

# Ray Tracing II



Image: Nvidia RTX ray tracer

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

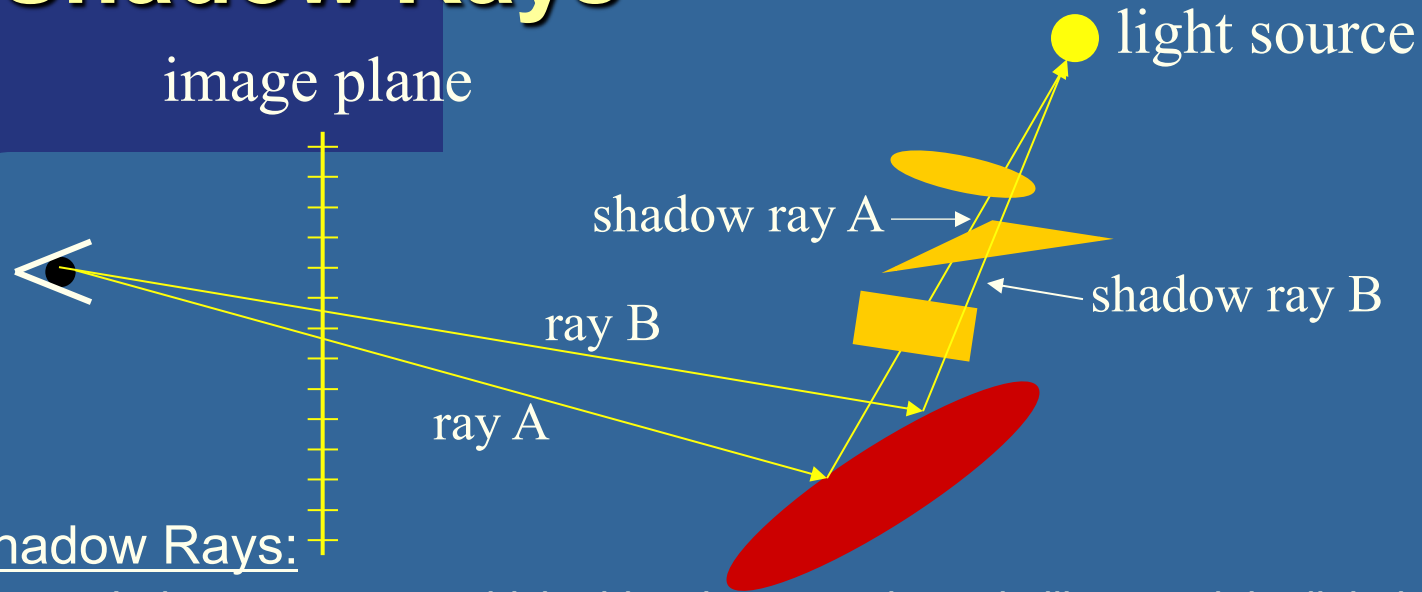
# Lab 7

- YOU MUST START NOW
  - Or you may fail!
  - Purpose of Lab 7:
    - Now, you have to implement more on your own (for real), without close guidance.
      - For real-time rendering and learning to do special effects.
      - Or: Path Tracer Lab for realistic beautiful rendering!

# Overview

- Shadow Rays and shadow cache:
  - can give speedup for shadow rays if cached triangle is large (i.e., high probability of next shadow ray hitting same triangle).
- Spatial data structures and ray traversal
  - Bounding volume hierarchies (BVHs)
  - BSP trees
  - Grids
  - Cache aware coding: Shoot primary rays according to a Hilbert Curve or Morton (Z) curve:
- Materials
  - Fresnell Effect
  - Beer's Law
- Additional ray tracing techniques
  - Constructive Solid Geometry
  - Fractals

# Shadow Rays



## Shadow Rays:

- It does not matter which object between the red ellipse and the light is detected
  - Any-hit-traversal instead of first-hit-traversal (here, "first" means first along the ray)
- The point is in shadow if we find at least **one** object between

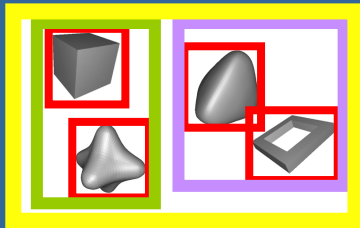
## Shadow Cache (optimization):

- Assume shadow ray A hits the triangle
  - You can 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.
- E.g., use a cache per level of reflection-/refraction-ray recursion
- Shadow cache not popular in *parallel* ray tracing. For *parallel* ray tracing, *packet* ray tracing and *ray streams* have a similar effect by utilizing ray coherence, together with "any-hit-traversal".

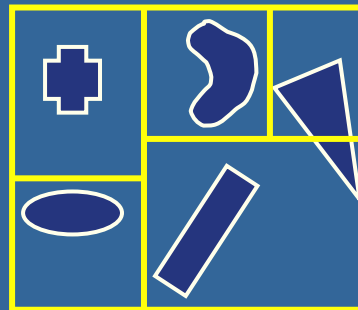
Alternatively, you could use shadow mapping.

# Spatial data structures and Ray Tracing

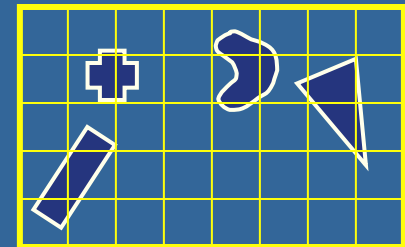
- Use spatial data structures to get faster rendering
  - Because ray tracing is often slow
  - Avoids intersection tests between the ray and each object in the scene.
    - Rather, you test a small subset
    - Typically,  $O(\log n)$  instead of  $O(n)$  for each ray
- We will look at



Bounding volume hierarchies (BVHs)



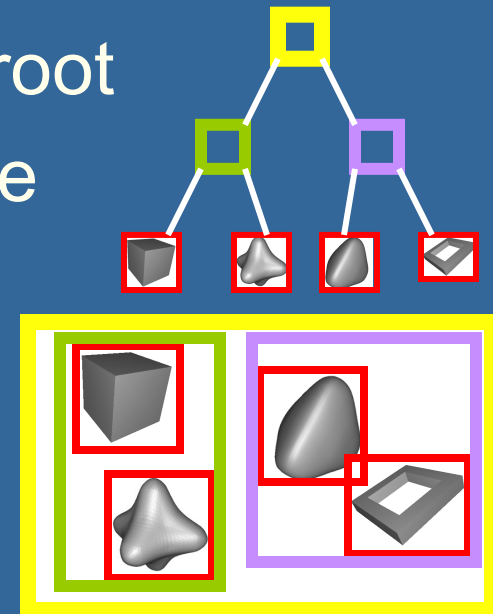
AA BSP trees



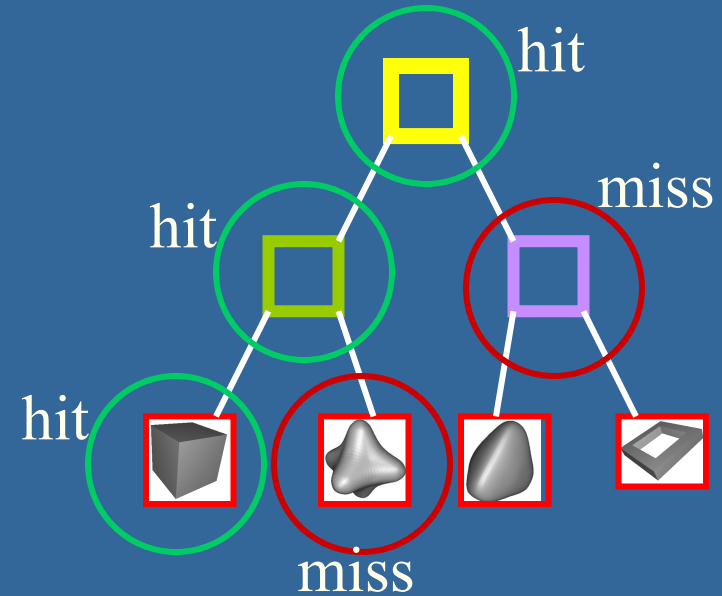
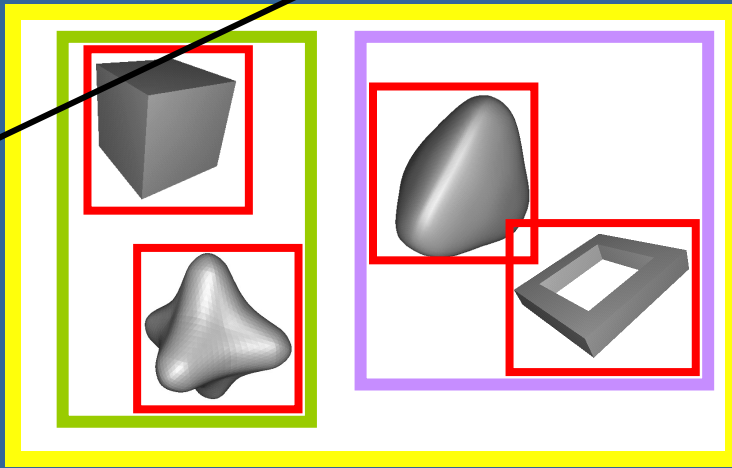
Grids

# 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



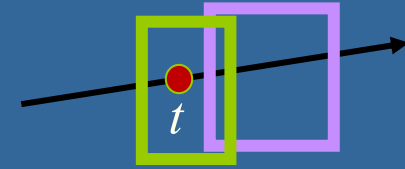
- Without BVH, we would test each triangle of every object against the ray
- With BVH:
  - Only test the triangles of the leaves against the ray
  - Plus some AABBs, but these are cheap

# 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

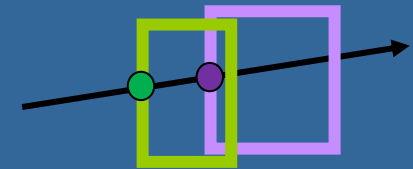


# BVH traversal optimizations



If an intersection  $t$  is found for the green box, there is no closer  $t$  in the purple box

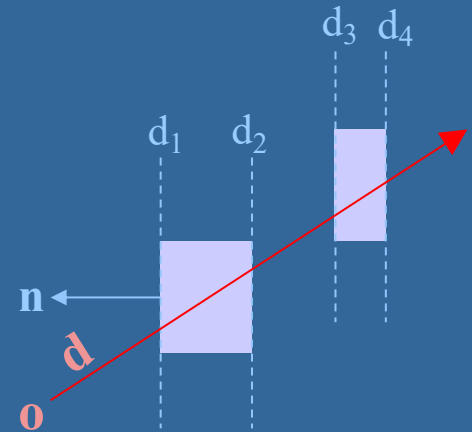
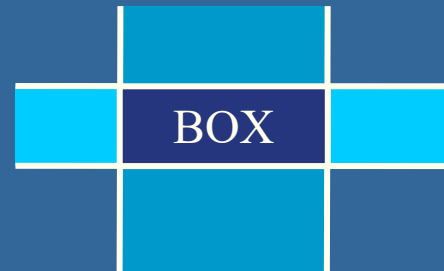
1. Use current closest intersection as an upper bound on how far the ray may "travel"
  - Example, if a ray hits a polygon at distance  $t$ , then we don't need to traverse a BV which is located farther than  $t$ .
2. Can also sort the BVs with respect to hit distance along ray, and only open up if necessary.
3. Shadow cache can be used for shadow rays



- Shadow cache is not efficient for small triangles and are not popular for parallel code (CPU/GPU). E.g.:
  - parallelism may break the assumption that the next hit is close to the previous hit.
  - But shadow maps may be used instead.
- They are also not popular for path tracing (see next week) due to rays being incoherent.
- However, they can be good for soft shadows.

# 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



$$\text{set } a = -(n \cdot o) / (n \cdot d)$$

$$b = -1 / (n \cdot d)$$

$$\Rightarrow t_1 = a + d_1 * b$$

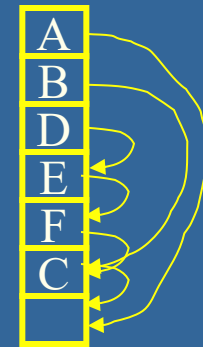
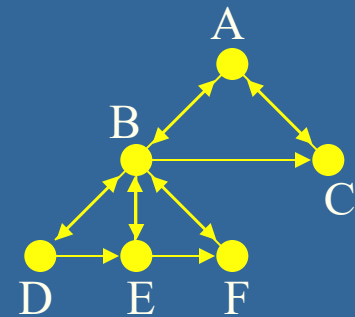
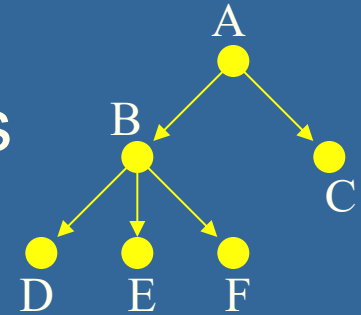
$$t_2 = a + d_2 * b$$

$$t_3 \dots$$

Just 1 madd instr. per ray/plane test

# BVH traversal... skip-pointer trees

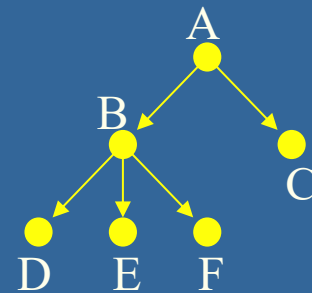
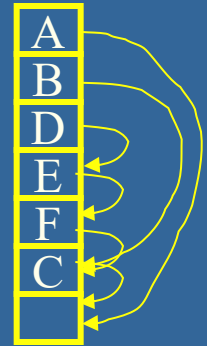
- Standard (depth-first) traversal is slow:
  - Involves recursion
  - And memory may be allocated once per node
- Left-child, right-sibling, parent pointers avoids recursion
  - Instead follow pointers
- Store these in a clever way, with skip pointers
  - Store nodes in depth-first order
  - A skip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



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

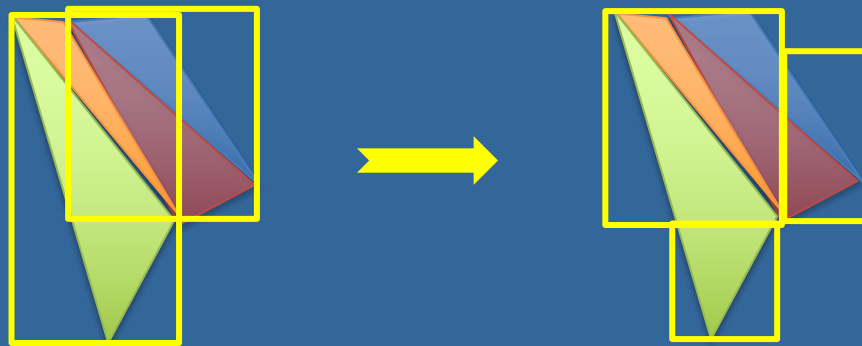
# BVH traversal... skip-pointer trees

- 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  $0..n-1$ ).
  - Is a problem when first sorting the children on distance.
  - Also, 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 intersection testing in parallel.
    - Then, better the parent just stores one pointer to an array of all children. This also decreases the number of pointers needed to store, which significantly lowers memory usage, which increases cache coherency.

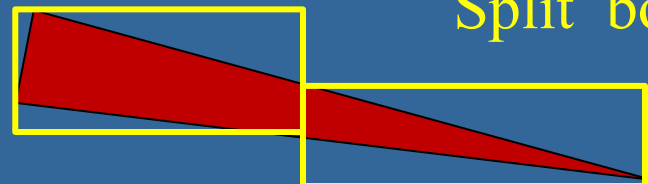
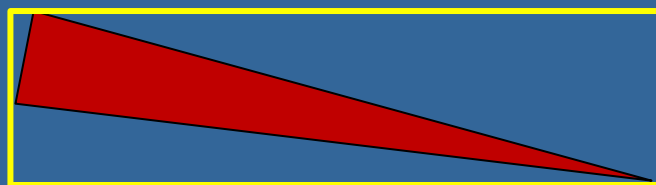


# Split BVHs (SBVH)

- SBVH – typically AABB hierarchies but allowing triangles to be part of several BV:s, to minimize empty volume and overlaps: E.g:
  - Requires careful analysis. See papers. Google on best ways or see Intel's Embree.
  - Popular for CPU ray tracing but not real-time GPU ray tracing.



Clip and split boxes. Add triangles to boxes if necessary.



Split boxes.

# Surface Area Heuristics (SAH)

SAH says that if a ray hits the parent, P, then the likelihood of hitting a child, i, is:  $\frac{Area_i}{Area_p}$

- Can be used to decide how to split a node to minimize ray traversal cost if a ray hits it. ...by minimizing the sum of the expected ray-traversal costs of the node's children after split.

- SAH cost =  $\sum_{i=1..#children} (\text{traversal\_cost} \times \text{traversal\_probability})_{\text{child}_i}$ 
  - traversal\_cost  $\approx$  #tris\_in\_child
  - traversal\_probability = (surface\_area<sub>ChildNode</sub> / surface\_area<sub>ParentNode</sub>)
  - Surface area means for the bounding volume.

Ray traversal cost of P  $\approx$  SAH cost = 
$$\sum_{i=1}^{\#ChildNodes} \#tris_i \frac{Area_i}{Area_p}$$

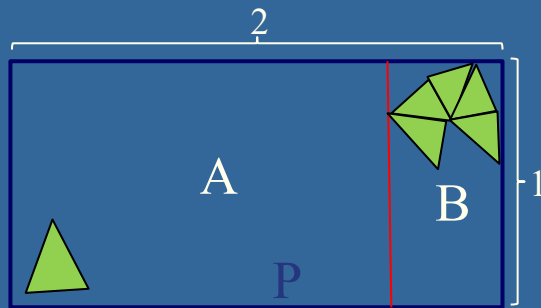
Where:

- #tris<sub>i</sub> = #triangles in child node i
- Area<sub>i</sub> = surface area of child i.
- Area<sub>p</sub> = surface area of parent node.

E.g., finding optimal split position (for kd trees)

Here:

- #tris<sub>A</sub> = 1
- #tris<sub>B</sub> = 5
- Area<sub>A</sub> = 5
- Area<sub>B</sub> = 3
- Area<sub>P</sub> = 6



SAH cost = 1 \* (5/6) + 5 \* (3/6)

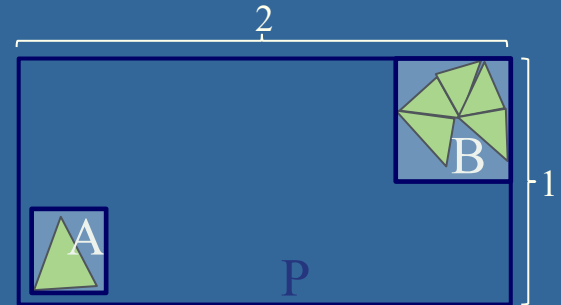
↑ 1 triangle      ↑ 5 triangles

Try many split positions and keep the one with lowest cost

E.g., finding optimal BVH subdivision (for BVHs)

Here:

- #tris<sub>A</sub> = 1
- #tris<sub>B</sub> = 5
- Area<sub>A</sub> = 1.5
- Area<sub>B</sub> = 2
- Area<sub>P</sub> = 6

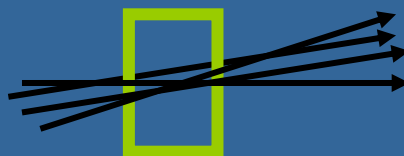
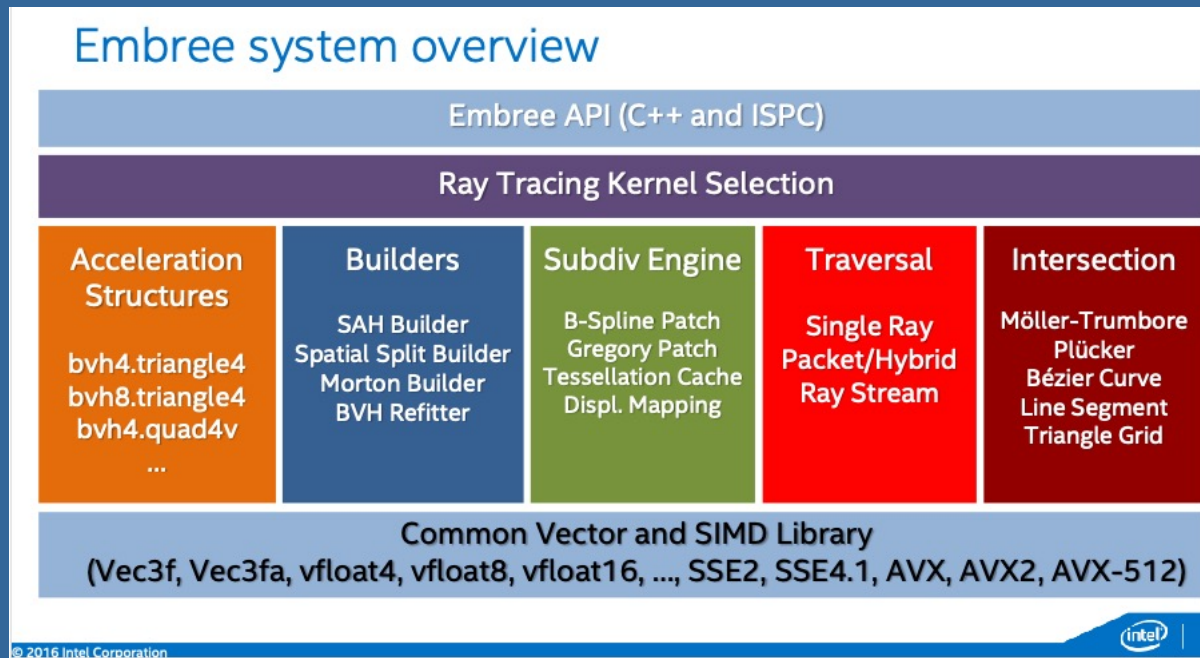


SAH cost = 1 \* (1.5/6) + 5 \* (2/6)

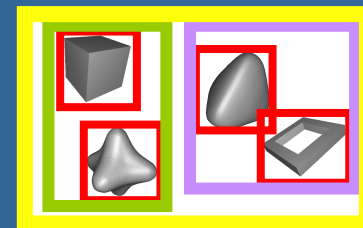
Try many divisions and keep the one with lowest cost

# Intel Embree

- Kind of the fastest general CPU ray tracer
- Uses BVHs
- Made by Ingo Wald et al.
- SAH – Surface Area Heuristics
- Packet ray tracing – traversing a packet of rays (e.g., 4,8,or 16) using SIMD instructions. Performance increase proportional to SIMD width
- Ray Streams – 1K-4K, 10-30% performance increase. Finds coherent ray batches requiring similar shading. (Shading is often most time consuming.)

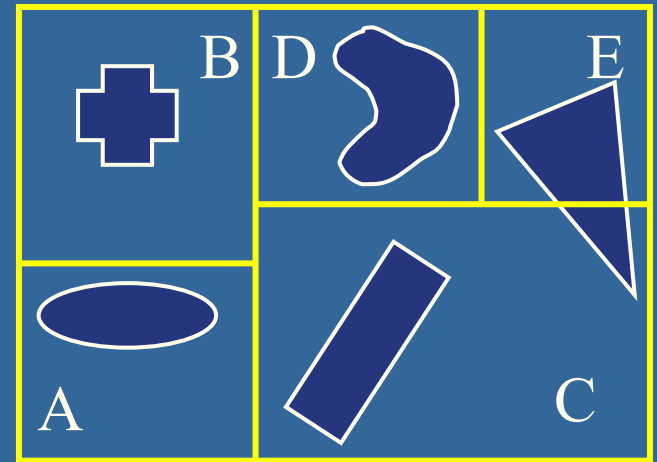


Packet ray tracing



# Axis-Aligned BSP trees

- An advantage is that that we automatically 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

```

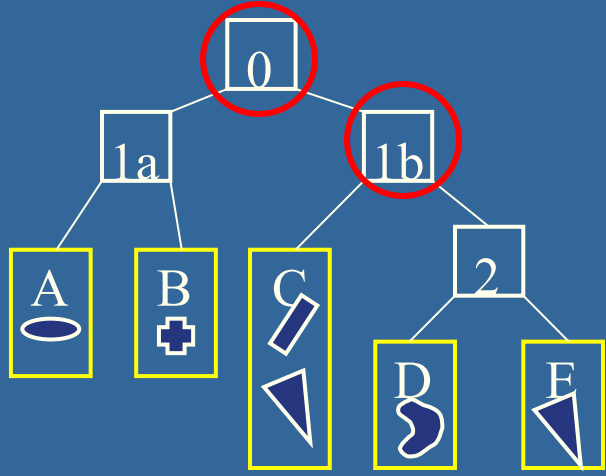
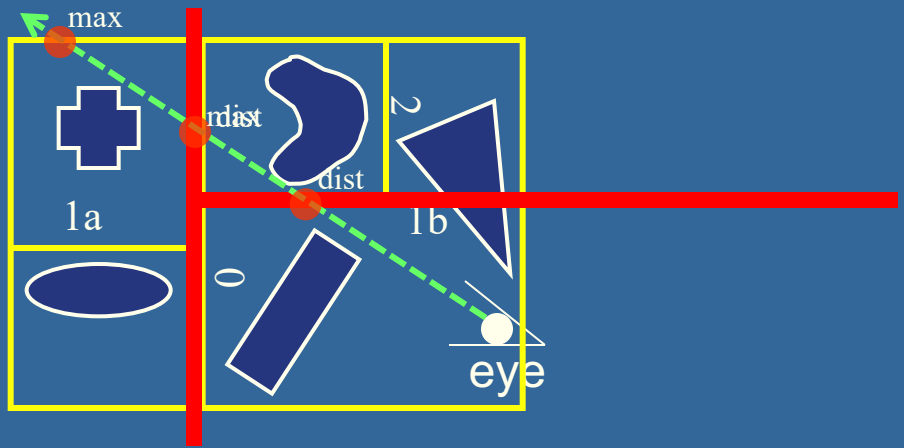
RayTreeIntersect(Ray, Node, min, max)
{
    if (node==NULL) return no_intersection;
    if (node is leaf)
    {
        test all primitives in leaf, discard if not between min and max;
        return closest intersection point if any;
    }
    dist = signed distance along Ray to cutting plane of Node;
    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, then farther side

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

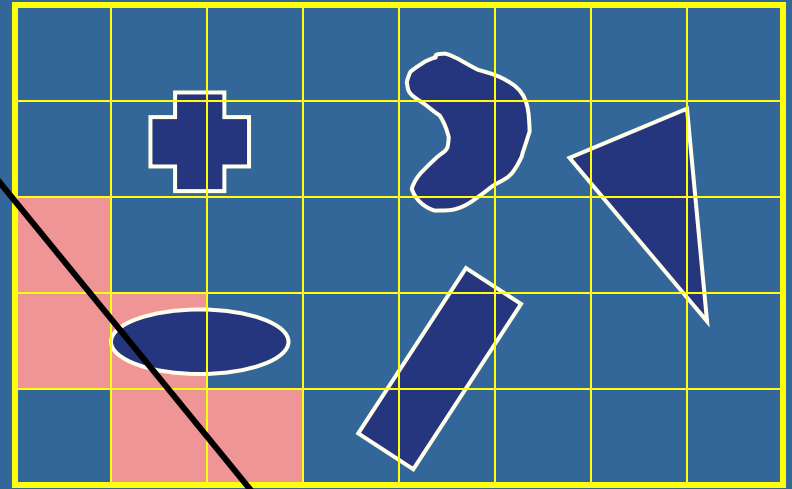


```

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
}
  
```

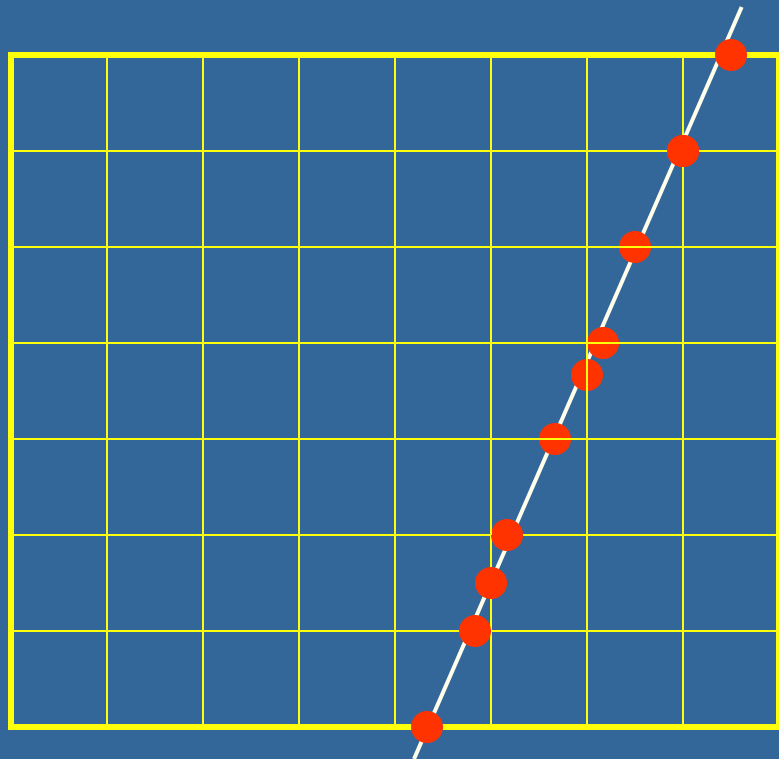
# Grids

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



# Grid Traversal Algorithm

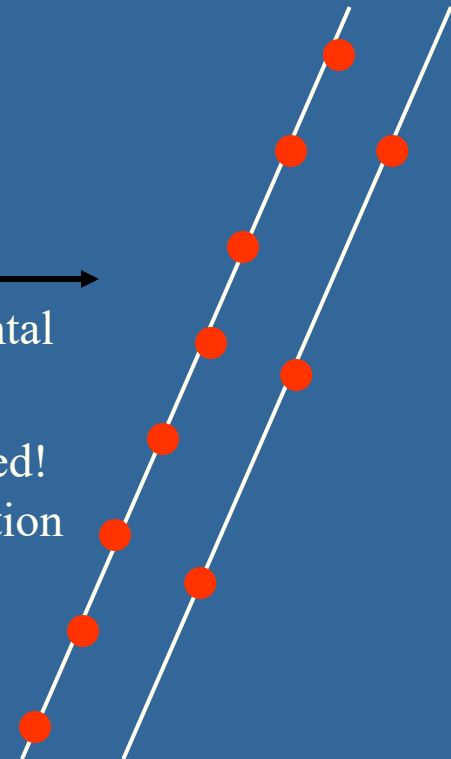
- A modified line generating algorithm can be used
  - Bresenham or DDA
- But easier to think in geometrical terms
  - Red circles mark where ray goes from one grid box to the next



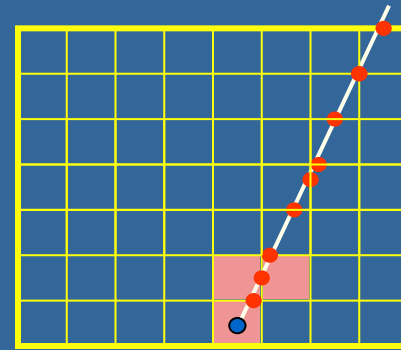
← Intersection points appear with irregular spacing

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)
```

```
    X = X + stepX;
```

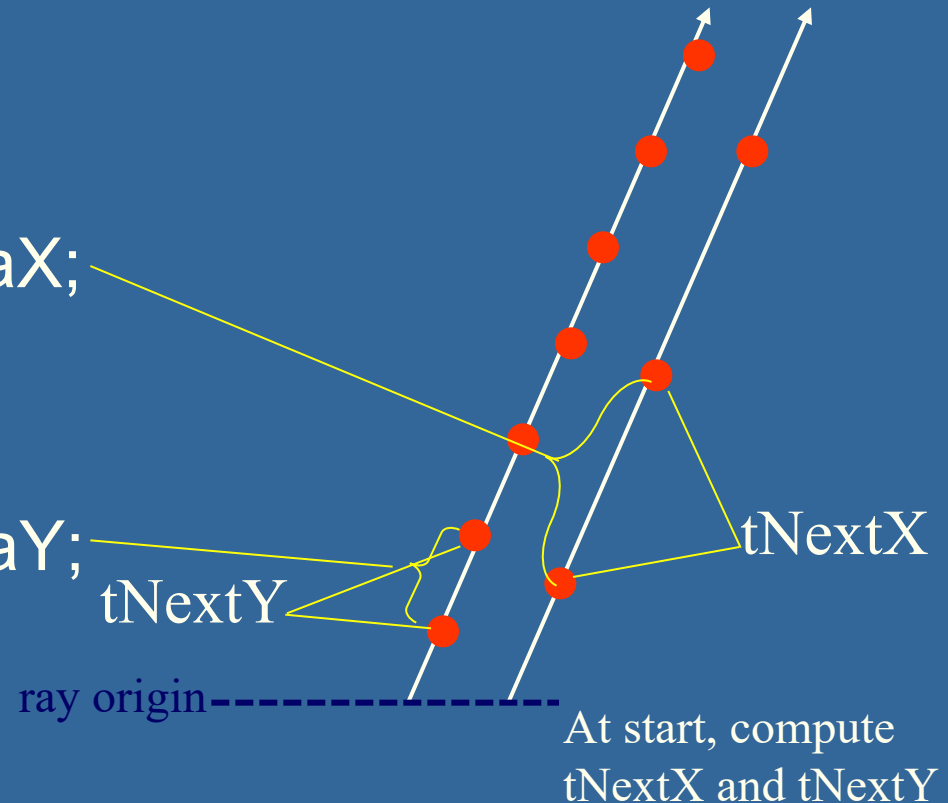
```
    tNextX += tDeltaX;
```

```
else
```

```
    Y = Y + stepY;
```

```
    tNextY += tDeltaY;
```

```
VisitVoxel(X, Y);
```



$tNextX$  = t-value at next step in x

$tNextY$  = t-value at next step in y

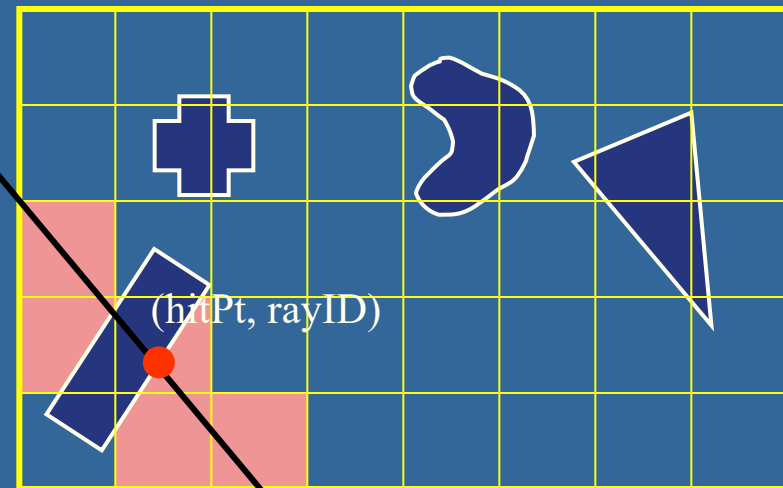
$stepX/Y = \pm 1$  depending on ray's slope

## 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 performance).
- 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.



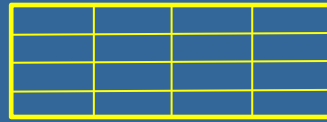
rayID

This is called  
mailboxing

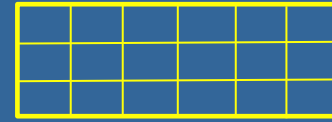
So then just fetch the hitpoint, stored with the object

# Choose a good grid resolution

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



4 x 4



3x6

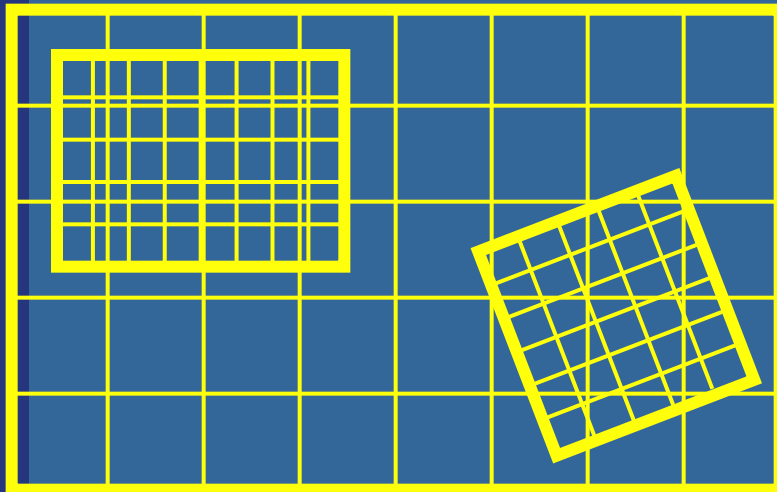
Using  $g = \sqrt[3]{n}$  is only good if root box is cubical!

- 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.

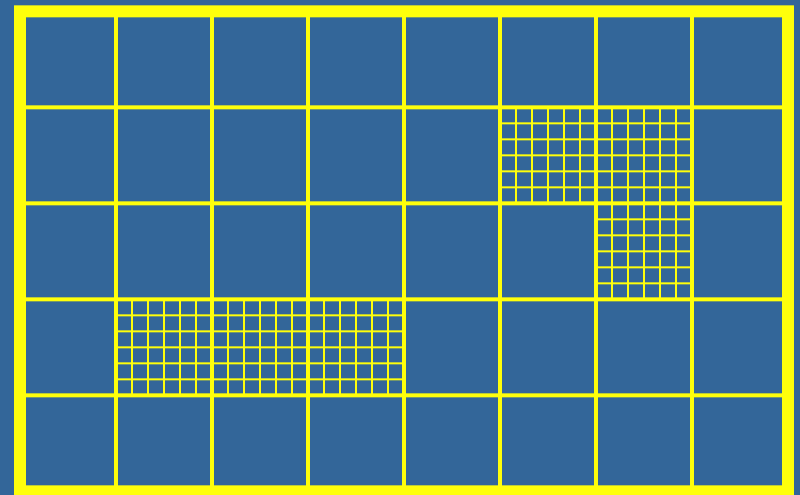


# Hierarchical and Recursive Grids

- 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

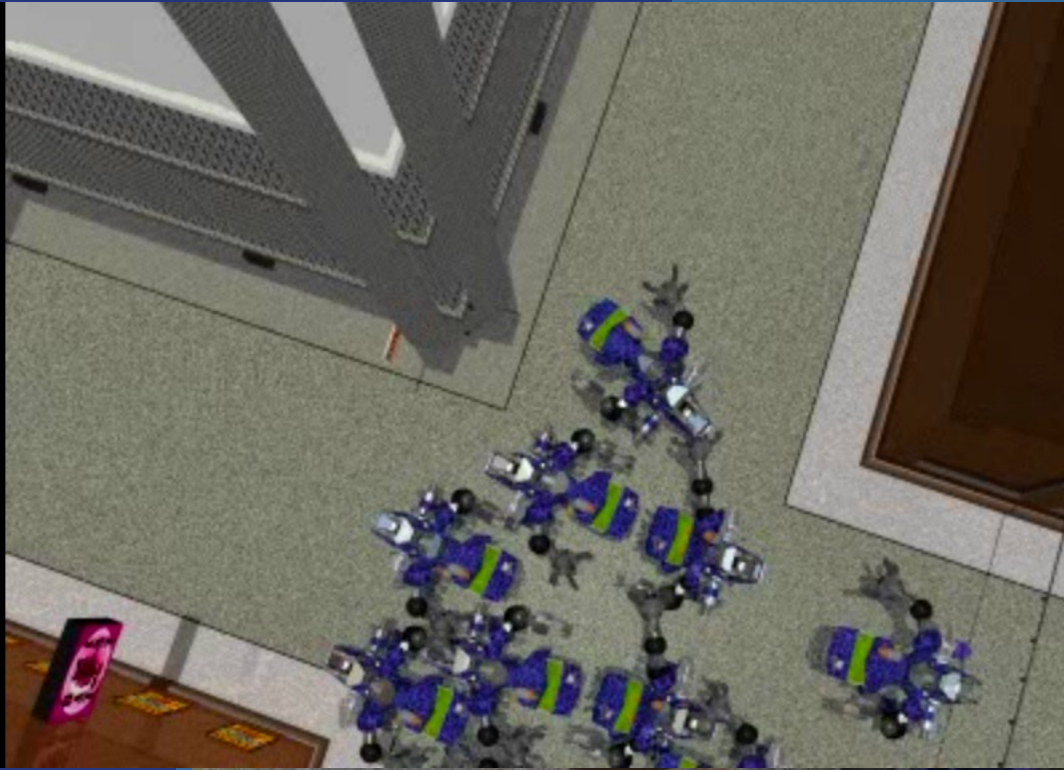


Hierarchical grid



Recursive grid

# Hierarchies to avoid the “Teapot in a stadium” problem



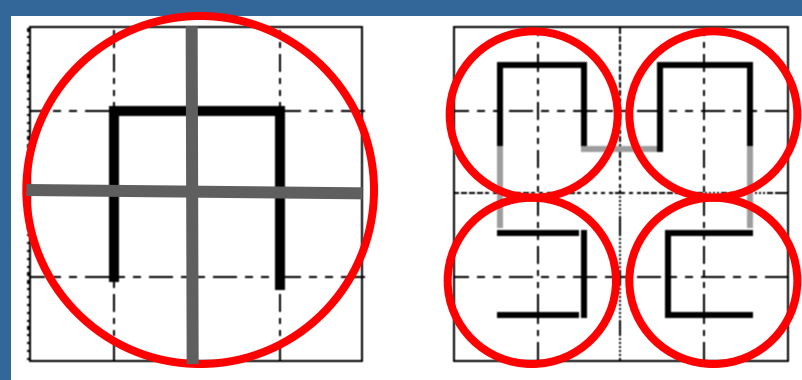
# Which spatial data structure is best?

- Depends on implementation, the type of scene, how complex shading, etc, etc.
- KD-trees:
  - Fastest to traverse, small memorywise, slow to build
- AABB-hierachies:
  - Fast to build, slower to traverse (not automatically in order along ray). Fast to update for moving rigid objects.
  - RTX – uses AABB hierarchy
  - CPU ray tracing: Split-BVHs (SBVH) currently the winner
- Grids
  - Fast to build, middle fast to traverse, typically needs to be hierarchial/recursive
  - Hierarchical grids can be very fast to update for moving rigid objects.
  - Recursive trees are surprisingly slower than hierarchical grids in terms of requiered computations. And slow to rebuild for moving objects.
- Sparse voxel octrees/DAGs:
  - Very fast to traverse.
  - DAGs have very small memory consumption
  - Slow to rebuild and does not store triangles.

# Cache awareness

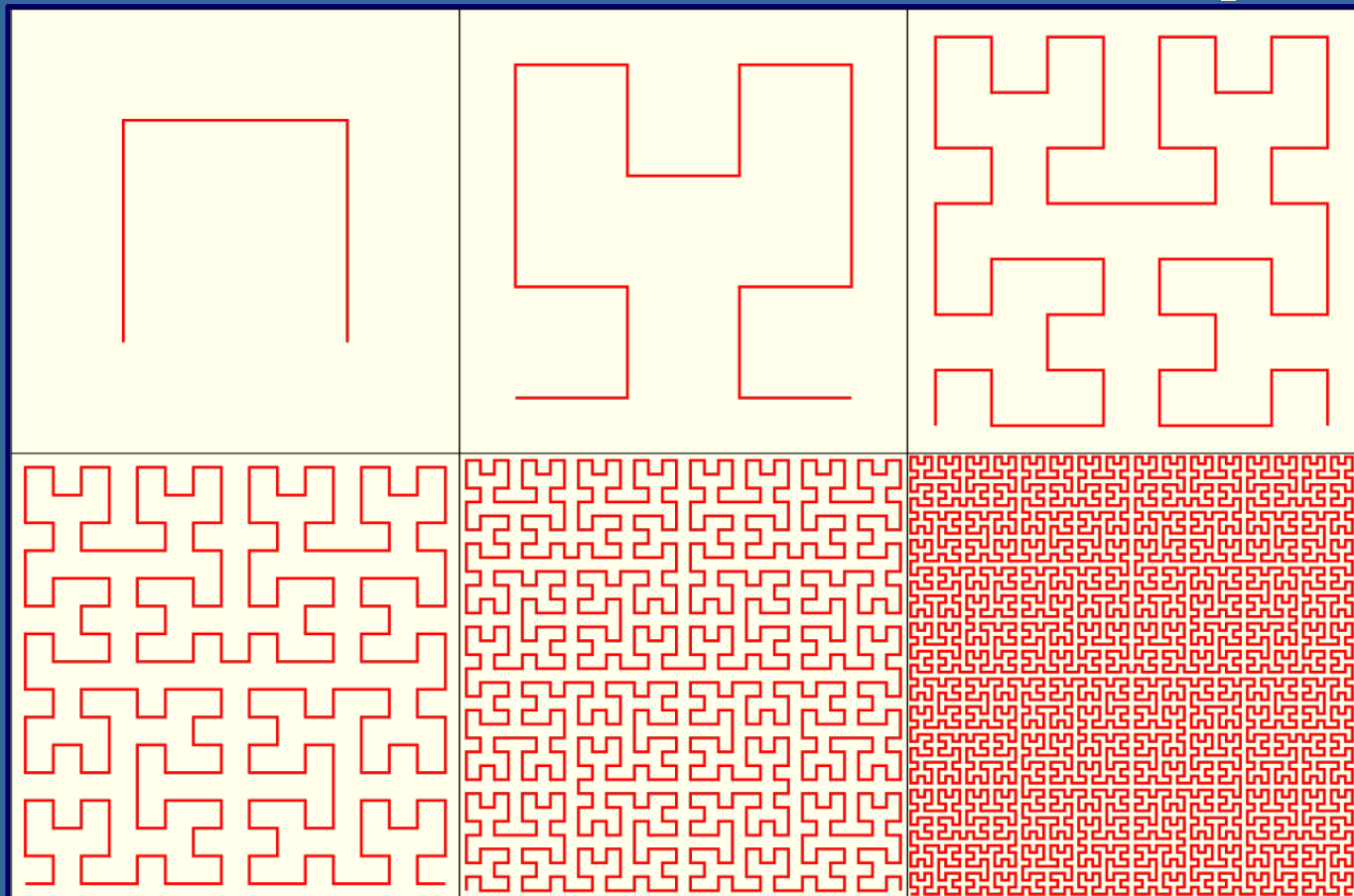
- To maximize cache locality, you can utilize that the next ray likely will access roughly the same memory locations since it will traverse roughly the same part of the tree and geometrical objects.
  - To maximize spatial locality, shoot the primary rays according to a Hilbert curve, instead of sequentially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

# Hilbert Curve



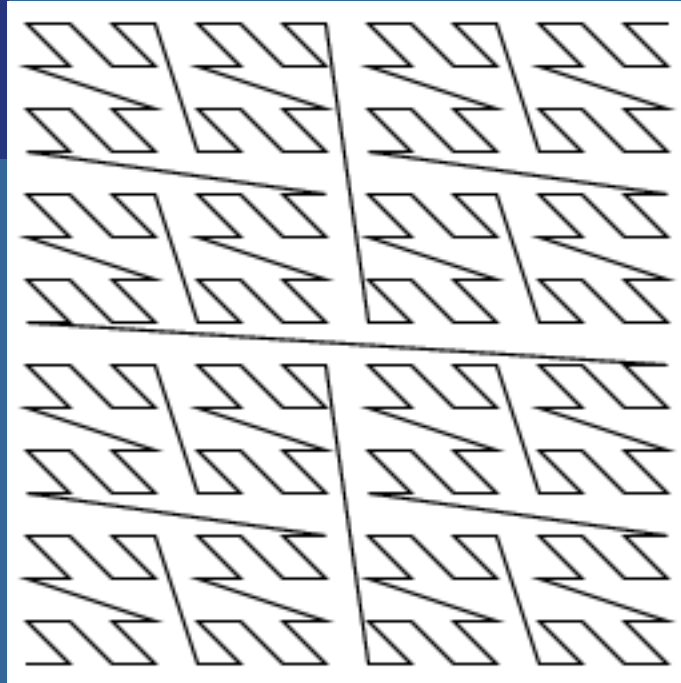
2x2 pixels

4x4 pixels



# Z-curve

or “Morton order”



Recipe to shoot primary rays in a Z-curve 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:

$$n = 1\ 1\ 0\ 1\ 0\ 1\ 1\ 0 = 214$$

→ e.g.,  $n = \dots y_3\ x_3\ y_2\ x_2\ y_1\ x_1\ y_0\ x_0$

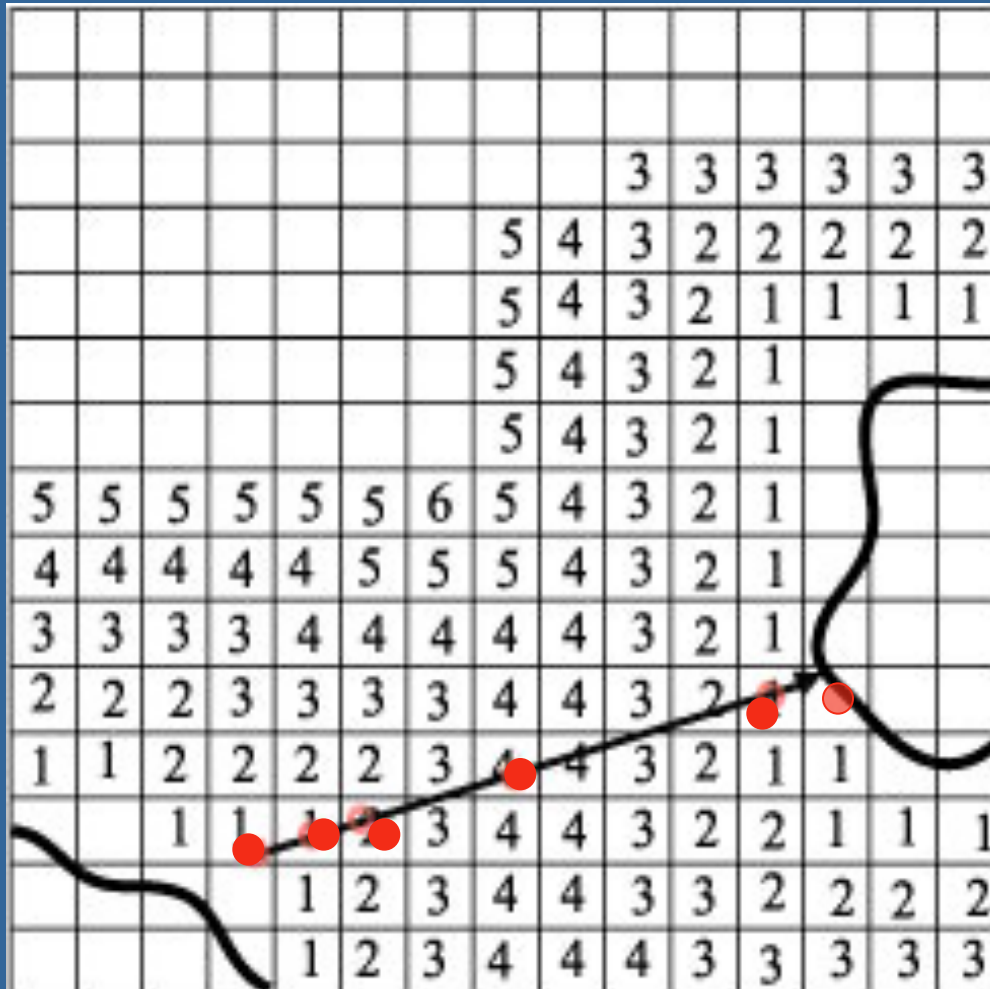
Then, the ray's x and y coordinates are:

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

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

Morton's trick is to take every 2<sup>nd</sup> bit for the x-coord and y-coord respectively

# Faster Grid Traversal using Proximity Clouds/Distance Fields



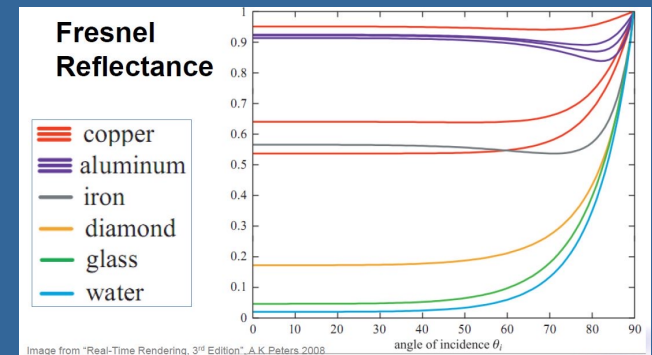
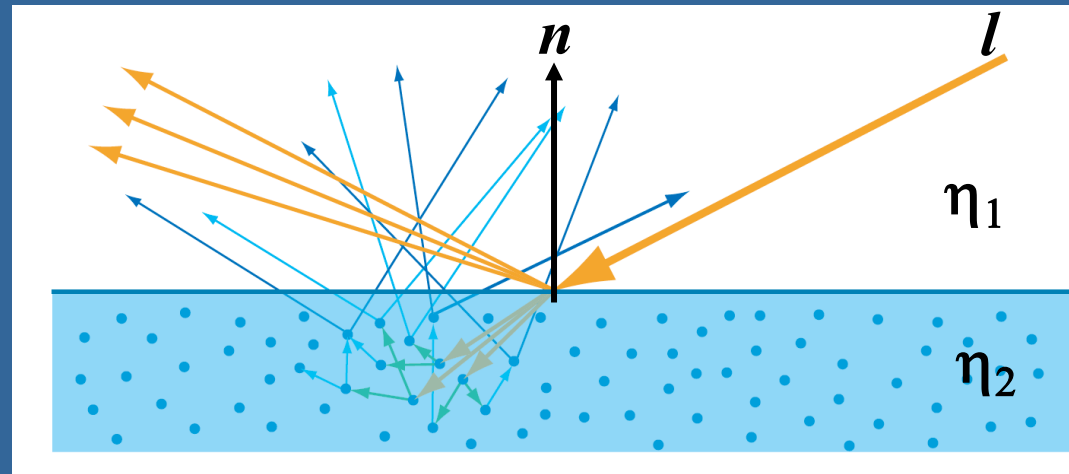
“Proximity Clouds – An Acceleration Technique for 3D Grid Traversal”, Daniel Cohen and Zvi Sheffer

**Demo**  
using SSE

# MATERIALS

- Types of material, and how light interacts
  - Dielectrics: Glass, plastic, stone, wood, water, ...
  - Metals (conductive)
  - Fresnel: can compute polarized, and unpolarized values for the light (in CG, we ignore polarization, often)

**Metall:** rgb-dependent Fresnel effect  
**Dielectrics:** not rgb-dependent.







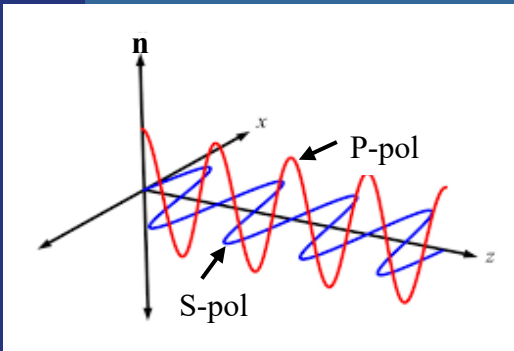
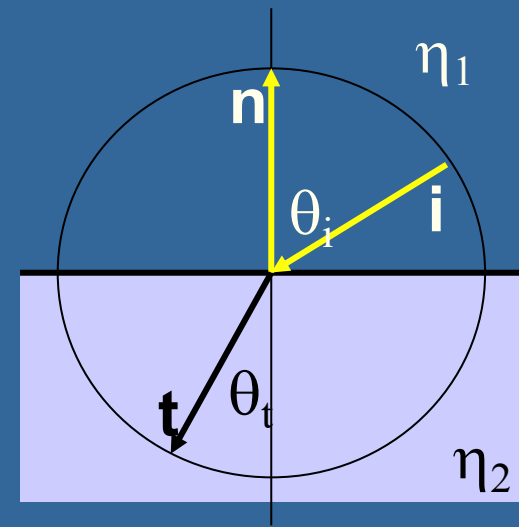
- At some places, the reflection is saturated (almost white), but mostly, it is clearly modulated by the copper color

- Plastic: highlight is just the color of the light source (i.e., not affected by the plastic's color)
- Metal: highlight has the reflection color also modulated by the metal color. (Fresnell values are rgb-dependent)



# Fresnel - polarization

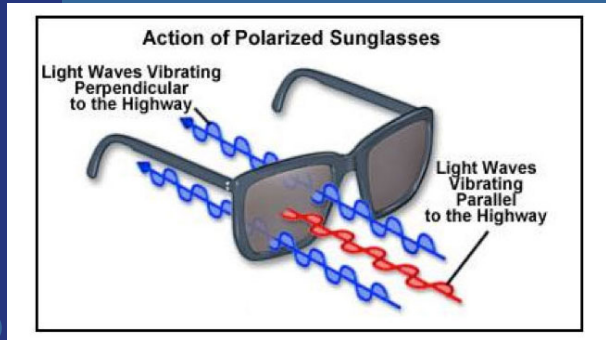
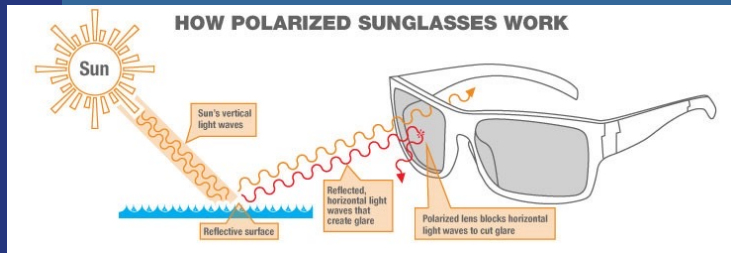
- The Fresnel effect for reflections/refractions is polarization dependent. Can compute polarized, and unpolarized values for the light (in CG, we ignore polarization, often)



$$R_s = \left| \frac{n_1 \cos \theta_i - n_2 \cos \theta_t}{n_1 \cos \theta_i + n_2 \cos \theta_t} \right|^2 = \frac{\left| n_1 \cos \theta_i - n_2 \sqrt{1 - \left(\frac{n_1}{n_2} \sin \theta_i\right)^2} \right|^2}{\left| n_1 \cos \theta_i + n_2 \sqrt{1 - \left(\frac{n_1}{n_2} \sin \theta_i\right)^2} \right|^2}$$

$$R_p = \left| \frac{n_1 \cos \theta_t - n_2 \cos \theta_i}{n_1 \cos \theta_t + n_2 \cos \theta_i} \right|^2 = \frac{\left| n_1 \sqrt{1 - \left(\frac{n_1}{n_2} \sin \theta_i\right)^2} - n_2 \cos \theta_i \right|^2}{\left| n_1 \sqrt{1 - \left(\frac{n_1}{n_2} \sin \theta_i\right)^2} + n_2 \cos \theta_i \right|^2}$$

Unpolarized light:  $R_{\text{eff}} = \frac{1}{2} (R_s + R_p)$



WITHOUT POLARIZED



WITH POLARIZED



WITHOUT POLARIZED



WITH POLARIZED



# Smooth dielectric materials

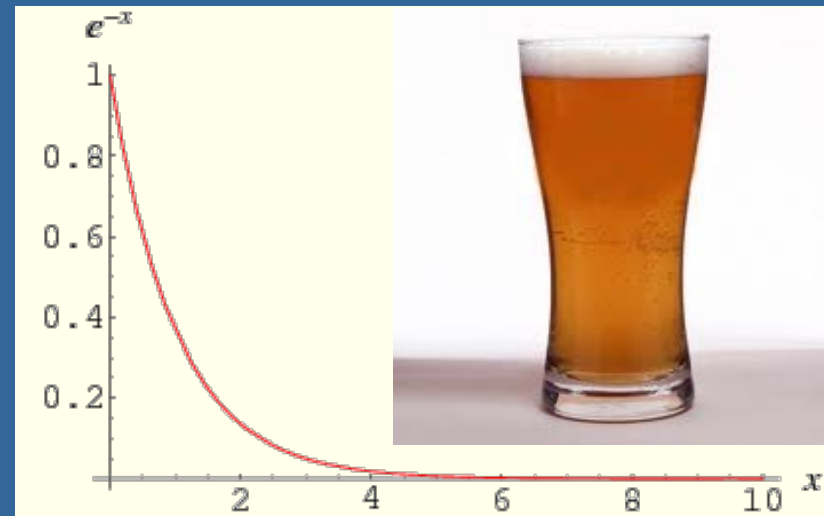
## Transparency and Homogeneous impurities

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

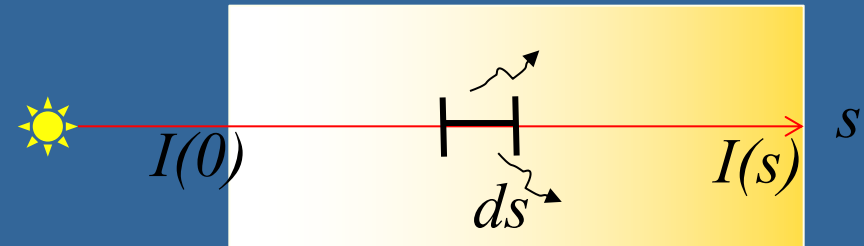
# Beer's Law

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



$$dI = -CIds$$

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



From August Beer, 1852

# Beer's law



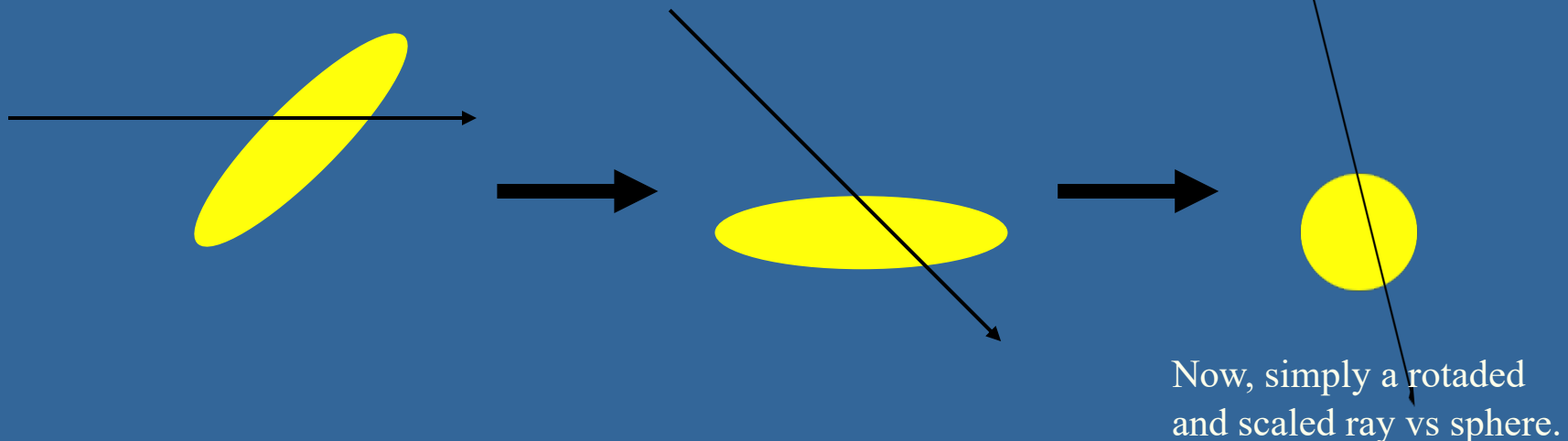
*The taller the glass, the darker the brew,  
The less the amount of light that comes through*

# 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

- Object-oriented programming
  - Makes it simple to add new geometrical objects from simpler ones. E.g., ellipsoid from scaled sphere.
- Just add a transform (TRS)
- The standard trick is not to apply the transform matrix to the object, but instead inverse-transform the ray



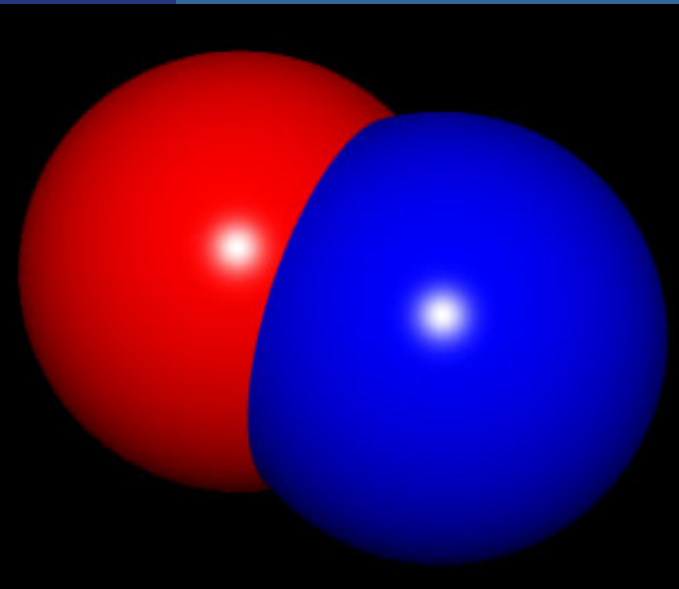


# Geometry: Constructive Solid Geometry (CSG)

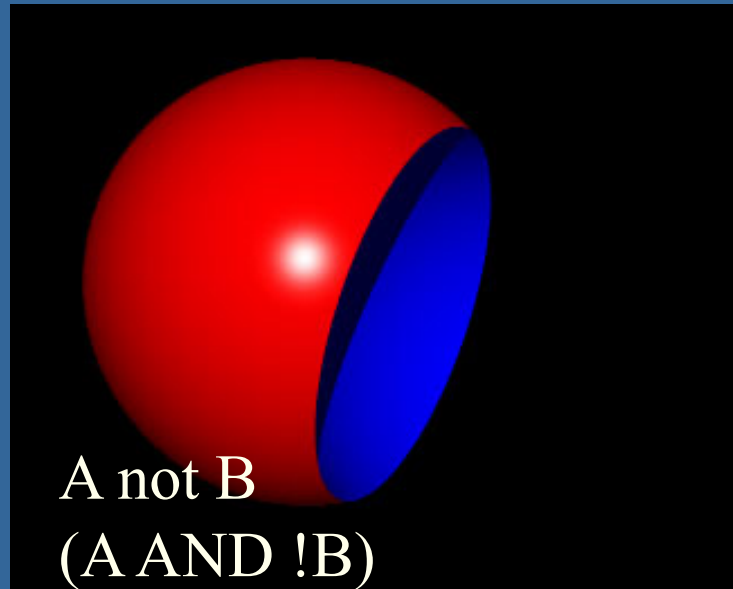
- Boolean operations on objects
  - Union (or)
  - Subtraction (A not B)
  - Xor
  - And
- Simple to implement
- Must find *all* intersections with a ray and an object
- Then do this for involved objects, and apply operators to found interval

# Geometry: Constructive Solid Geometry (CSG)

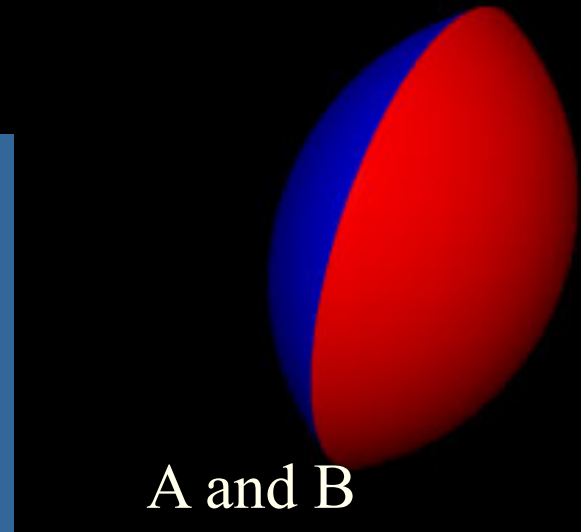
- Examples, operators:



A union B (OR)



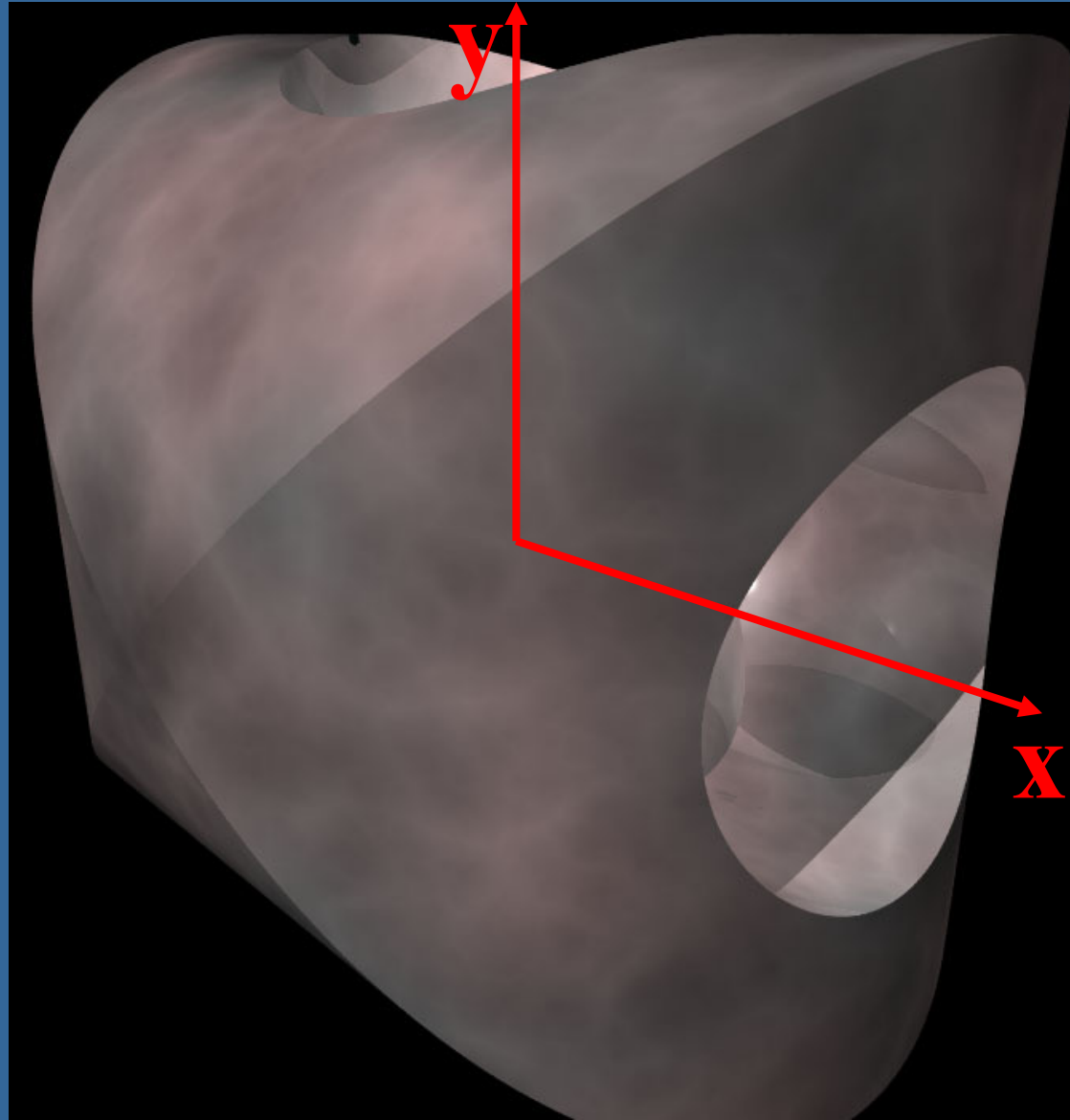
A not B  
(A AND !B)



A and B

# Geometry: Constructive Solid Geometry (CSG)

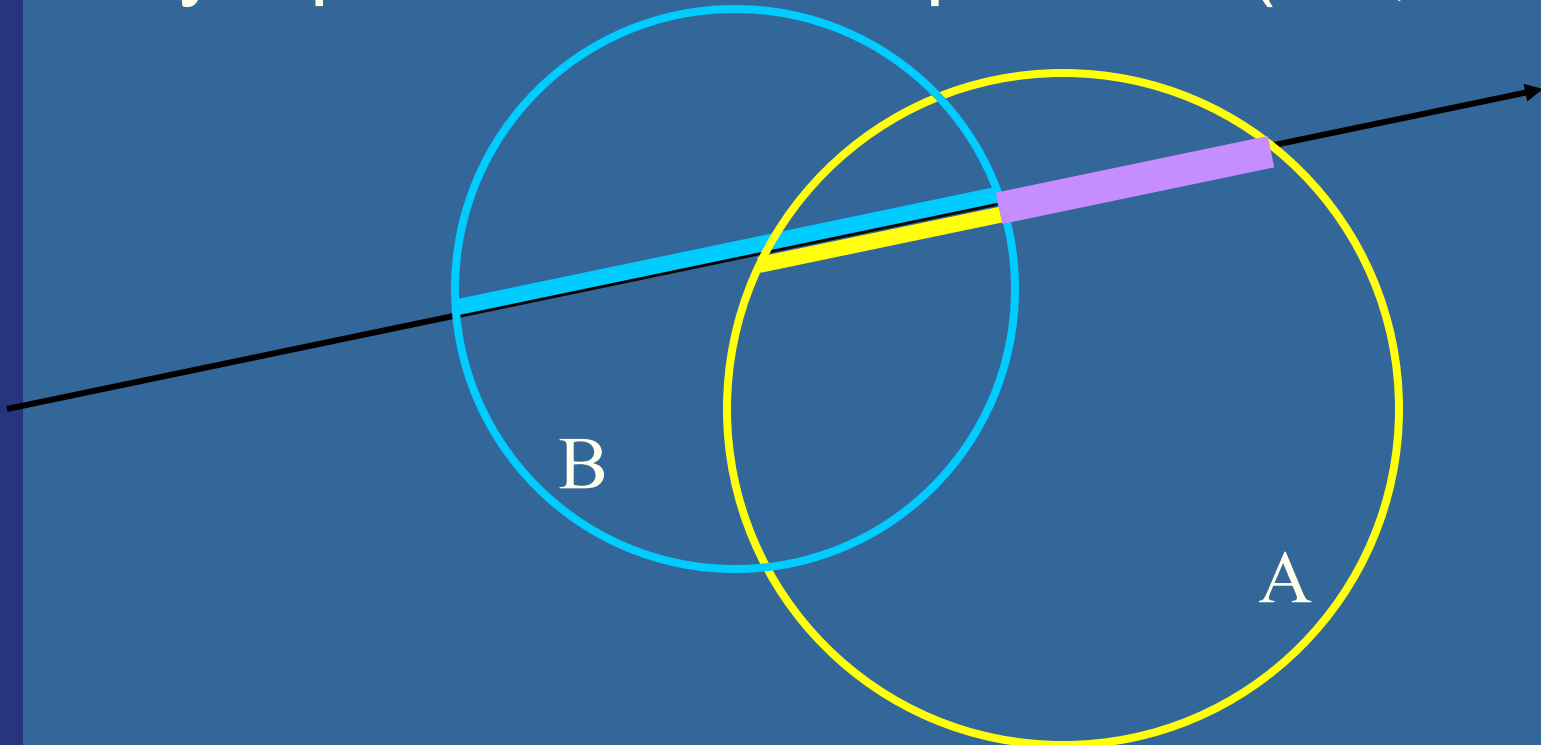
- Another example
- Done with 4 cylinders



# Constructive Solid Geometry (CSG)

## How to implement

- Try: sphere A minus sphere B (i.e., A not B)



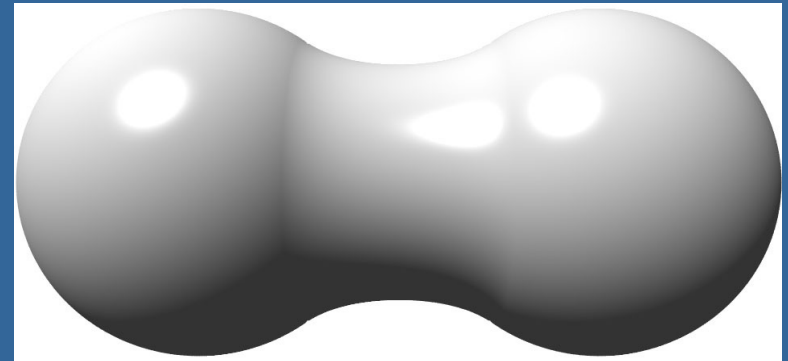
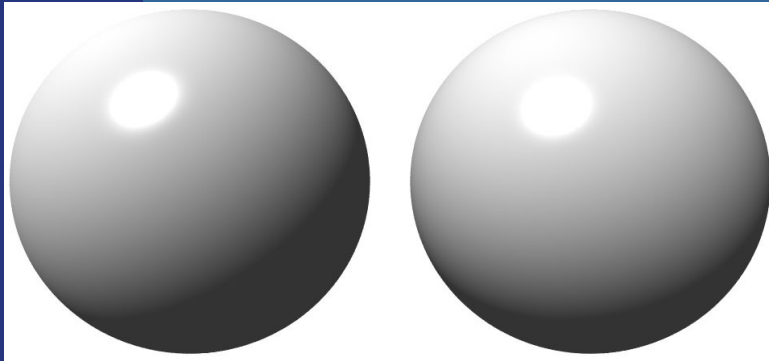
- In summary: find both entry and exit points on both spheres. Such two points on a sphere is an interval (1D). Apply the operator on these intervals

# CSG

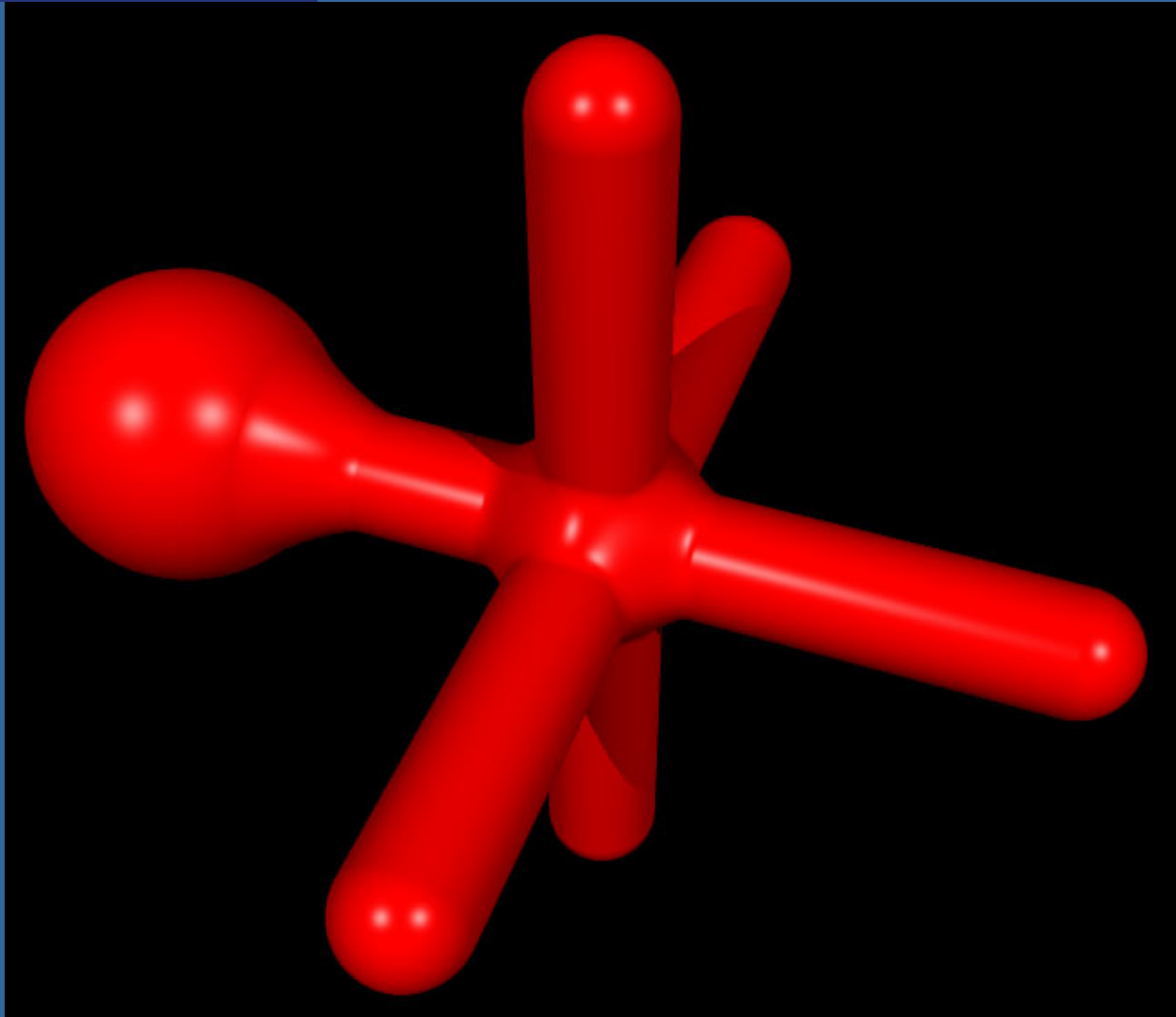
- Works on any geometrical object, as long as you can find **all** intersection point along a line
  - So, be careful with optimizations...
- And objects should be closed
  - Example: put caps on cylinder.

# 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

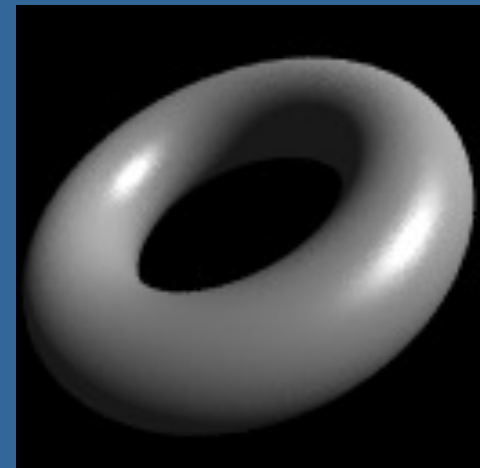
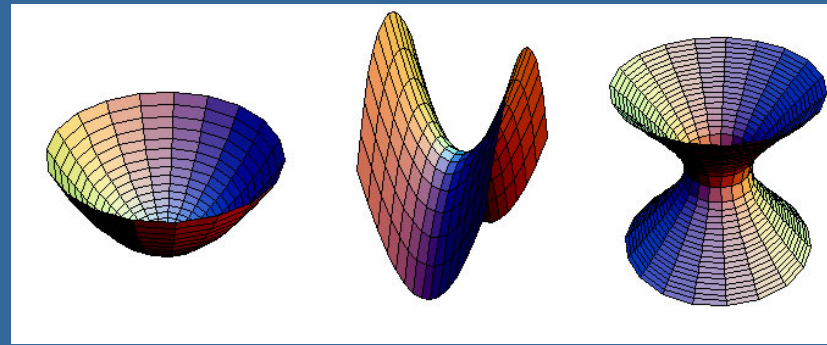


# Blob example



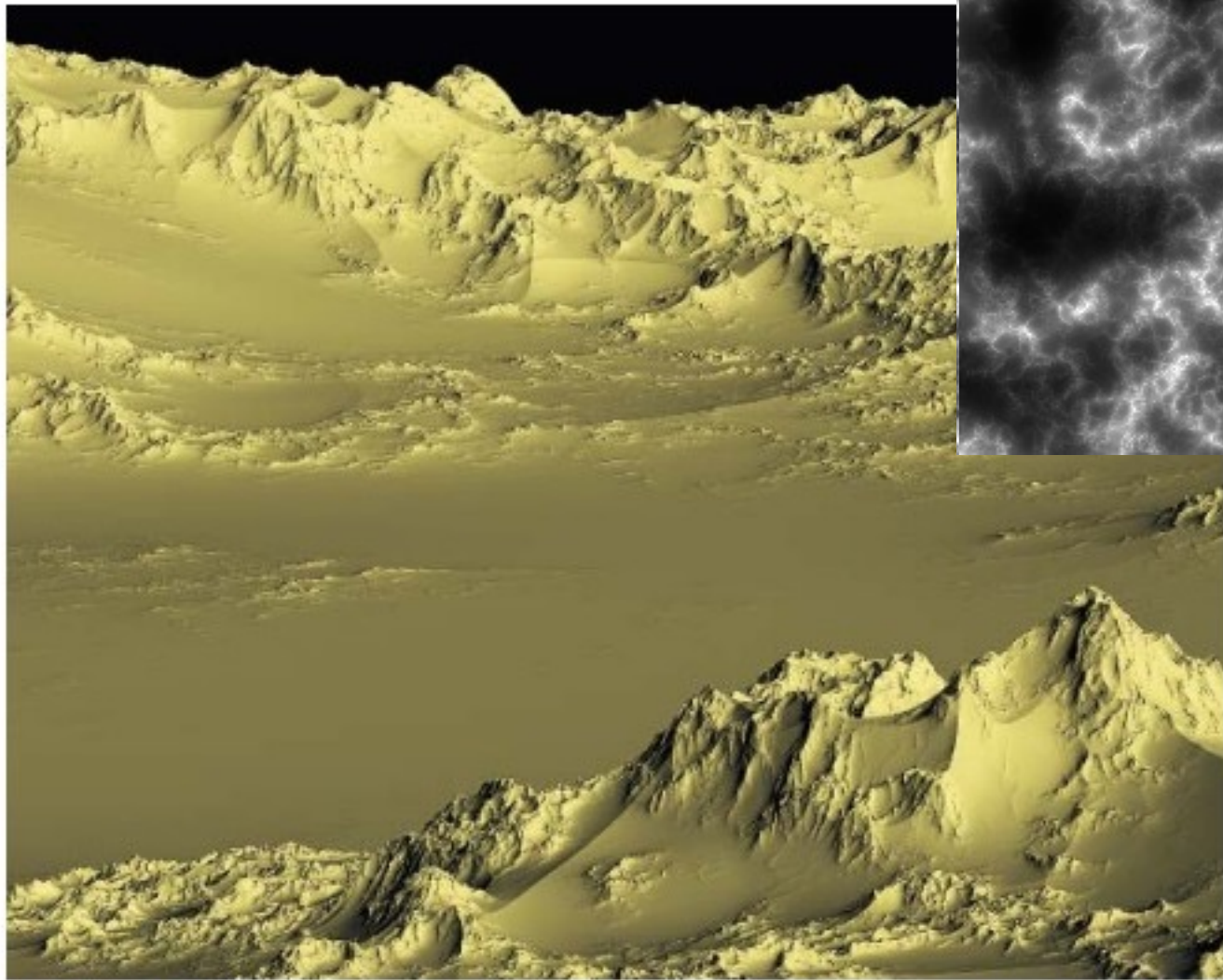
# Geometry

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



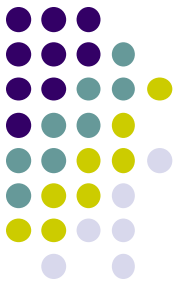


# Fractals

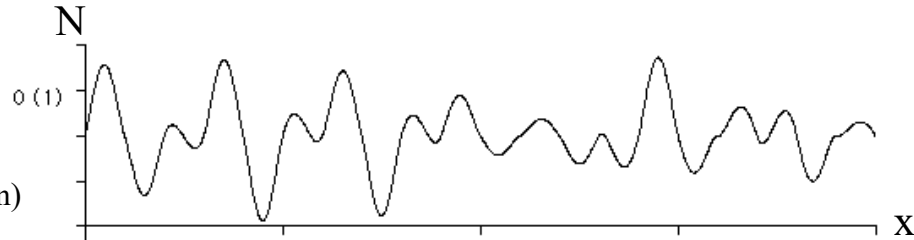


**FIGURE 17.4** The very first QAEB-traced terrain. The terrain model is the “ridged multifractal” function described in Chapter 16. Copyright © F. Kenton Musgrave.

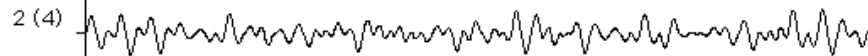
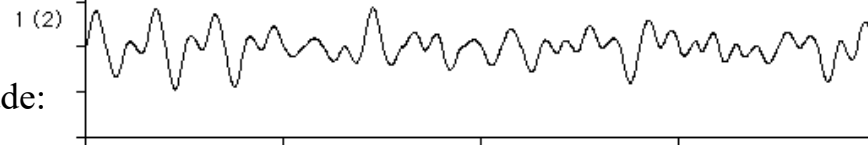
# Perlin Noises in 1-D



Noise signal with certain frequency and amplitude:  
(E.g., use random-number generator and spline interpolation)



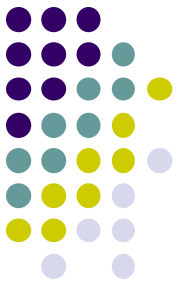
Next octave: ~double frequency, ~half amplitude:



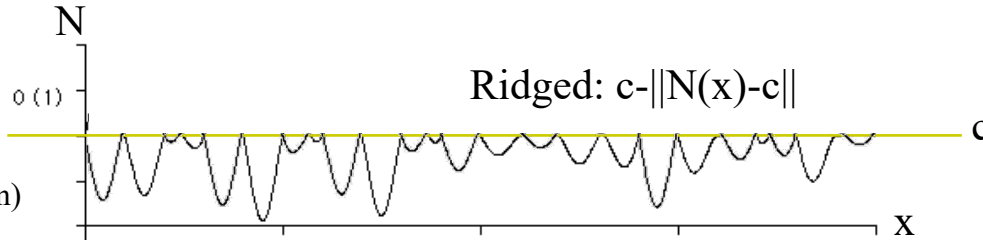
Adding gives Fractal Noise:



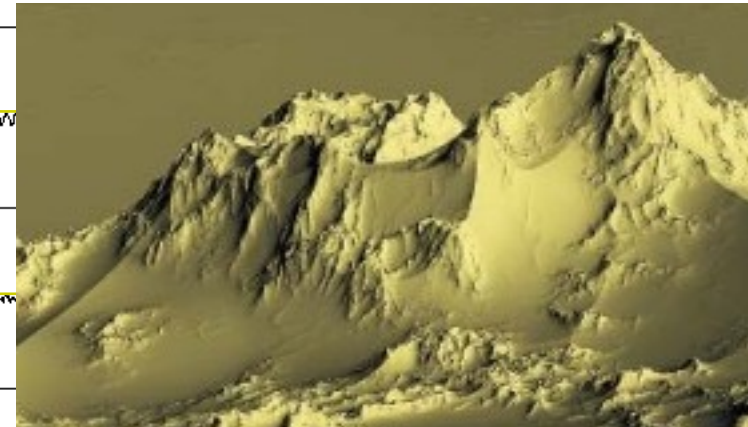
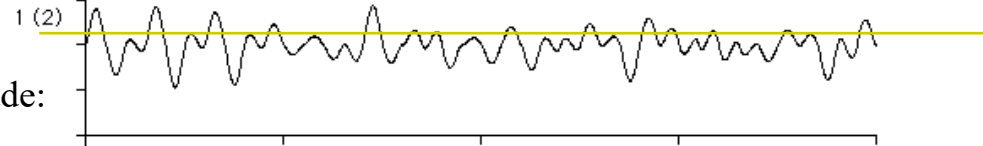
# Perlin Noises in 1-D



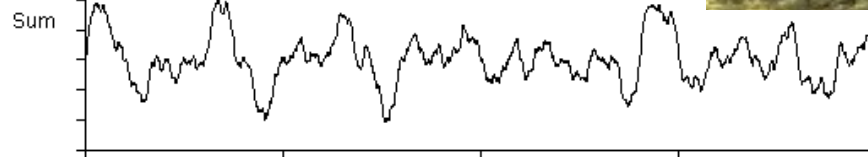
Noise signal with certain frequency and amplitude:  
(E.g., use random-number generator and spline interpolation)



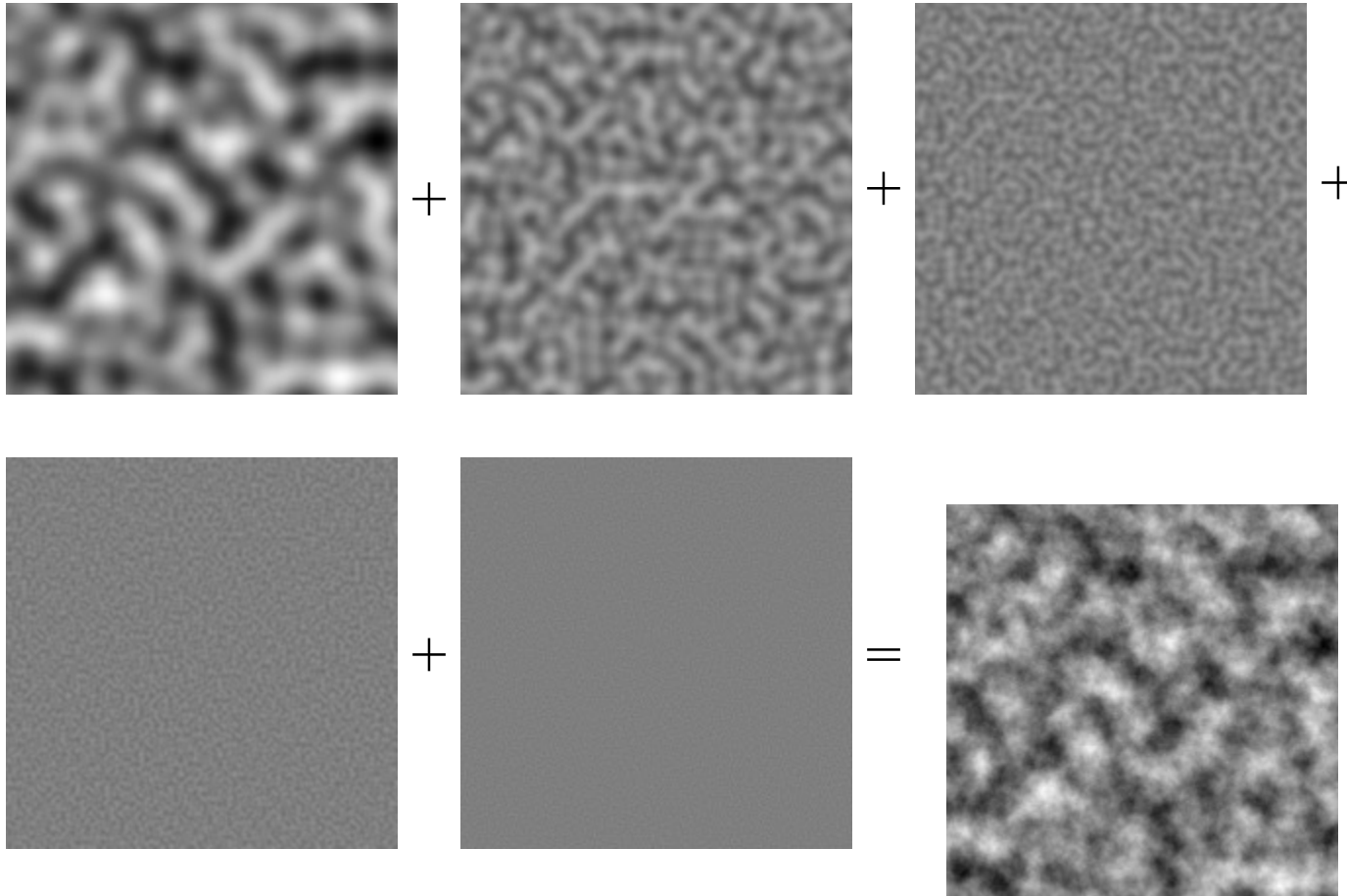
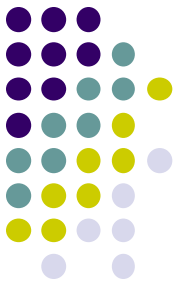
Next octave: ~double frequency, ~half amplitude:



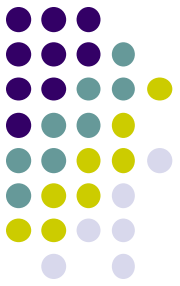
Adding gives Fractal Noise:



# Perlin Noises in 2-D

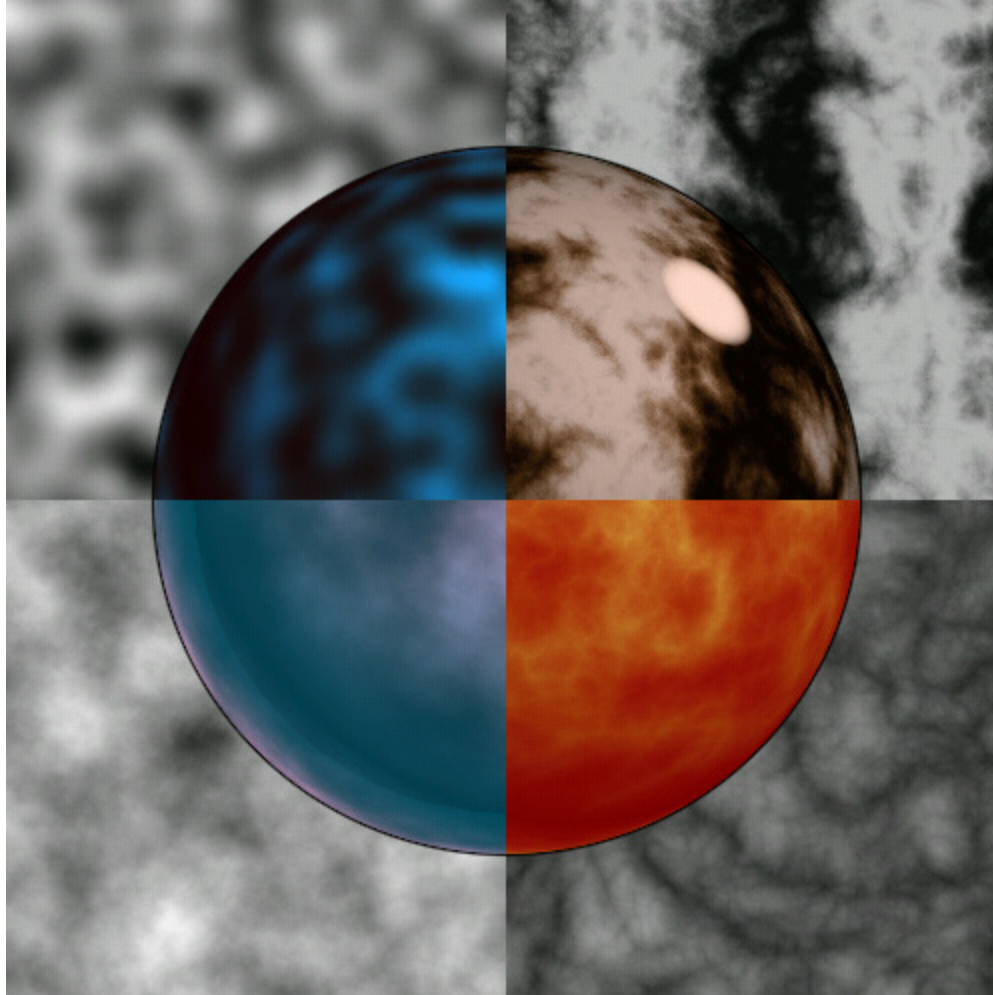


# Weighted Sums



## Noise (1 octave):

- 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

Terragen v4, free for students



# Iñigo Quilez - Brave

Worked on procedural terrain, moss, grass, etc.



Realtime OpenGL with ambient occlusion.

<https://www.iquilezles.org/www/articles/multiresaocc/multiresaocc.htm>

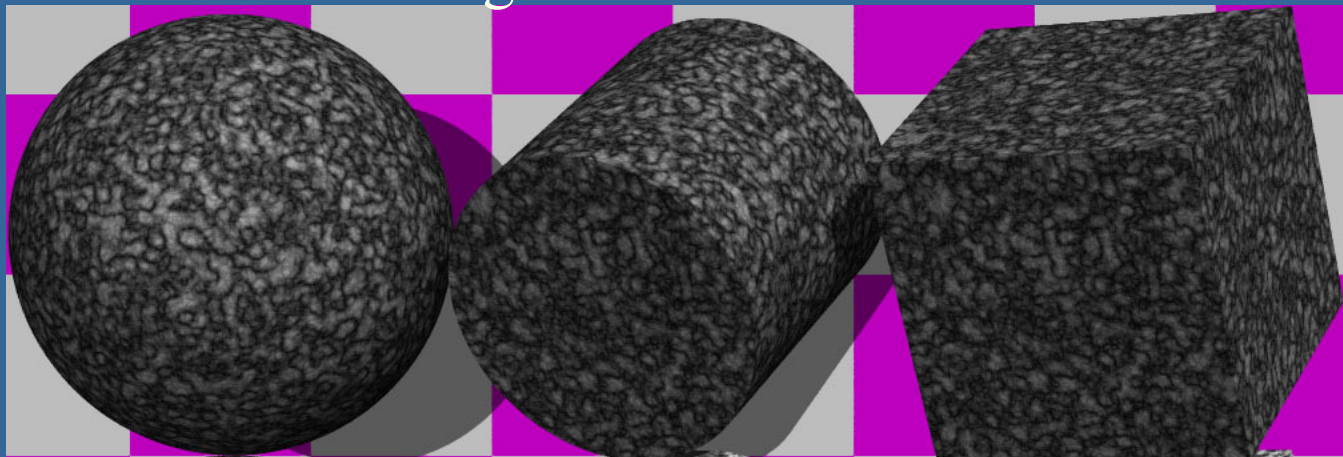
# More fractal examples...

WebGL examples:

- <https://www.shadertoy.com/view/MdfGRX>
- Ladybug (fully procedural, open in Chrome):  
<https://www.shadertoy.com/view/4tByz3>



Procedural texturing:



Texturing and Modeling – a procedural approach, by Perlin, Musgrave, Ebert...

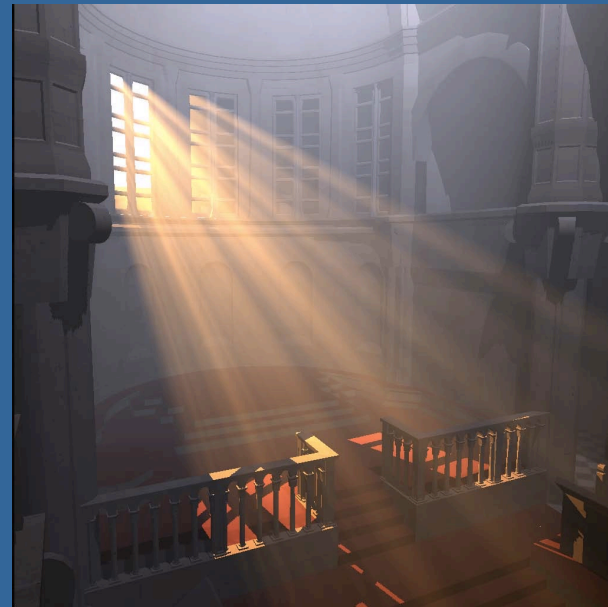


# Optics

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



Participating media



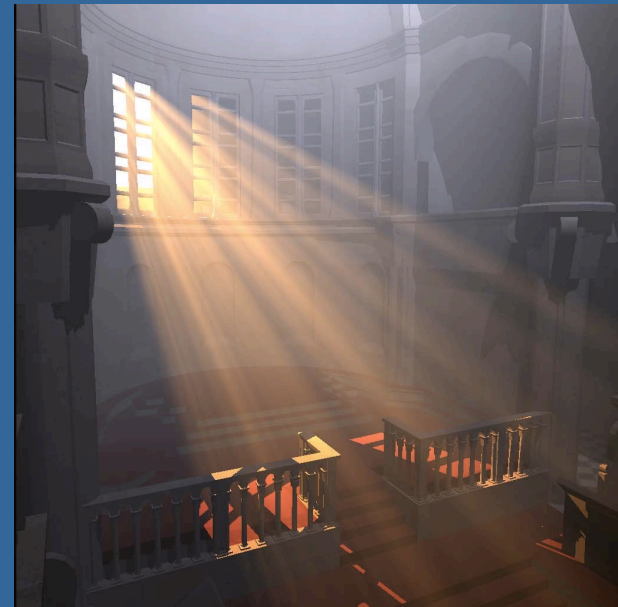
... also for real-time rendering

# More effects

- You can add
  - Optics
    - Depth of field
  - Motion blur
    - i.e., temporal antialiasing
  - Light scattering in participating media
    - Fog
    - Smoke
    - Clouds
    - Shafts of Light
    - ...



Participating media

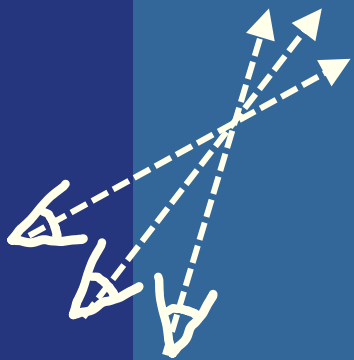
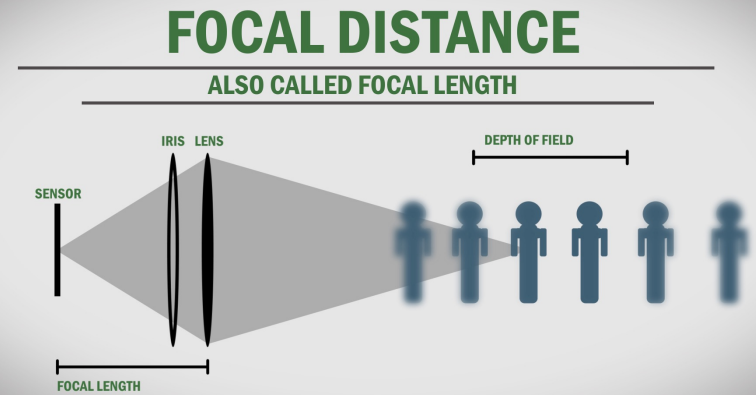


... also for real-time rendering

# Optics

- Depth-of-field

- Add more samples on a virtual camera lens



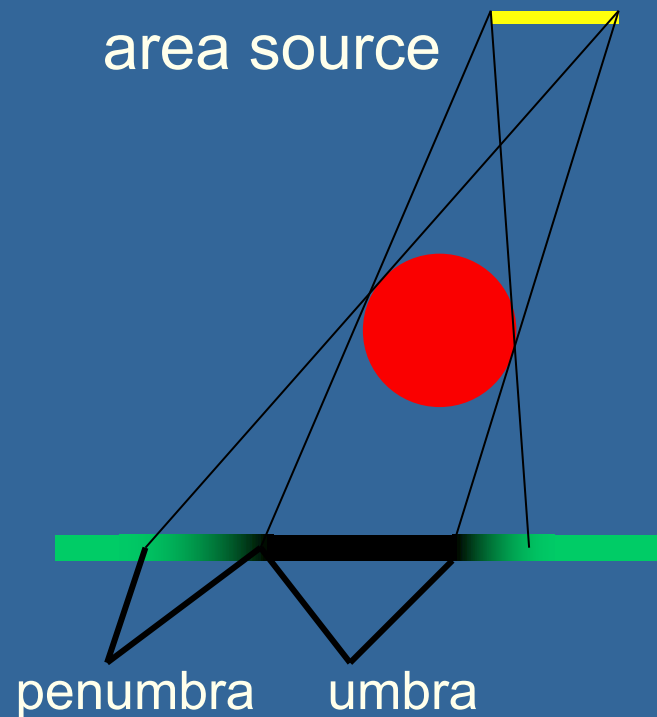
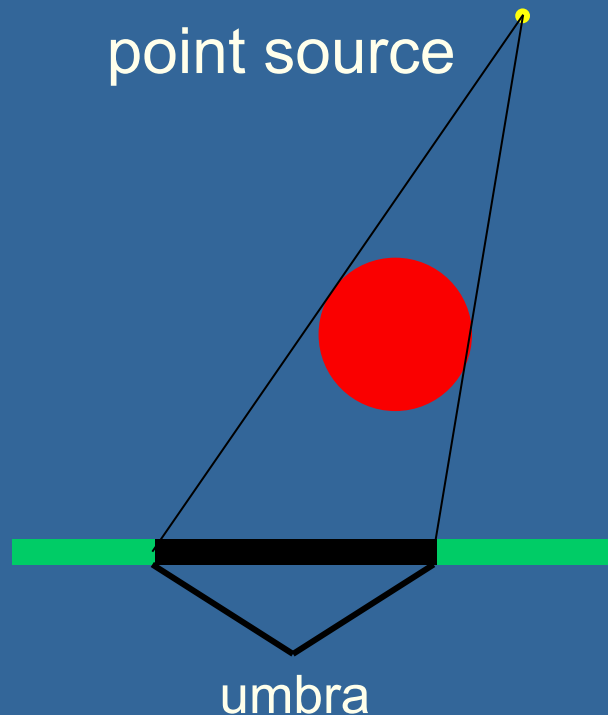
# Soft shadows

- Soft shadows are typically more realistic than hard shadows
- Examples:

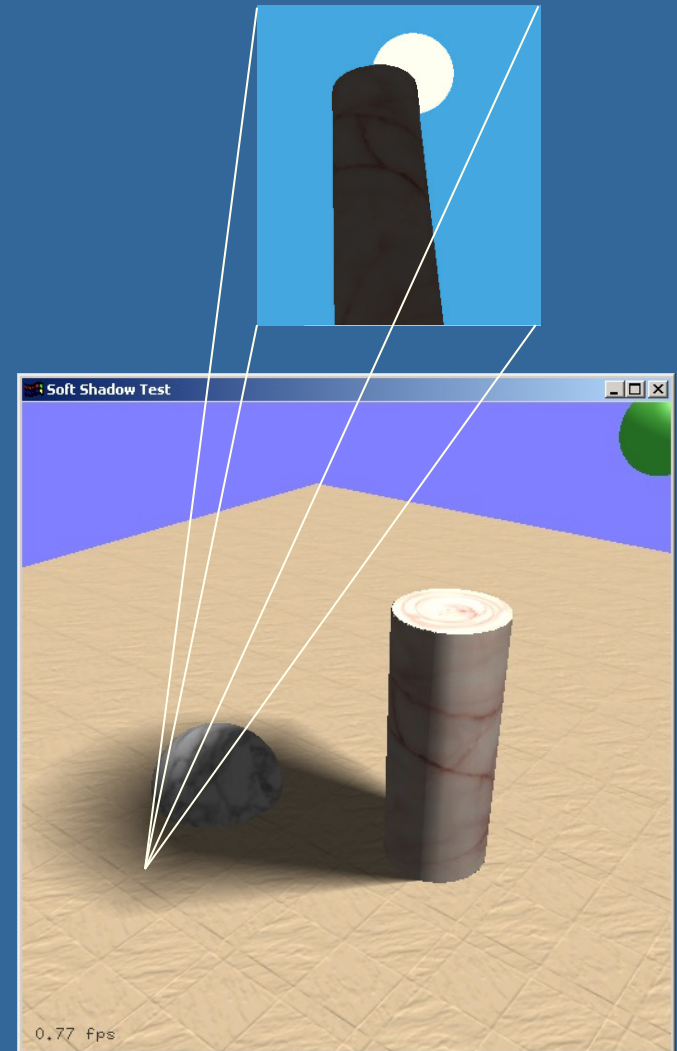
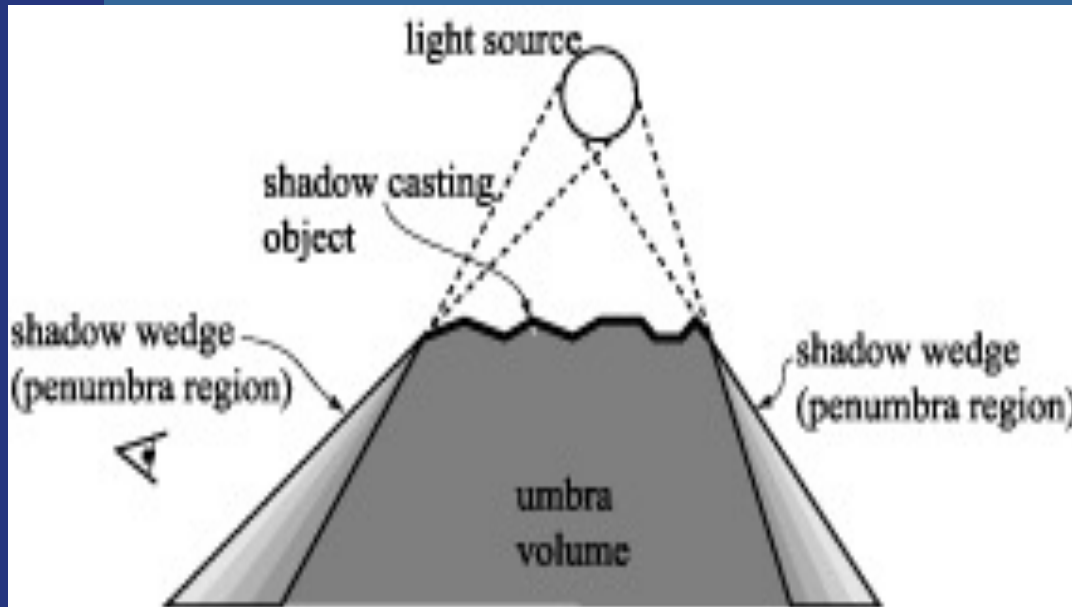


# Soft shadows

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

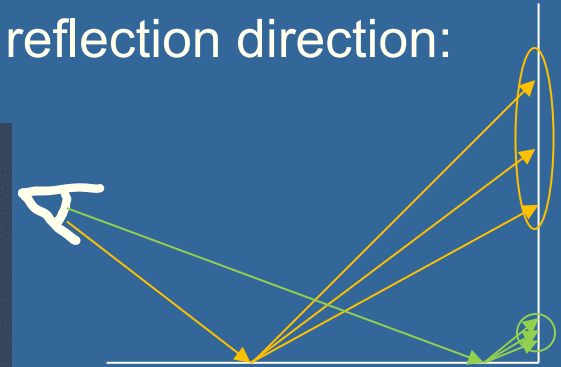
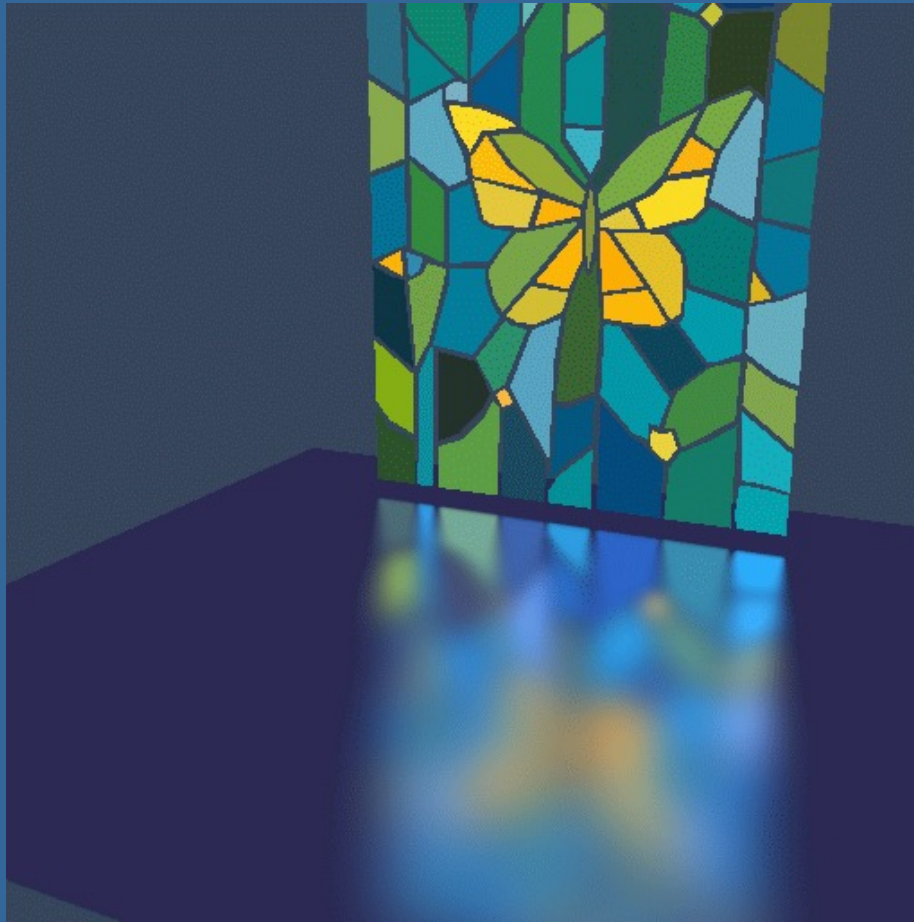


# Example



# Glossy (blurry) reflections

- Trace many reflection directions
  - Each perturbed slightly from the main reflection direction:



Do the same  
with the transmission  
vectors

# 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
  - However
    - 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, AABSP-tree (kd-tree), polygon-aligned BSP tree, Sphere/AABB/OBB-tree,
- What's a
  - skip-pointer tree?
  - Shadow cache?
  - Kd-tree? (=AABSP with fixed split-plane order) + A MOVIE...
- Describe ray/BVH intersection test
- The Fresnel-effect: metal vs dielectrics
  - How does dielectrics (e.g., plastic/glass/water/air) behave?
  - How does metal behave?
- Describe how ray trace using Constructive Solid Geometry

THE END

# Guardians of the Galaxy 2

## – Ego's Planet

- Sierpinsky gasket

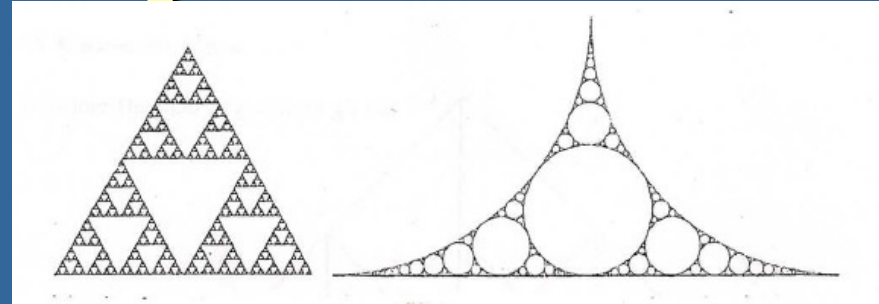
- Ego's world constructed by fractals (partially)

- By procedural generation and boolean operators
    - Some geometry triangulated by photogrammetry
    - Photogrammetry = 3D reconstruction of a scene just from many photographs of it.

- James Gunn – director

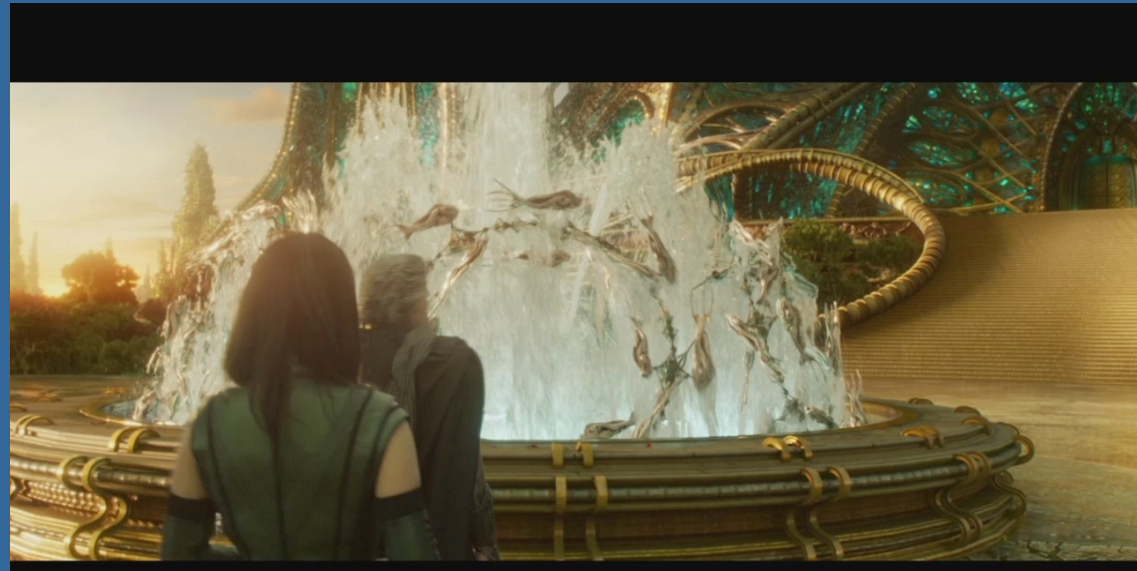
- Fish fountain

- Victoria Alonso,  
now: President of Physical,  
Post Production, VFX and  
Animation at Marvel Studios



Sierpinsky triangle

Apollonian circle packing



Start Movie  
(12 min)