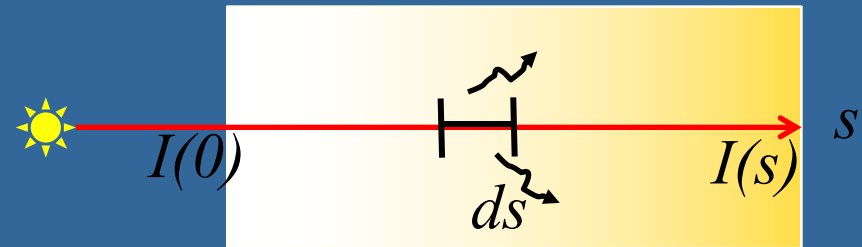
# Beer's Law

Constant intensity decrease at greater distance due to out-scattering and absorption.



$$dI = -CIds$$

$$I(s) = I(0)e^{-C*s}$$



# Beer's law



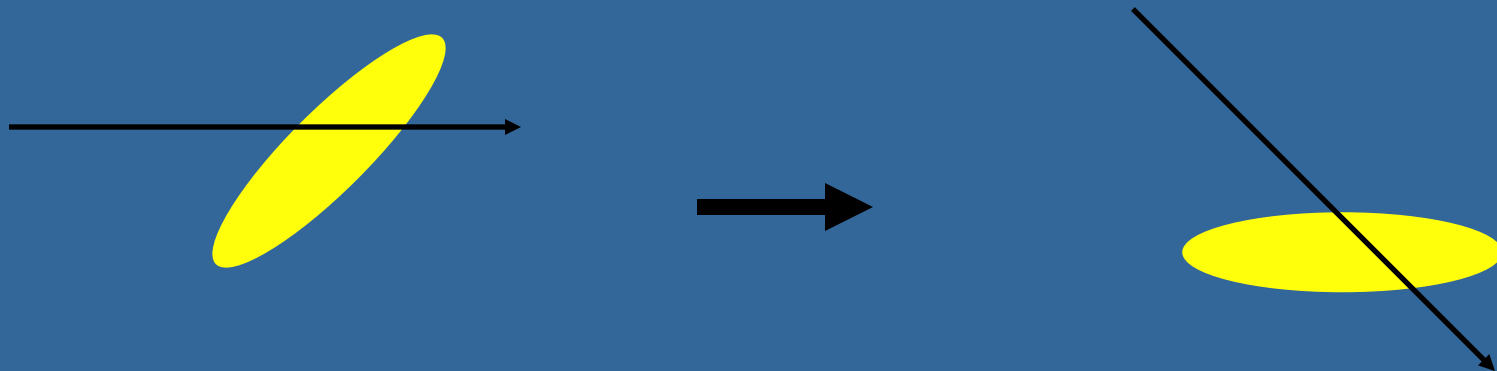
*The taller the glass, the darker the brew,  
The less the amount of light that comes through*

# RAY TRACING ADDITIONALS

- Geometrical objects
  - Ray intersections: transform ray into object space
  - Constructive Solid Geometry
  - Blobs
- Procedural textures
  - Fractals from noise
- Optics
  - E.g., depth-of-field

# 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

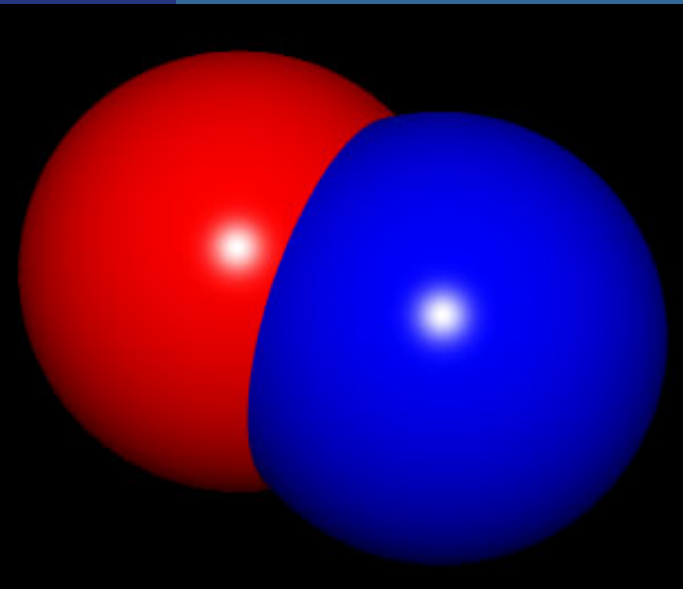


# Geometry: Constructive Solid Geometry (CSG)

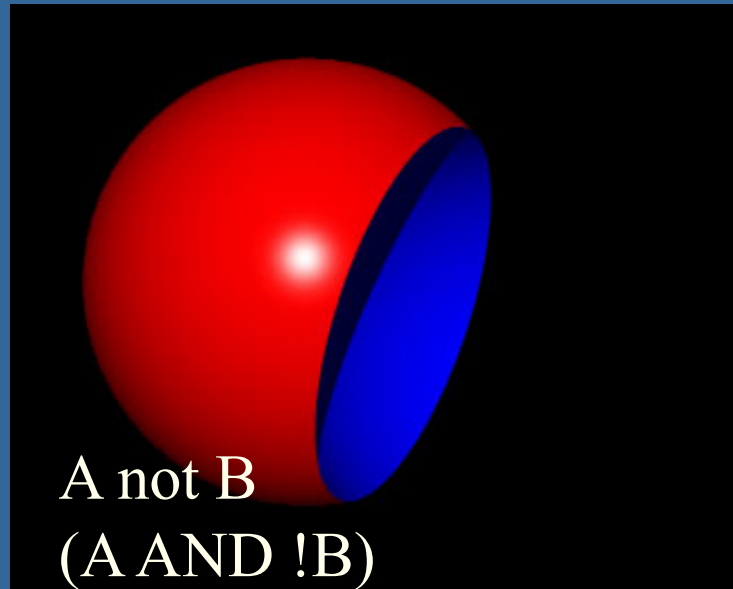
- Boolean operations on objects
  - Union
  - Subtraction
  - Xor
  - And
- Simple to implement
- Must find *all* intersections with a ray and an object
- Then do this for involved objects, and apply operators to found interval

# Geometry: Constructive Solid Geometry (CSG)

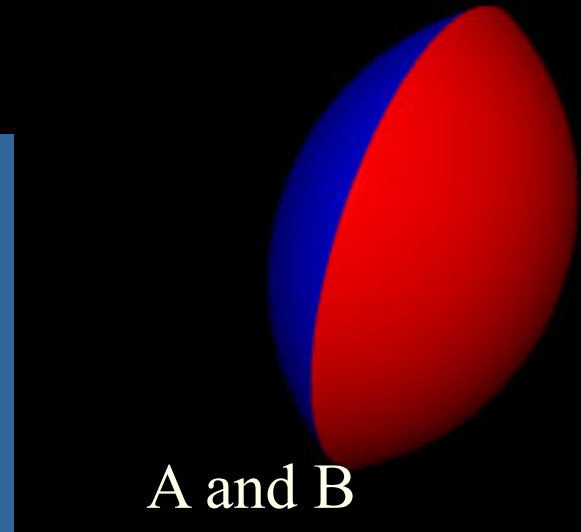
- Examples, operators:



A union B (OR)



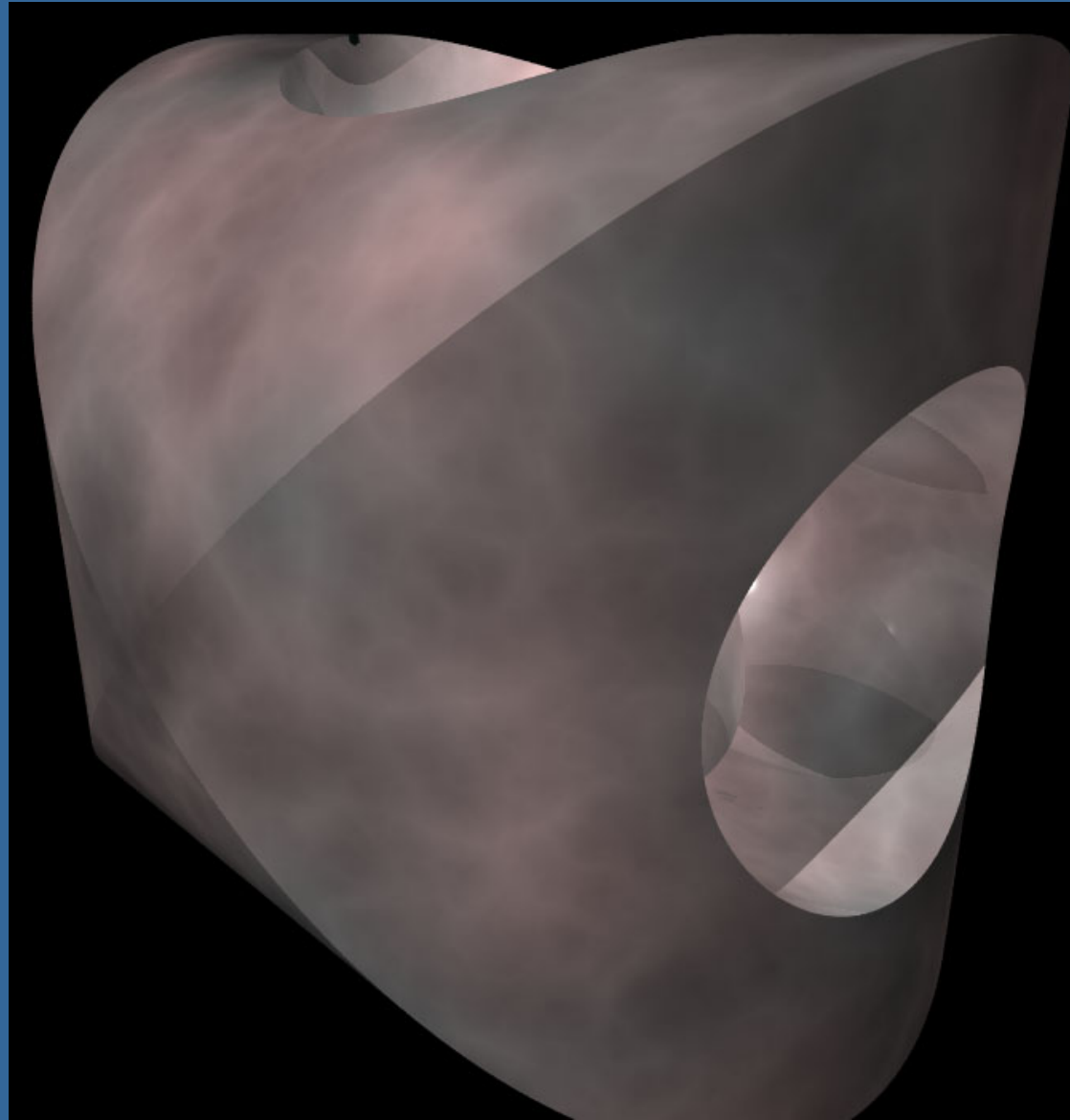
A not B  
(A AND !B)



A and B

# Geometry: Constructive Solid Geometry (CSG)

- Another example
- Done with 4 cylinders

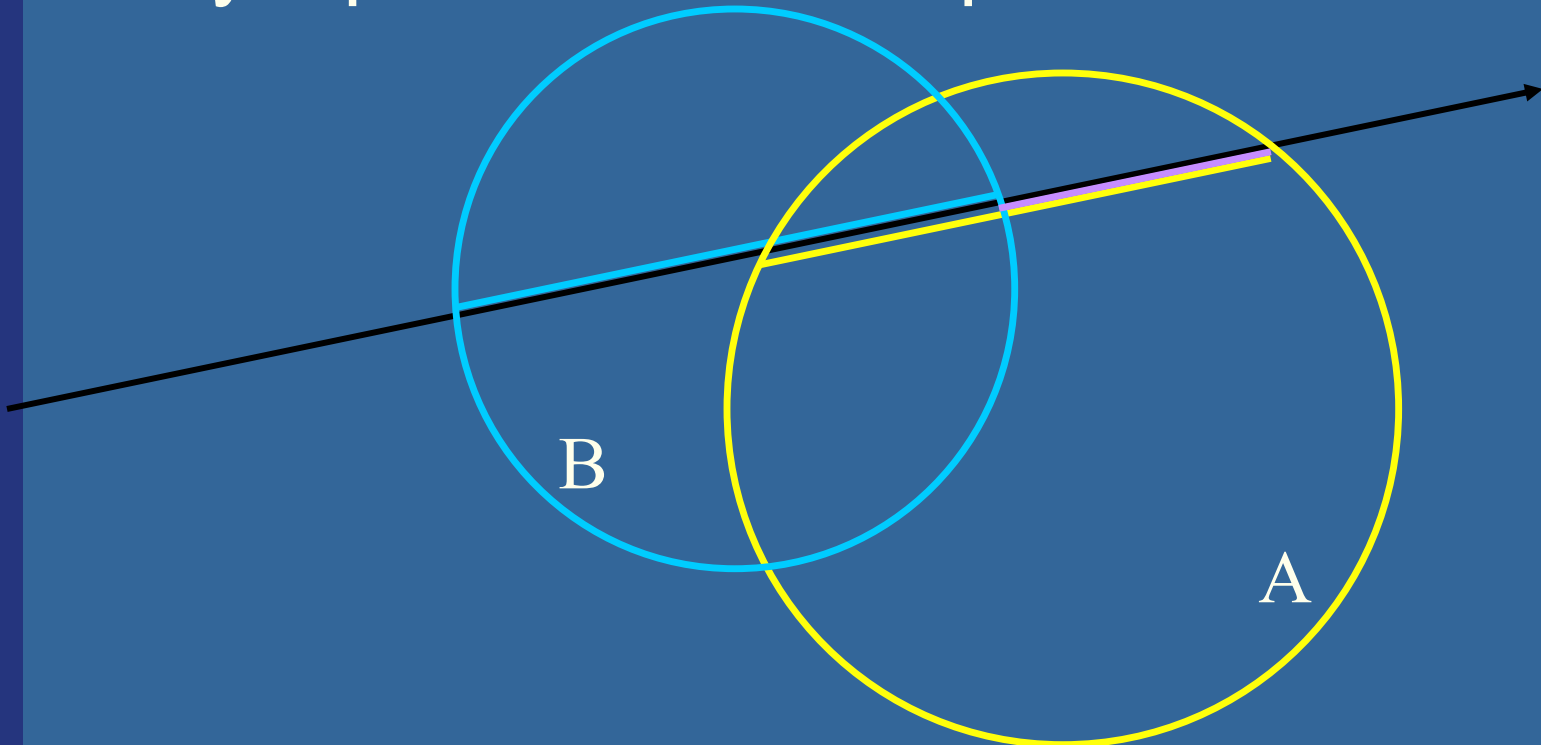




# Constructive Solid Geometry (CSG)

## How to implement

- Try: sphere A minus sphere B



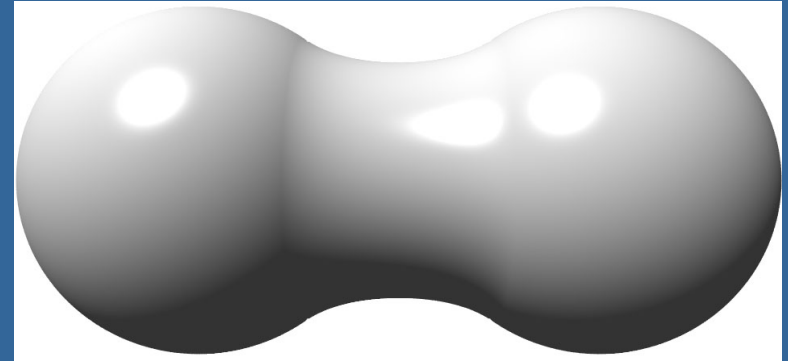
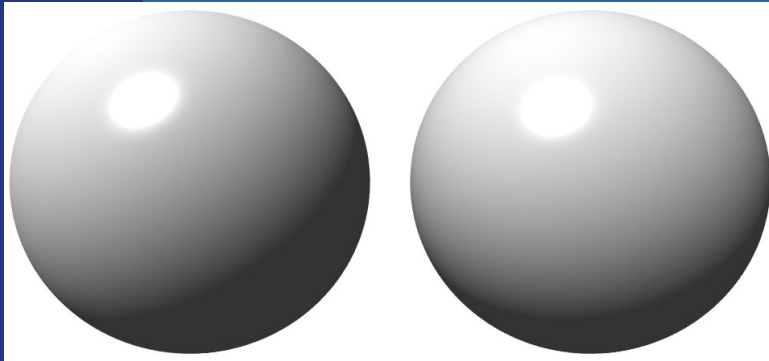
- In summary: find both entry and exit points on both spheres. Such two points on a sphere is an interval (1D). Apply the operator on these intervals

# CSG

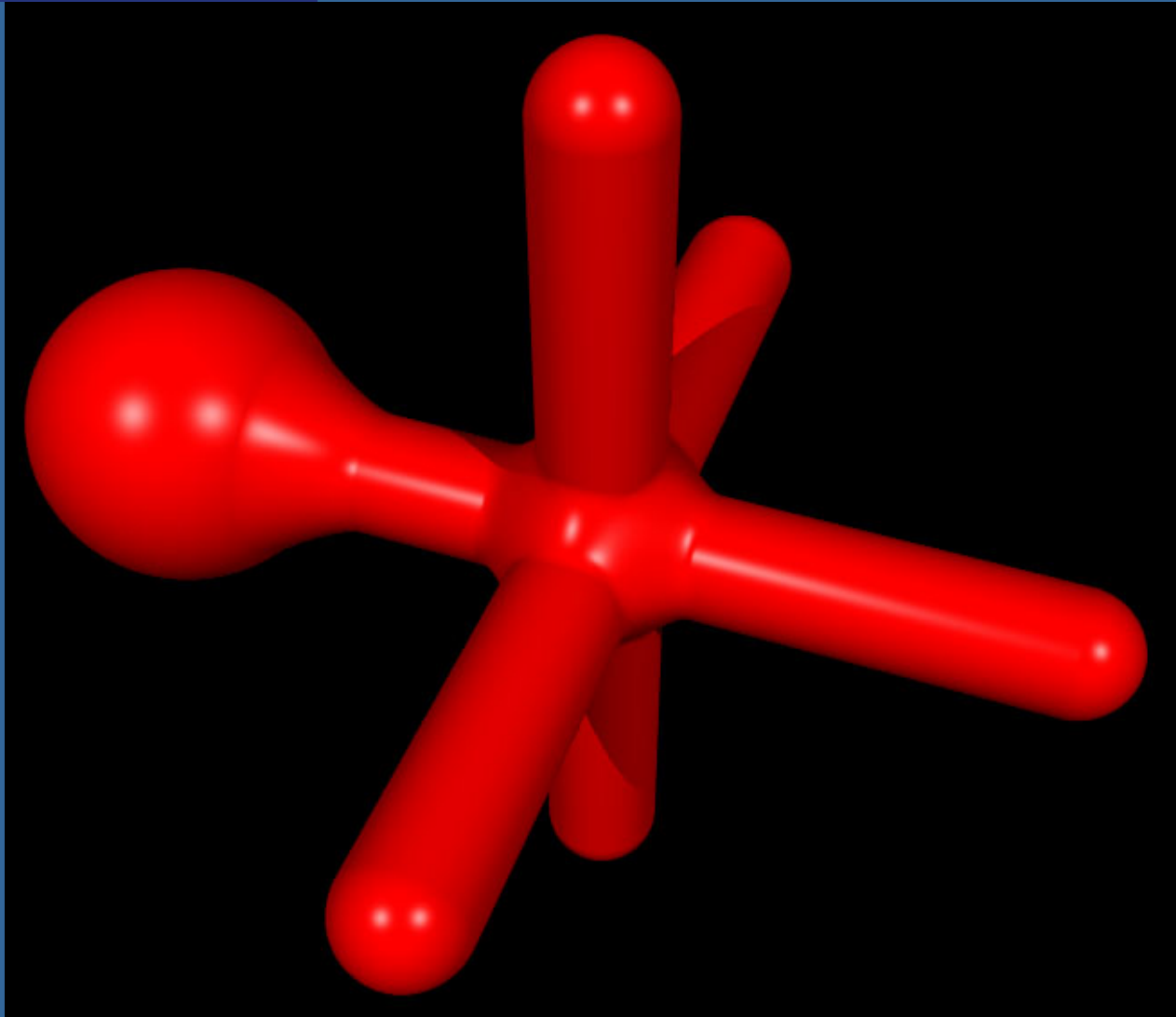
- Works on any geometrical object, as long as you can find **all** intersection point along a line
  - So, be careful with optimizations...
- And objects should be closed
  - Example: put caps on cylinder.

# 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

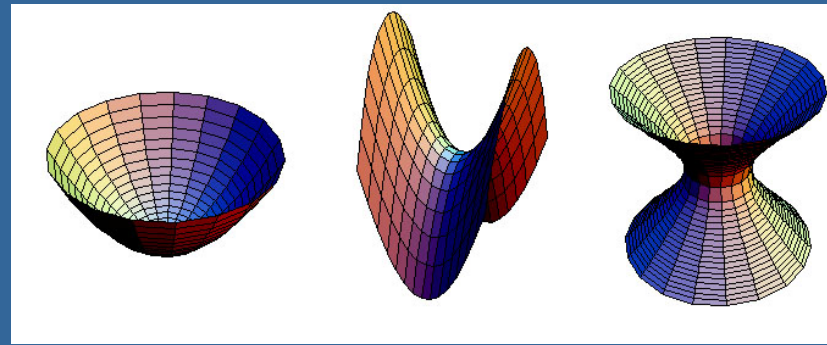


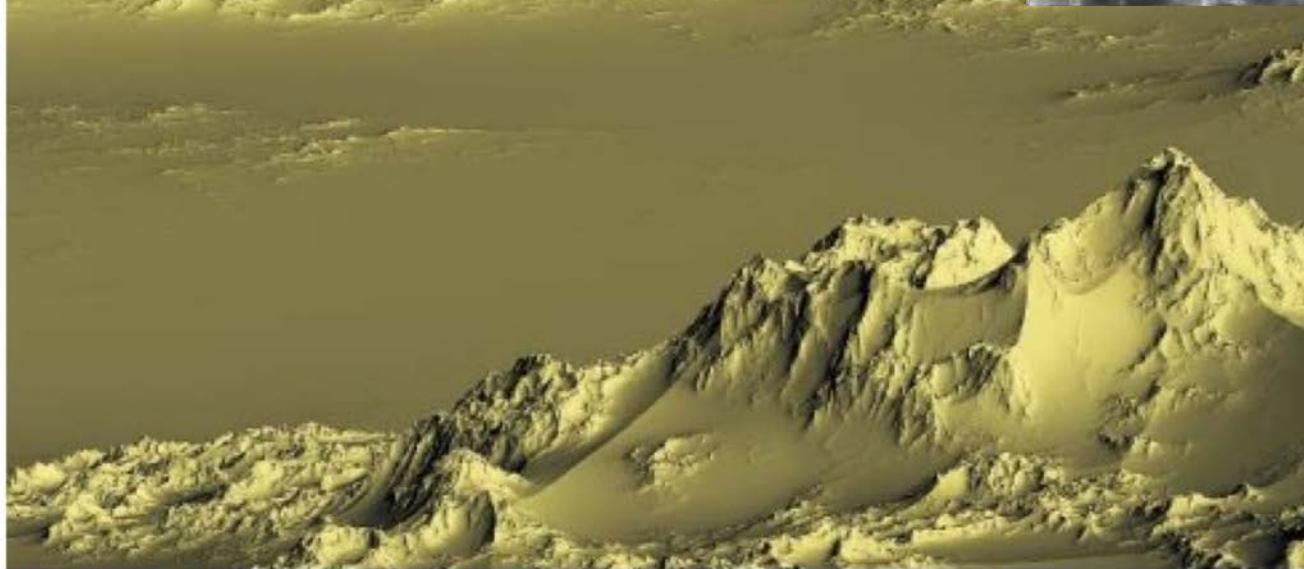
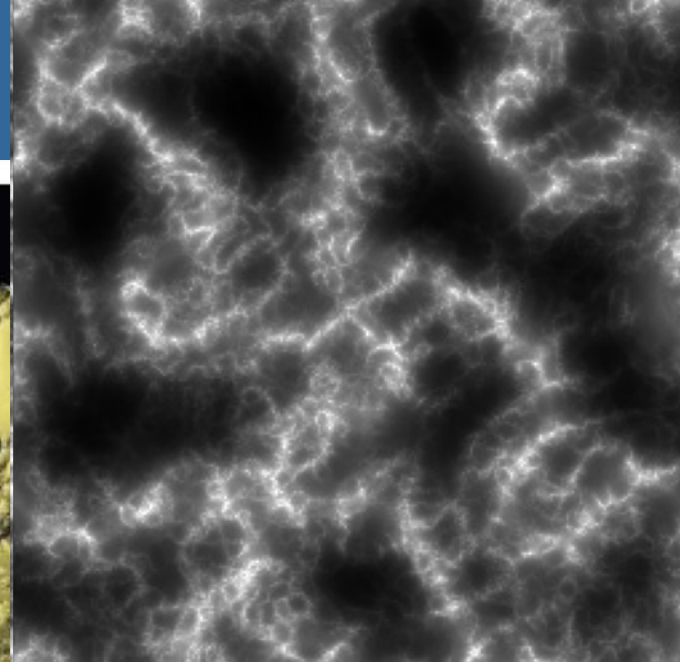
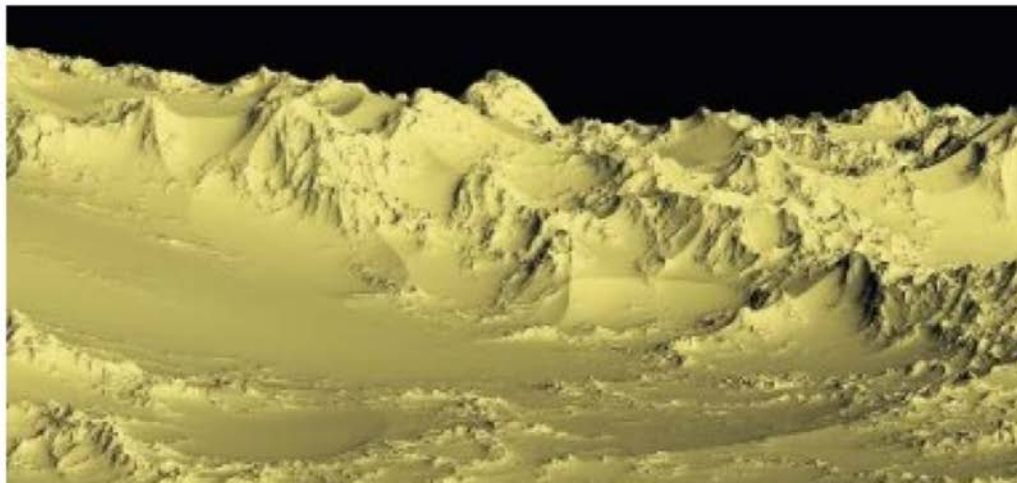
# Blob example



# Geometry

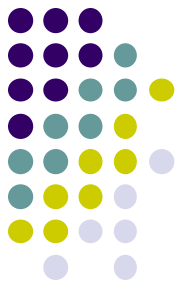
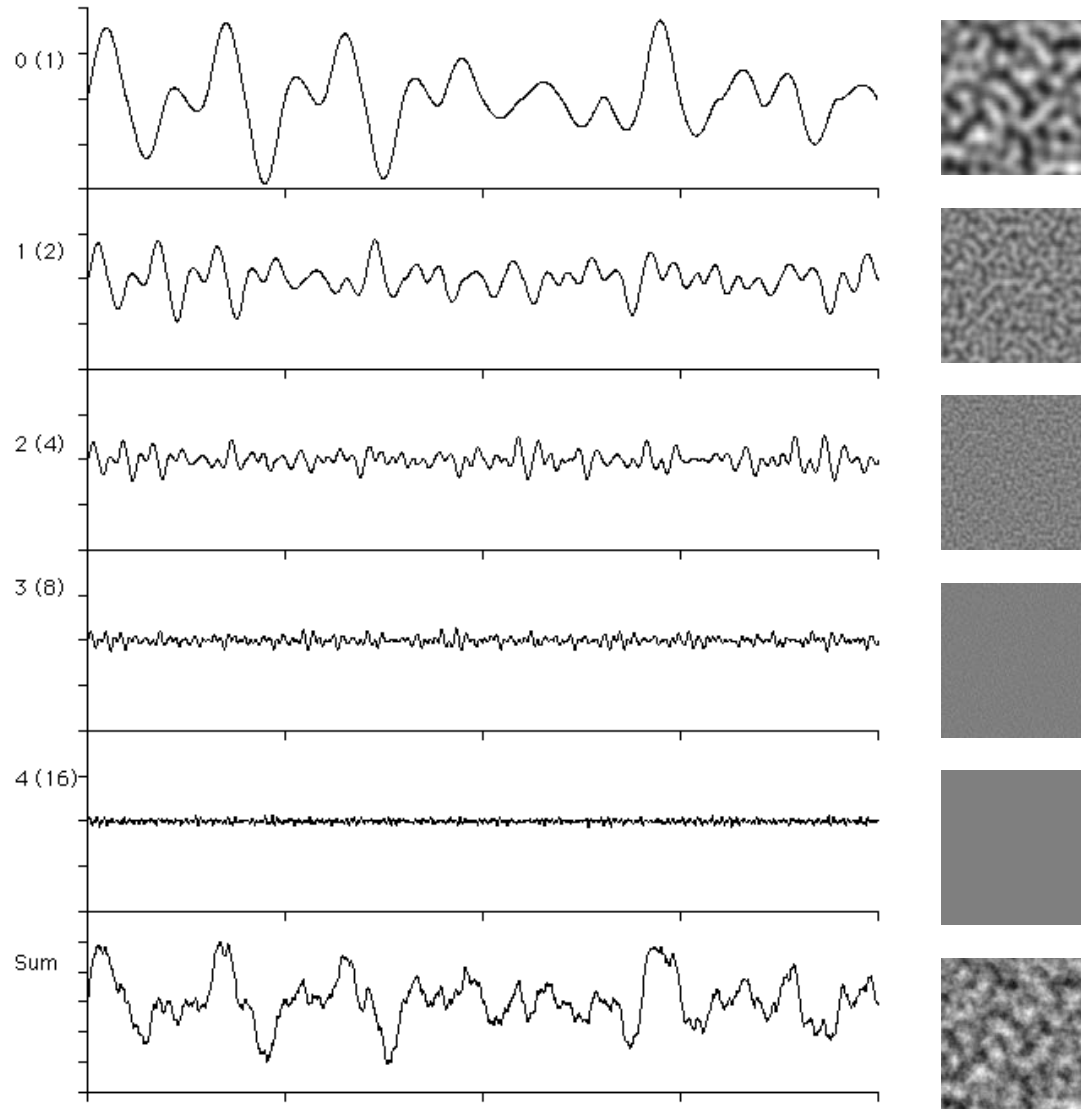
- Quadrics (2:a-gradsytor)
  - Cone, cylinder, paraboloids hyperboloids, etc.
- Higher order polynomial surfaces
  - Example: torus, 4th degree
- Fractal landscapes
  - Pretty simple, fast algorithm exist

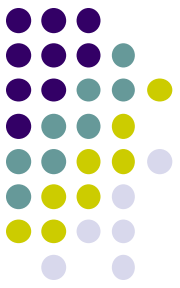




**FIGURE 17.4** The very first QAEB-traced terrain. The terrain model is the “ridged multifractal” function described in Chapter 16. Copyright © F. Kenton Musgrave.

# Perlin Noises in 1-D

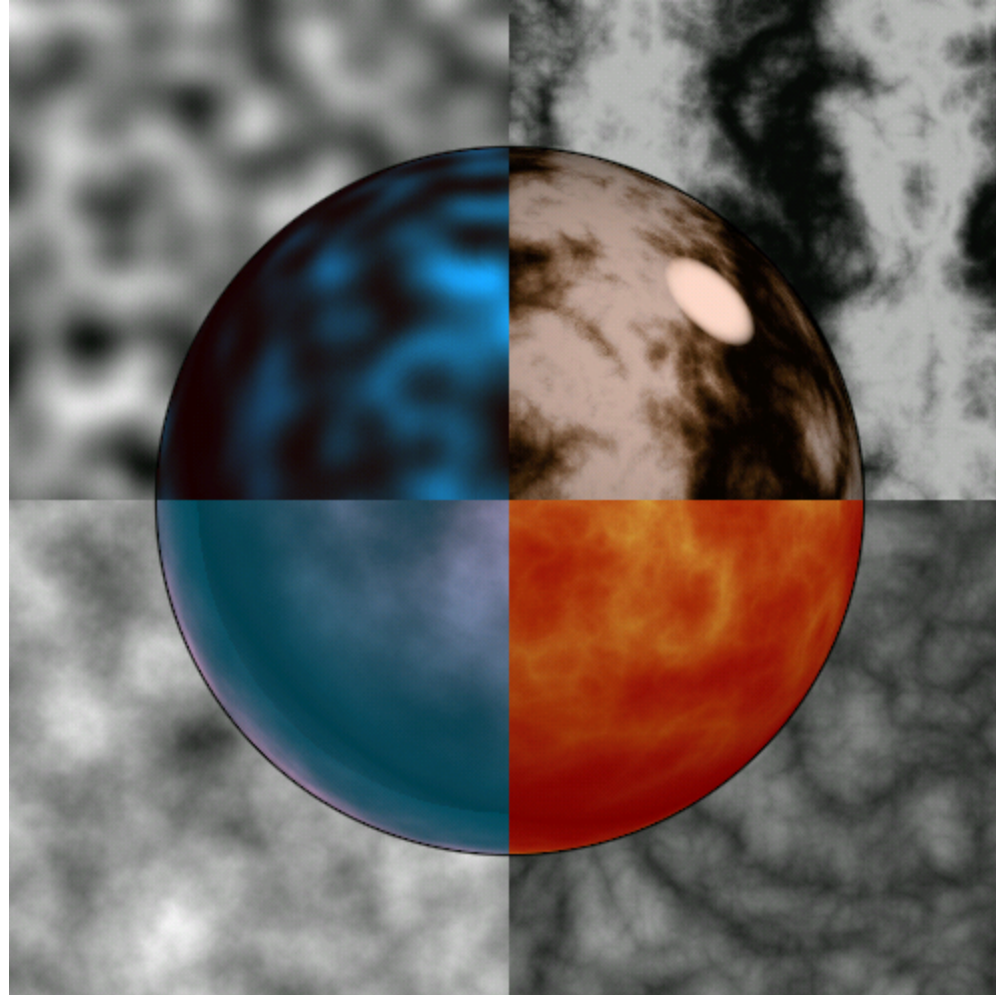




# Weighted Sums

## noise:

- Worn metal
- Water wave



## $\text{Sin}(x +$

## $\text{Sum}[1/f * |\text{noise}| ] ):$

- Turbulent flows
- Fire
- Marble

## $\text{Sum}[1/f * \text{noise}]:$

- Rock
- Mountains
- Clouds

## $\text{Sum}[1/f * |\text{noise}| ]:$

- Turbulent flows
- Fire
- Marble
- Clouds





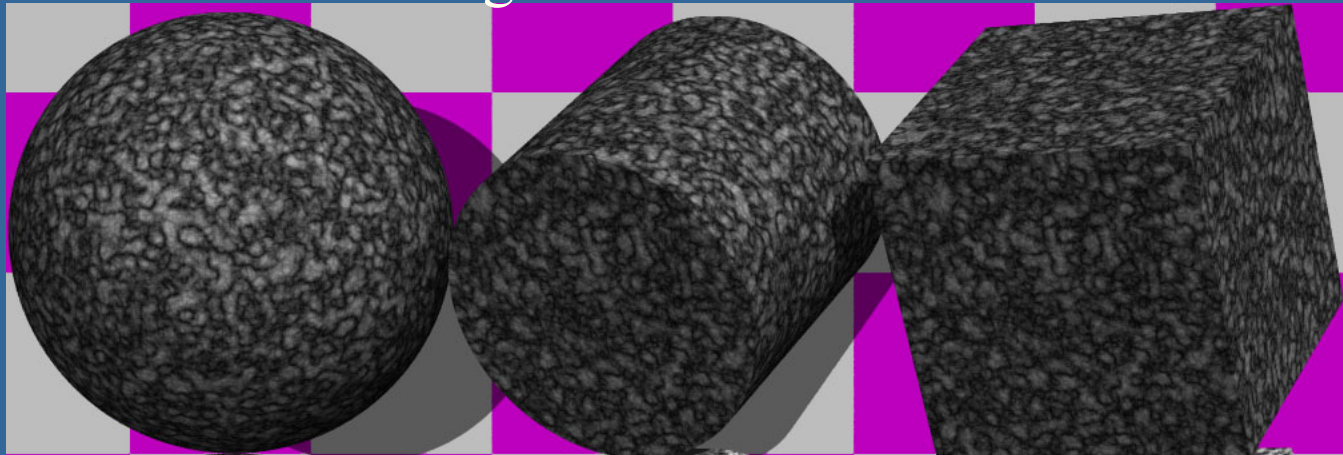
# More fractal examples...

WebGL examples:

- <https://www.shadertoy.com/view/MdfGRX>



Procedural texturing:



# Optics

- You can add
  - Fog
  - Light fall off :  $1/d^2$
  - Fresnel equations
  - Depth of field
  - Motion blur
  - ...

# Optics

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



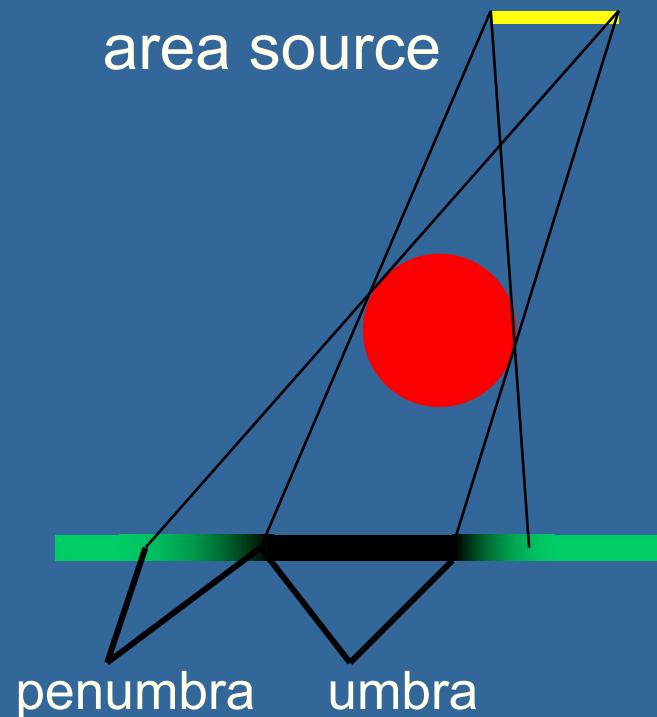
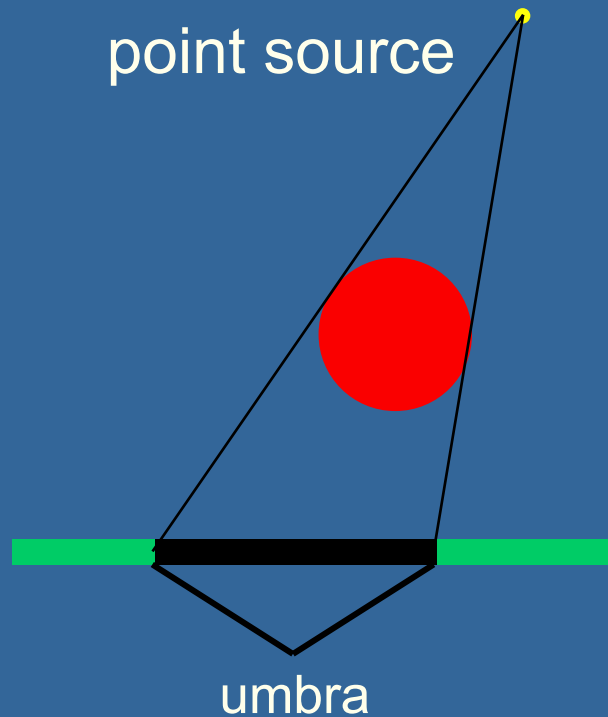
# Soft shadows

- Higher level of realism
- Examples

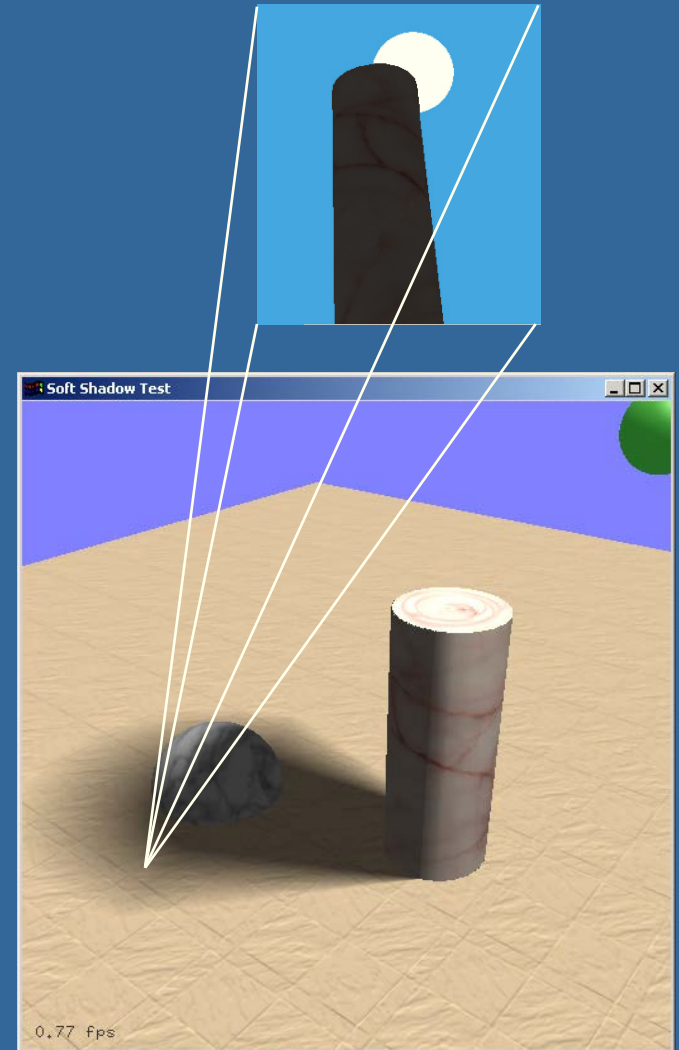
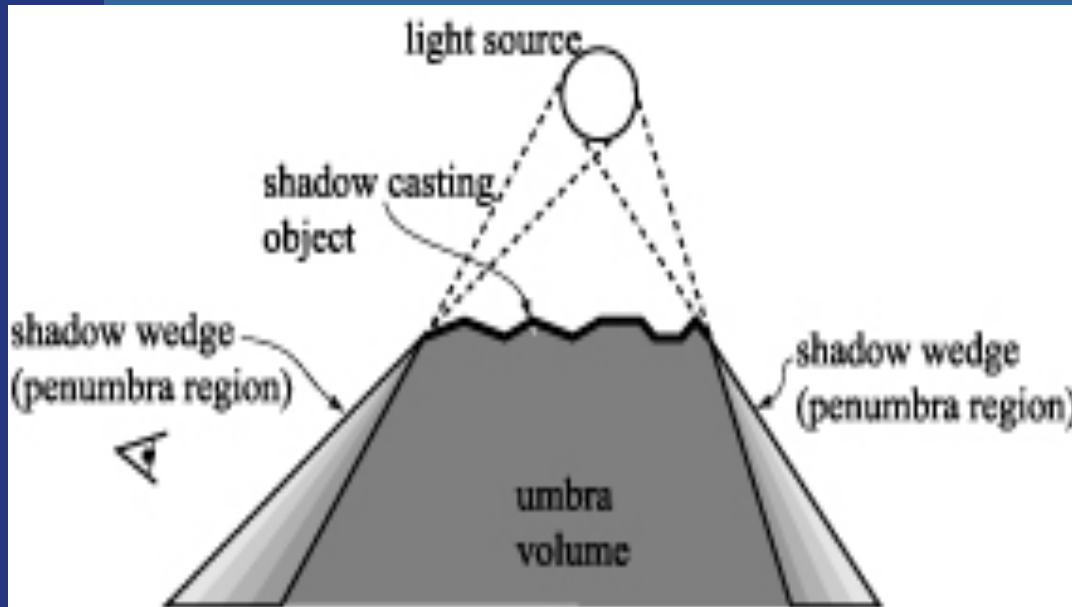


# Soft shadows

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

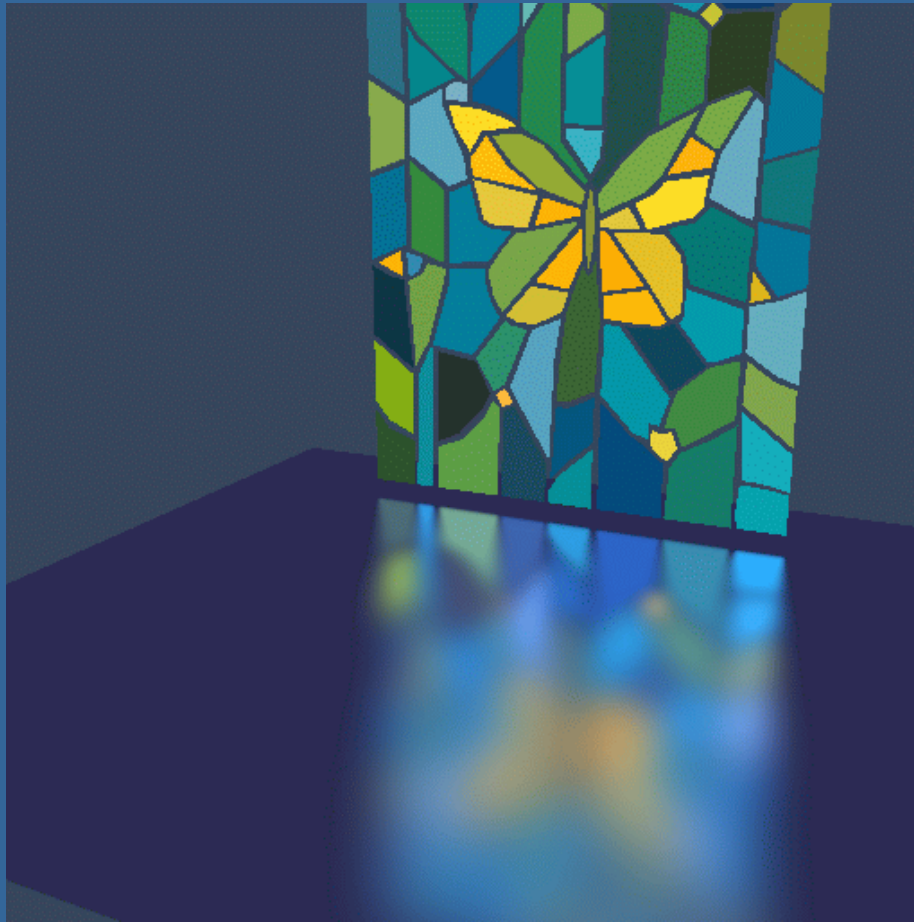


# Example



# Glossy (blurry) reflections

- Trace many reflection directions
  - Each perturbed slightly



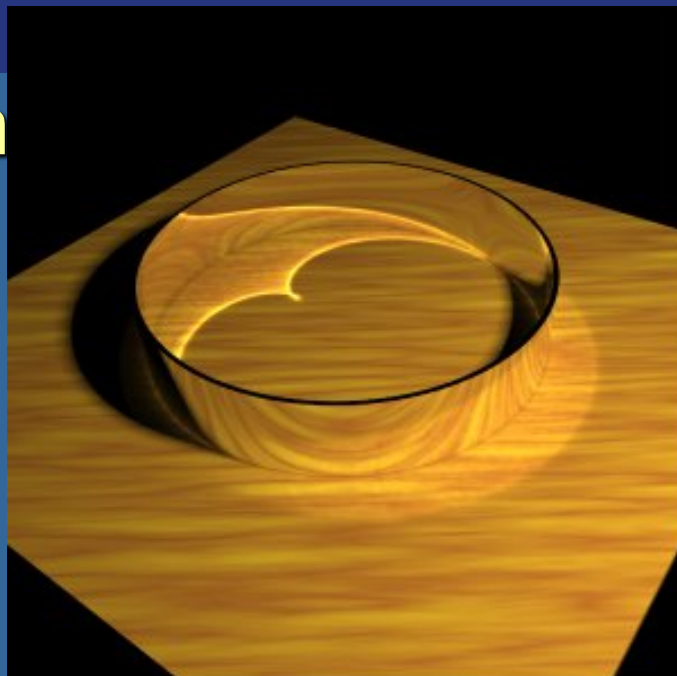
Do the same  
with transmission



# Caustics

Photorealism

etc



# Speed-up techniques

- 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
    - the primary rays (eye rays) are typically so few compared to all other secondary rays, so often not worth optimizing.

# Typical Exam Questions

## – what you need to know

- Draw grid (plain/hierarchical/recursive)
  - Mailboxing
- Draw all our other spatial data structures:
  - Octree/quadtrees, AABBSP-tree (kd-tree), polygon-aligned BSP tree, Sphere/AABB/OBB-tree,
- What's a
  - skip-pointer tree?
  - Shadow cache?
  - Shading cache?
  - Kd-tree? (=AABBSP with fixed split-plane order)
- Describe ray/BVH intersection test
- The Fresnel-effect: metal vs dielectrics)
  - How does glass/water/air behave?
  - How does metal behave?
- Describe how to implement ray/object intersection for Constructive Solid Geometry

THE END