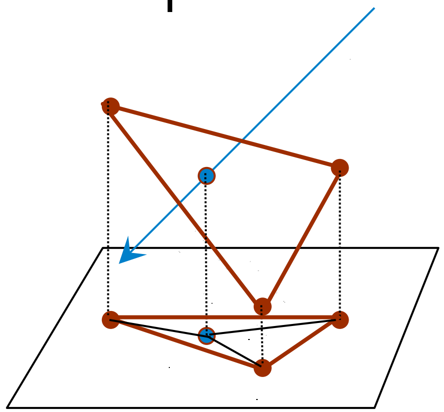
Interscetion Ray—Triangle

[Badouel 1990]

- Use same method like ray—polygon; or
- Be clever: use barycentric coords + projection
- Intersect ray with plane (implicit form) → t → point in space
- Project point & triangle on coord plane
- Compute baryzentric coords of 2D point
- baryzentric coords of 2D point (α,β,γ) =
  baryzentric coords of orig. 3D point!



- 3D point is in triangle ⇔ $\alpha,\beta,\gamma > 0$, $\alpha+\beta+\gamma = 1$
- Alternative method: see Möller & Haines "Real-time Rendering"
  - Code: http://jgt.akpeters.com/papers/MollerTrumbore97/
- Faster method, if intersection point not needed [Segura & Feito]

Alternative Intersection Method for Ray—Triangle

[Möller]

- Line equation: $X = P + t \cdot \mathbf{d}$
- Plane equation: $X = A + r \cdot (B - A) + s \cdot (C - A)$
- Equate both:

$$-t \cdot \mathbf{d} + r \cdot (B - A) + s \cdot (C - A) = P - A$$

- Write in matrix form:

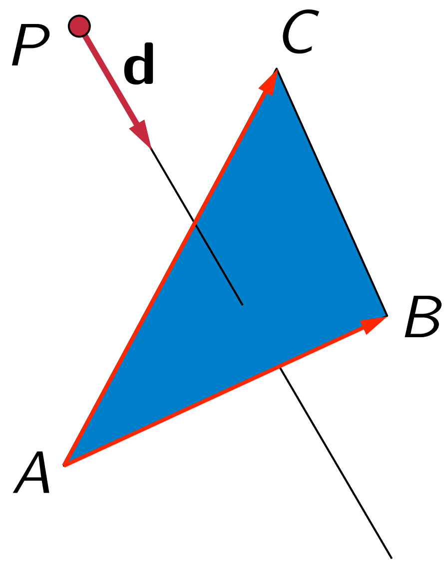
$$\begin{pmatrix} \vdots & \vdots & \vdots \\ -\mathbf{d} & \mathbf{u} & \mathbf{v} \\ \vdots & \vdots & \vdots \end{pmatrix} \begin{pmatrix} t \\ r \\ s \end{pmatrix} = \mathbf{w}$$

where

$$\mathbf{u} = B - A$$

$$\mathbf{v} = C - A$$

$$\mathbf{w} = P - A$$



- Use Cramer's rule:

$$\begin{pmatrix} t \\ r \\ s \end{pmatrix} = \frac{1}{\det(-\mathbf{d}, \mathbf{u}, \mathbf{v})} \cdot \begin{pmatrix} \det(\mathbf{w}, \mathbf{u}, \mathbf{v}) \\ \det(-\mathbf{d}, \mathbf{w}, \mathbf{v}) \\ \det(-\mathbf{d}, \mathbf{u}, \mathbf{w}) \end{pmatrix}$$

$$\det(\mathbf{a}, \mathbf{b}, \mathbf{c}) = \mathbf{a} \cdot (\mathbf{b} \times \mathbf{c}) = (\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c}$$

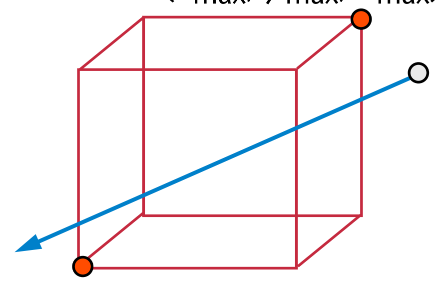
$$\begin{pmatrix} t \\ r \\ s \end{pmatrix} = \frac{1}{(\mathbf{d} \times \mathbf{v}) \cdot \mathbf{u}} \cdot \begin{pmatrix} (\mathbf{w} \times \mathbf{u}) \cdot \mathbf{v} \\ (\mathbf{d} \times \mathbf{v}) \cdot \mathbf{w} \\ (\mathbf{w} \times \mathbf{u}) \cdot \mathbf{d} \end{pmatrix}$$

- Cost: 2 cross products + 4 dot products
- Yields both line parameter t and barycentric coords of hit point
- Still need to test whether s,t in [0,1] and s+t <= 1

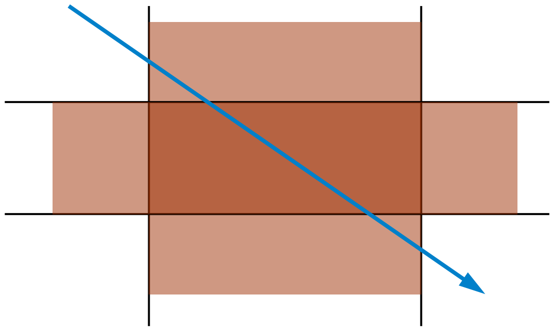
Intersection of Ray and Box

- Box (Quader) is most important bounding volume!
- Here: just axis-aligned boxes (AABB = *axis-aligned bounding box*)
- AABB is usually specified by two extremal points

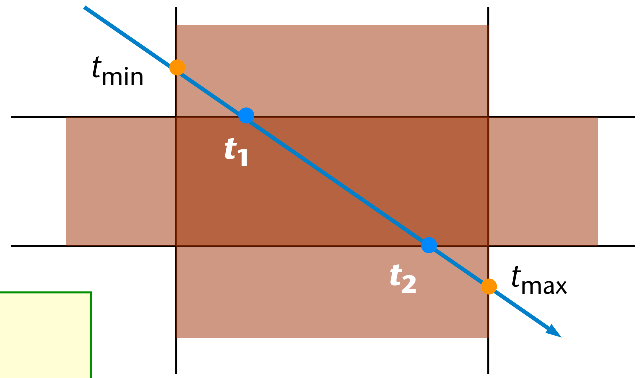
  $$(x_{min}, y_{min}, z_{min}) \text{ and } (x_{max}, y_{max}, z_{max})$$



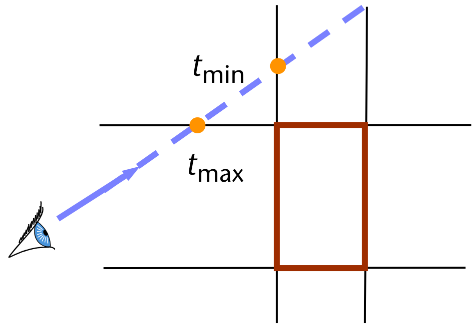
- Idea of the algorithm:
  - A box is the intersection of 3 *slabs* (*slab* = subset of space enclosed between two parallel planes)
  - Each slab cuts away a specific interval of the ray
  - So, successively consider two parallel (= opposite) planes of the box

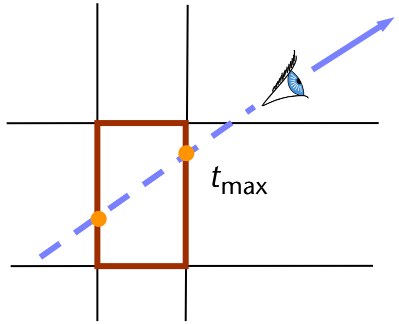


- The algorithm:



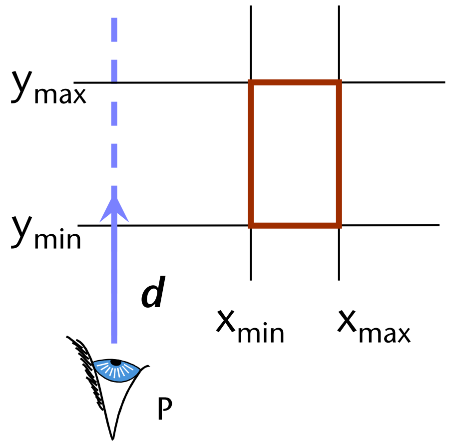

```
let tmin = -∞ , tmax = +∞
loop over all (3) pairs of planes:
  intersect ray with both planes
  → t1, t2
  if t2 < t1:
    swap t1, t2
  // now t1 < t2 holds
  tmin ← max(tmin, t1 )
  tmax ← min(tmax, t2 )
// now: [tmin,tmax] = interval inside box
if tmin > tmax → no intersection
if tmax < 0    → no intersection
```





Remarks

- Optimization: both planes of a slab have the same normal ⟶ can save one dot product
- Remark: the algorithm also works for "tilted" boxes (called *OBBs = oriented bounding boxes*)
- Further optimization: if AABB, exploit fact that normal has exactly one component = 1, other = 0!
- Warning: "shit happens"
  - Here: test for parallel situations!
  - In case of AABB:




```
if |dx| < ε:
  if Px < xmin || Px > xmax:
    ray doesn't intersect box
  else:
    t1, t2 = ymin, ymax    // or vice versa!
```

Intersection Ray—Sphere

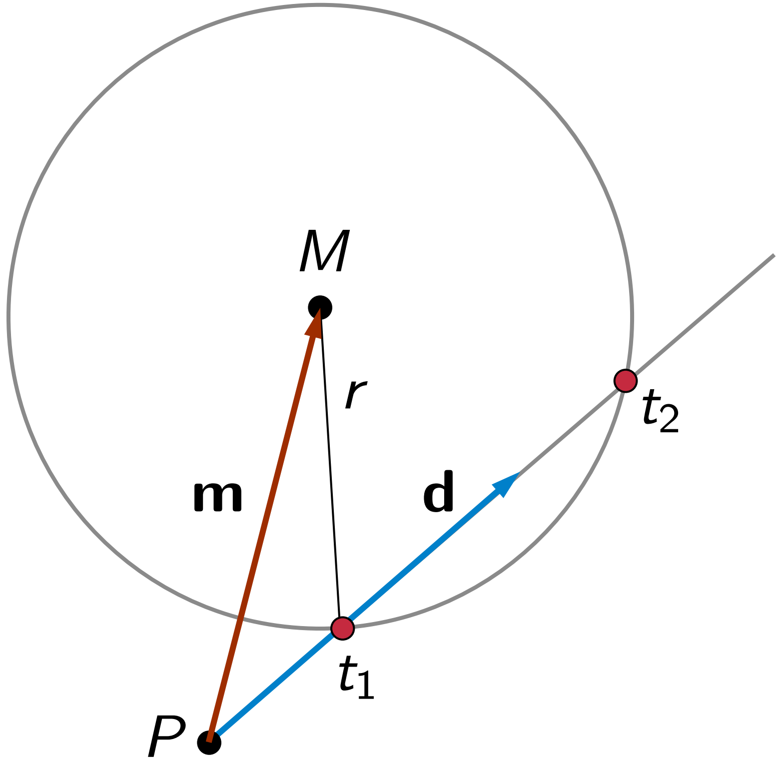
- Assumption: **d** has length 1

- The geometric method:

$$\left| t \cdot \mathbf{d} - \mathbf{m} \right| = r$$

$$(t \cdot \mathbf{d} - \mathbf{m})^2 = r^2$$

$$t^2 - 2t \cdot \mathbf{md} + \mathbf{m}^2 - r^2 = 0$$

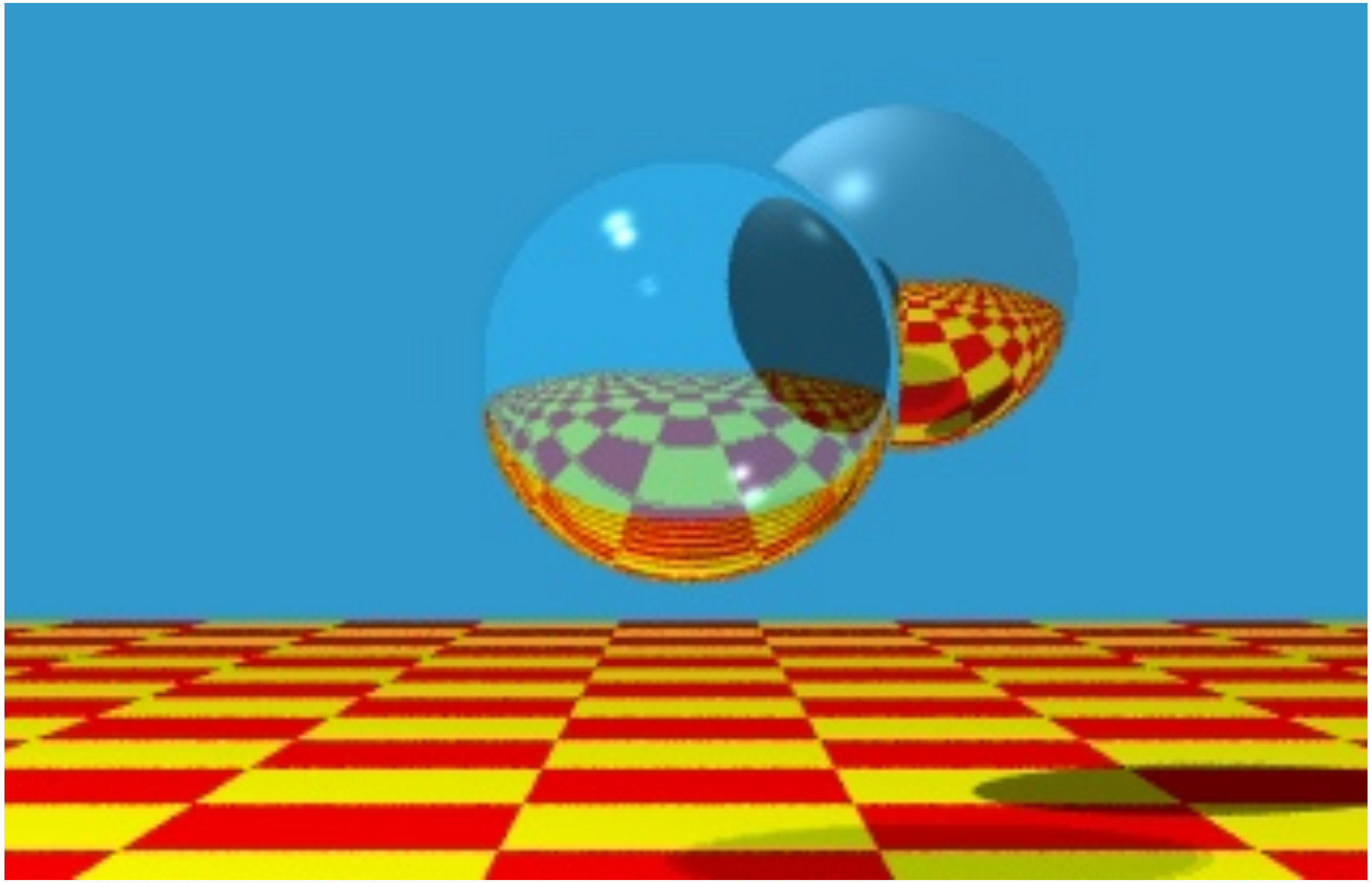


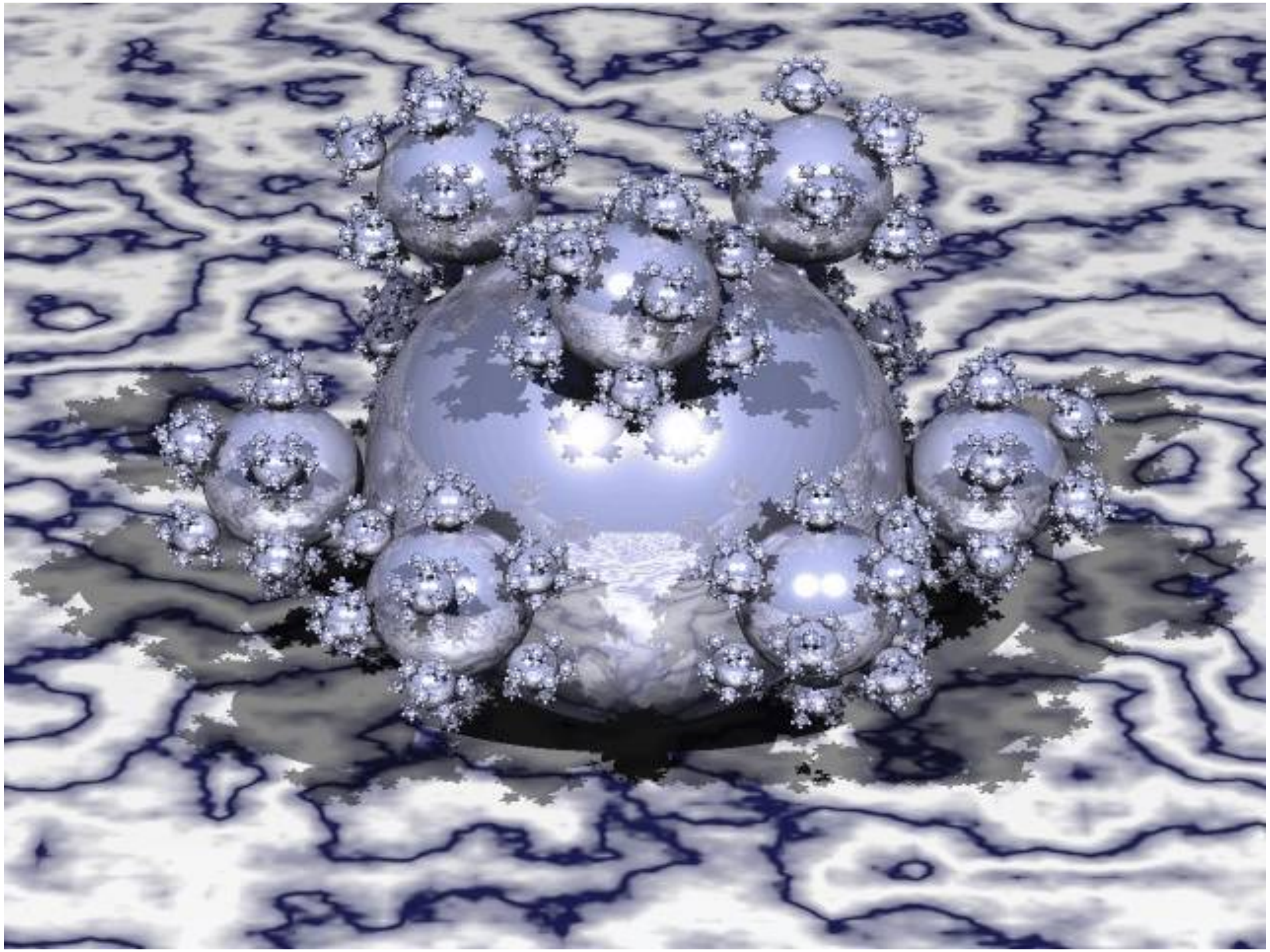
- The algebraic method:
  insert ray equation into implicit sphere equation
- There are many more approaches ...

- The algorithm, with small optimization:

```
calculate m²−r²
calculate b = m·d
if  m²−r² >= 0         // ray origin is outside sphere
    and b <= 0:        // and direction away from sphere
then
     return "no intersection"
let d = b² − m² + r²
if d < 0:
    return "no intersection"
if  m²−r² > ε :
    return t1 = b − √d  // enter; t1 is > 0
else:
    return t2 = b + √d  // leave; t2 is > 0  (t1<0)
```

- Ray-sphere intersection is so easy that all ray-tracers have spheres as geometric primitives! ☺





The "sphere flake"

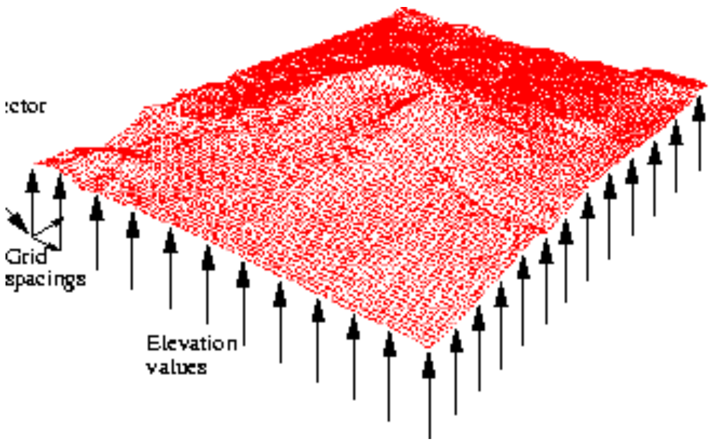
Ray-Tracing Height Fields

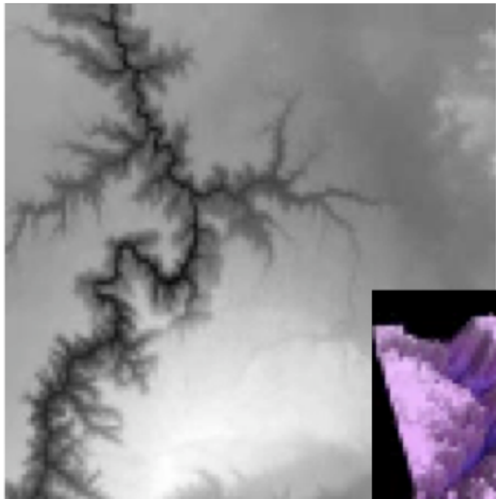
[Henning & Stephenson, 2004]

- Height Field = all kinds of surfaces that can be described by such a function

$$z = f(x, y)$$

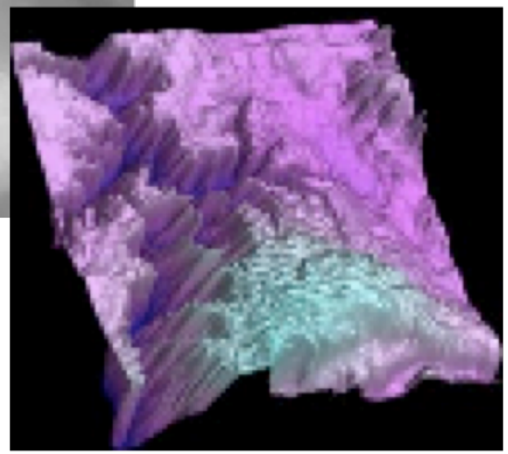
  - Examples: terrain, measurements sampled on a plane, 2D scalar field





Height field (= Bitmap)

Rendered

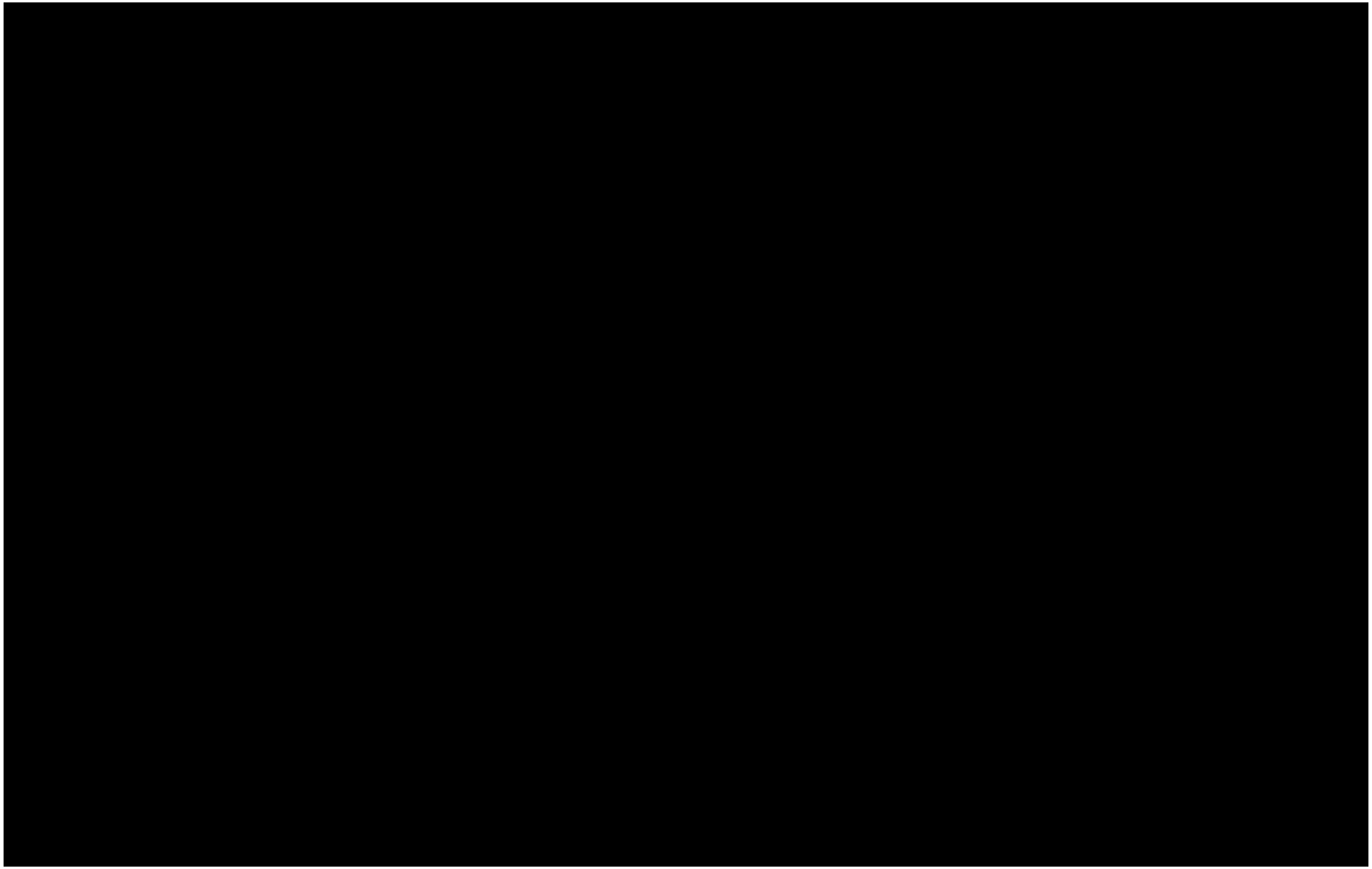


Example for Terrains

Turtmann Valley Dataset

- **3 datasets** of **4k x 4k** height-samples each @ 2m planar, 0.25m vertical inter-pixel spacing
- Normal-maps derived from input height-map (**3x4096x4096**), mixed **JPEG** and **S3TC** compression
- Compressed dataset size: **33 MB**
- Flight speed is around 540 km/h ~ **Mach 0,5**

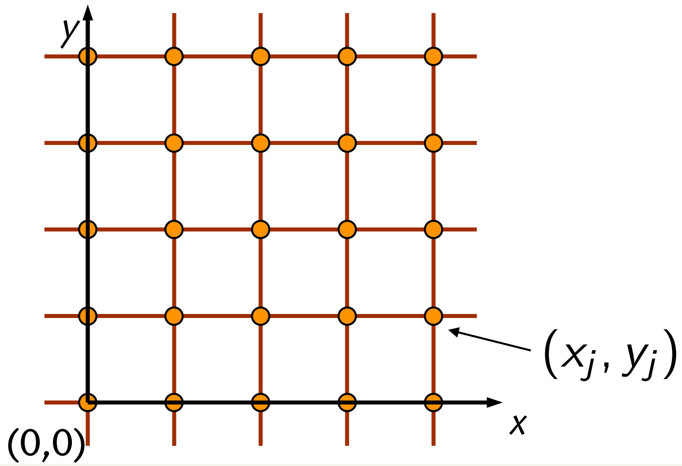
Bonn University

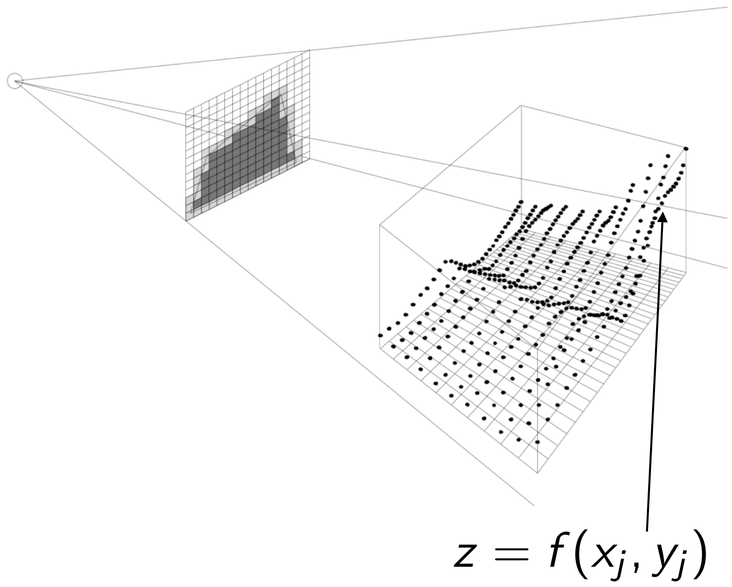


Valles Marineris, Mars - http://mars.jpl.nasa.gov

The Situation

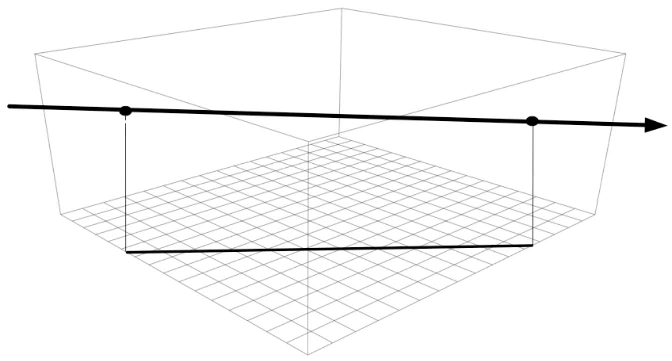
- The naïve method to ray-trace a height field:
  - Convert to $2n^2$ triangles, test ray against each triangle
  - Problems: slow, needs lots of memory
- Goal: direct ray-tracing of a height field represented as 2D array
- Given:
  - Ray
  - Array [0...n]x[0...n] with heights:



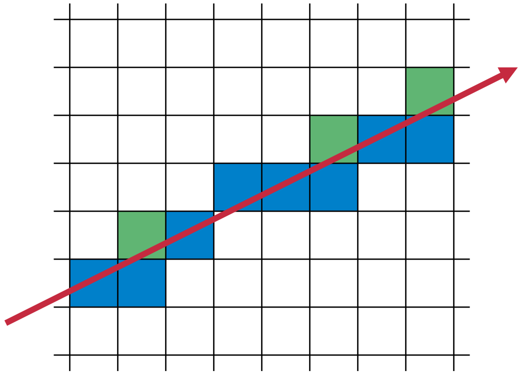


The Method

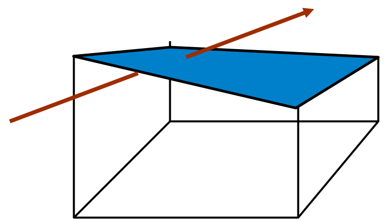
1. Reduce the dimension:
   - Project ray into xy plane



2. Visit all cells that are hit by the ray, starting with the nearest one
   - Notice similarity to scan conversion!
   - Use one of the DDA algorithms from CG1 😀

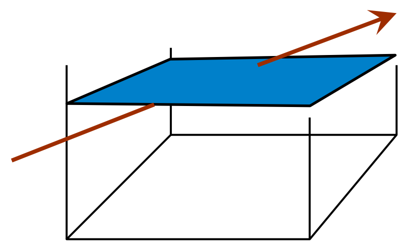


3. Test ray against the surface patch spanned by the 4 corners of the cell

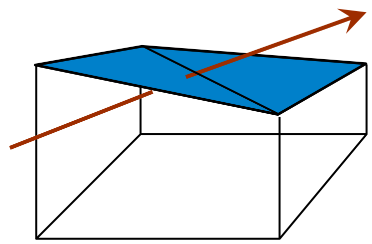


Intersection of Ray with Surface Patch of Cell

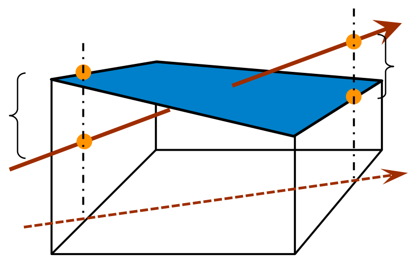
- Naïve methods:
  - "Nearest neighbor":
    - Compute average height of the 4 corner height values
    - Intersect ray with horizontal square of that average height
    - Problem: very imprecise



  - Tessellate by 2 triangles:
    - Construct 2 triangles from the 4 corner points
    - Problem: tessellation is not unique, diagonal edge could produce severe artifact

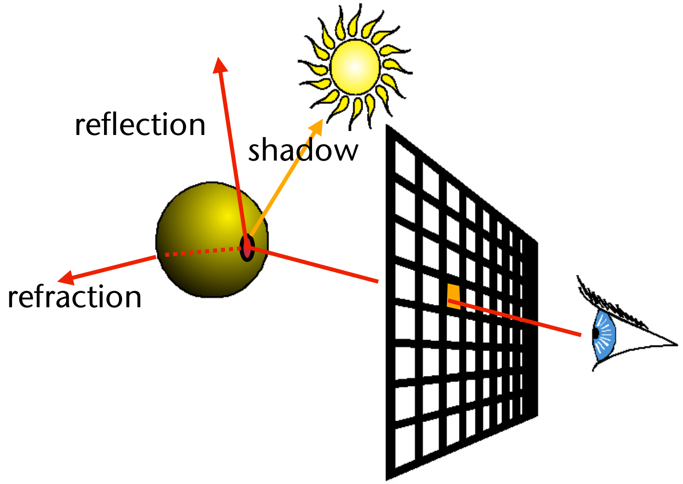


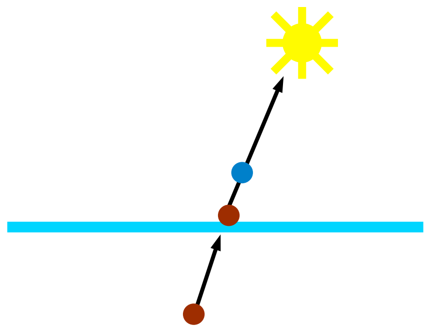
- Better: *bilinear interpolation*
  - Determine height of the surface on the edge of the cell above the point where the ray enters/leaves the cell ⟶ linear interpolation of corner heights
  - Compare signs
  - Insert ray equation into bilinear equation of surface ⟶ quadratic equation for line parameter t
  - (The surface is called a parabolic hyperboloid)



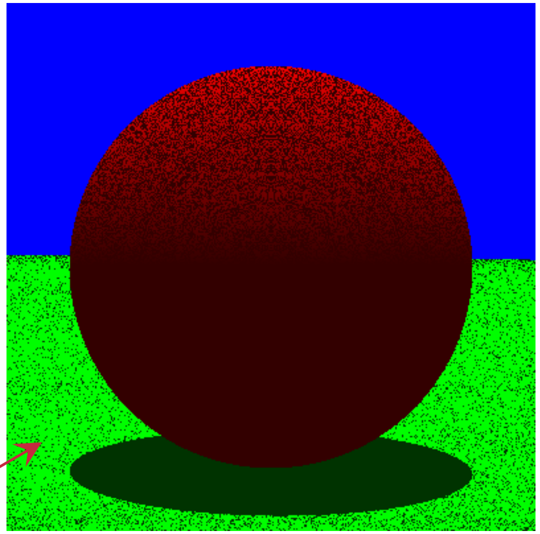
The evil ε

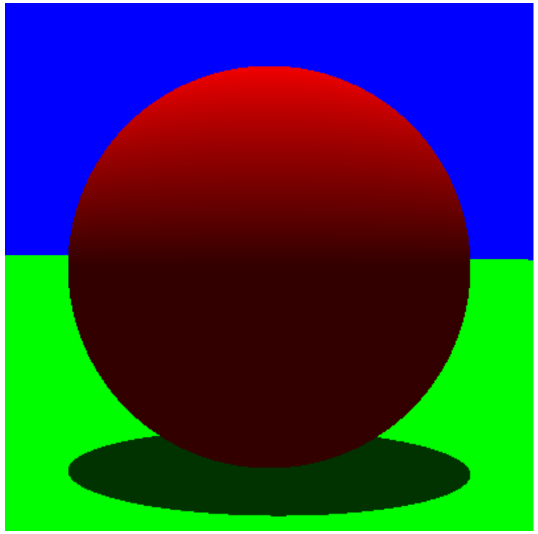
- What happens when the origin of a ray is "exactly" on the surface of an object?
- Remember: floating-point calculations are always imprecise!
  - Consequence: in subsequent ray-scene intersection tests, the ray might appear to be inside the original object!
  - Further consequence: we get wrong intersection points!
- "Solution": move the origin of the ray by a small ε along the direction of the (new) ray





More Glitch Pictures

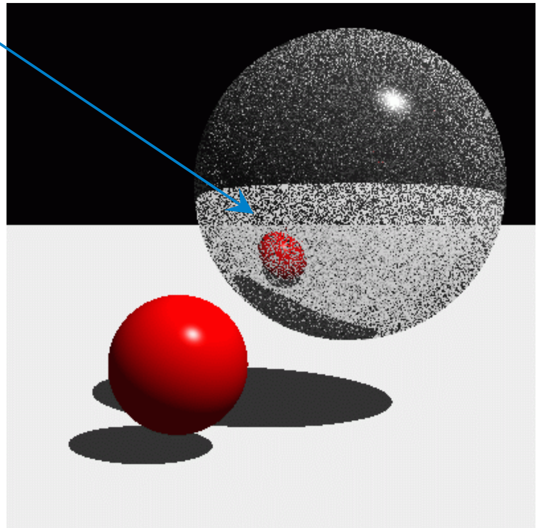


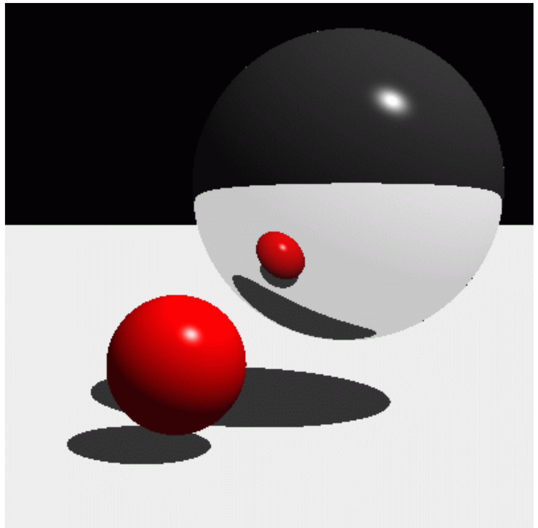


Speckles

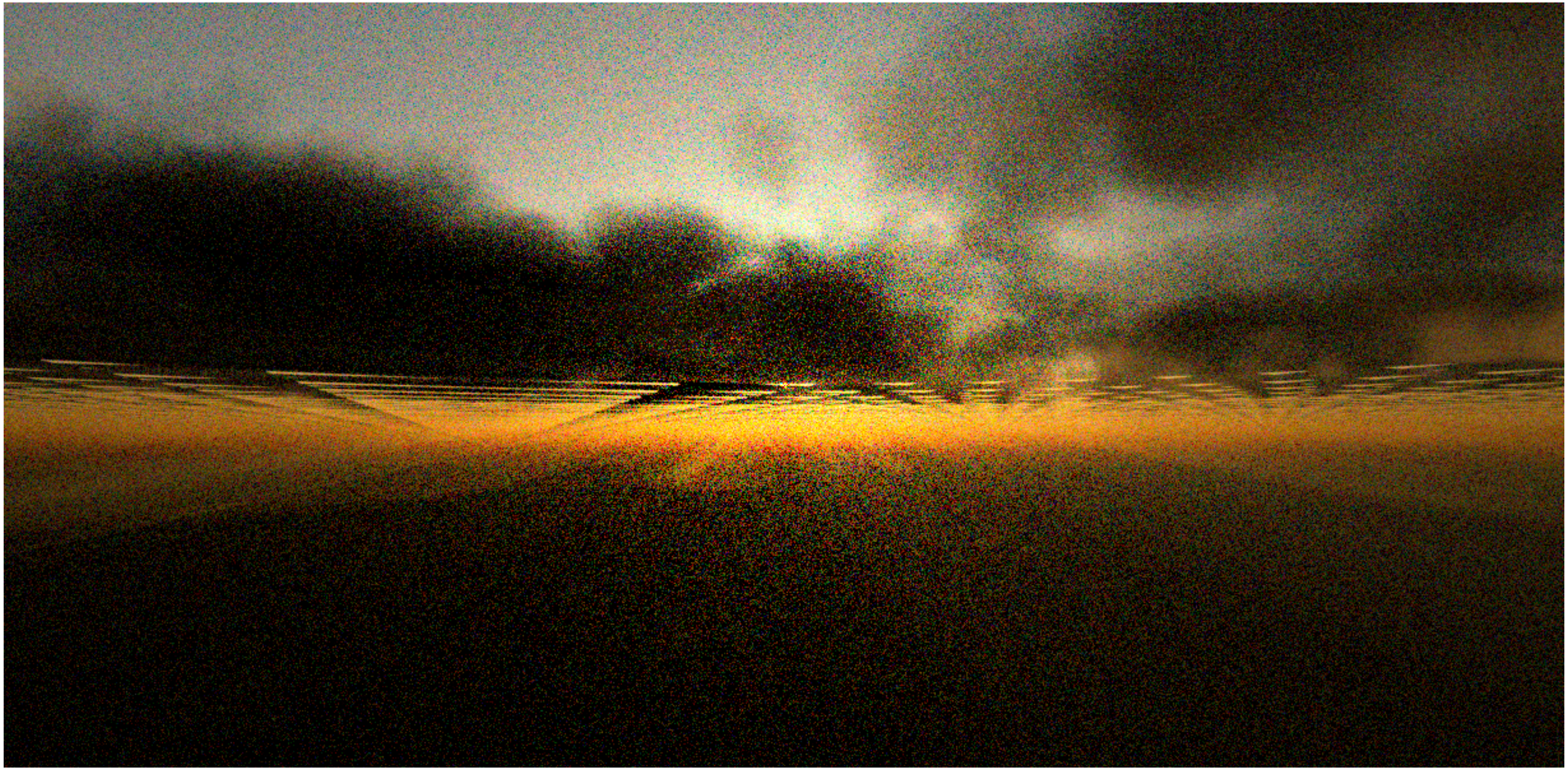
Without epsilon

With epsilon





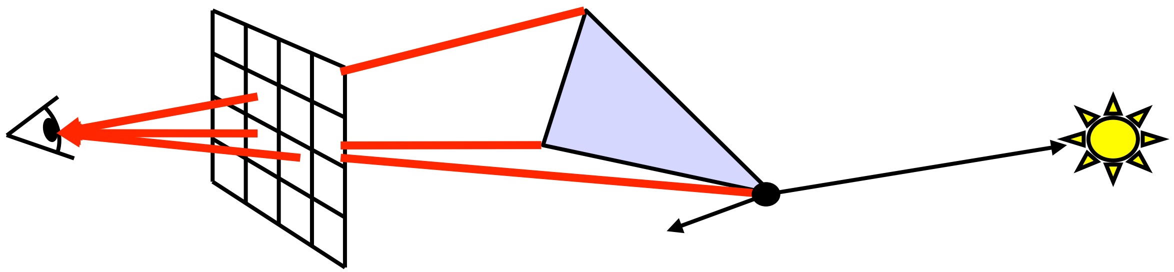
Numerically unstable cloud layer intersection



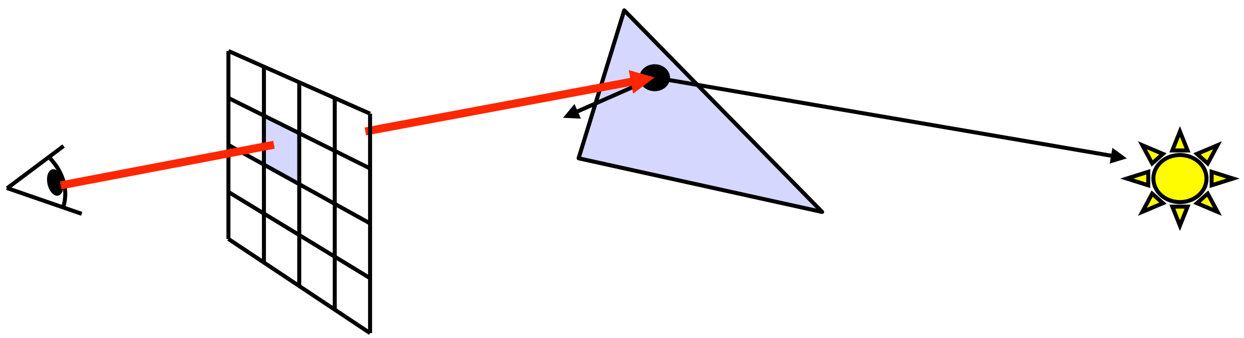
Source: necro (http://igad2.nhtv.nl/ompf2)

Comparison of Scan Conversion and Ray-Tracing

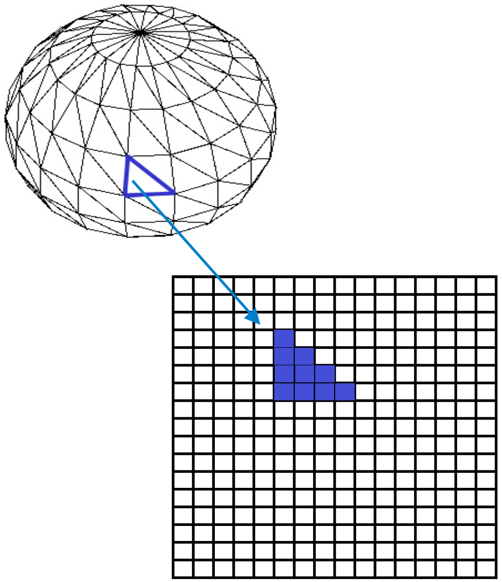
- Scan conversion: start with triangles, project each vertex = send ray through each vertex



- Raytracing: start with pixels, send ray through each pixel

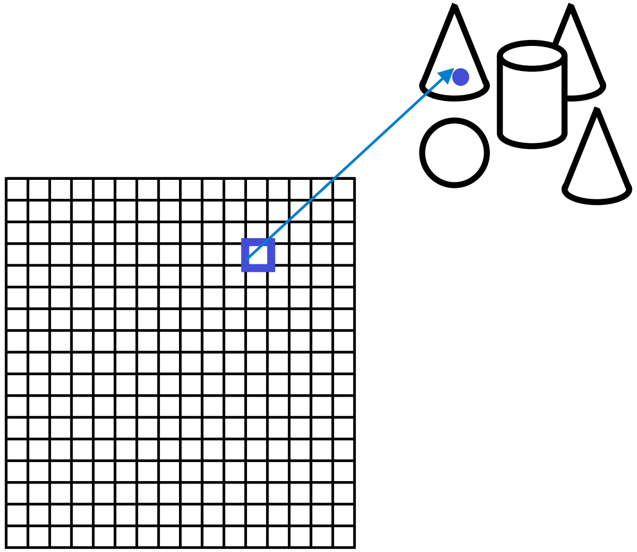


- For rendering a complete scene using scan conversion ...



... scan-convert each triangle

- For rendering a complete scene using raytracing ...



... trace a ray through each pixel

Advantages & Disadvantages

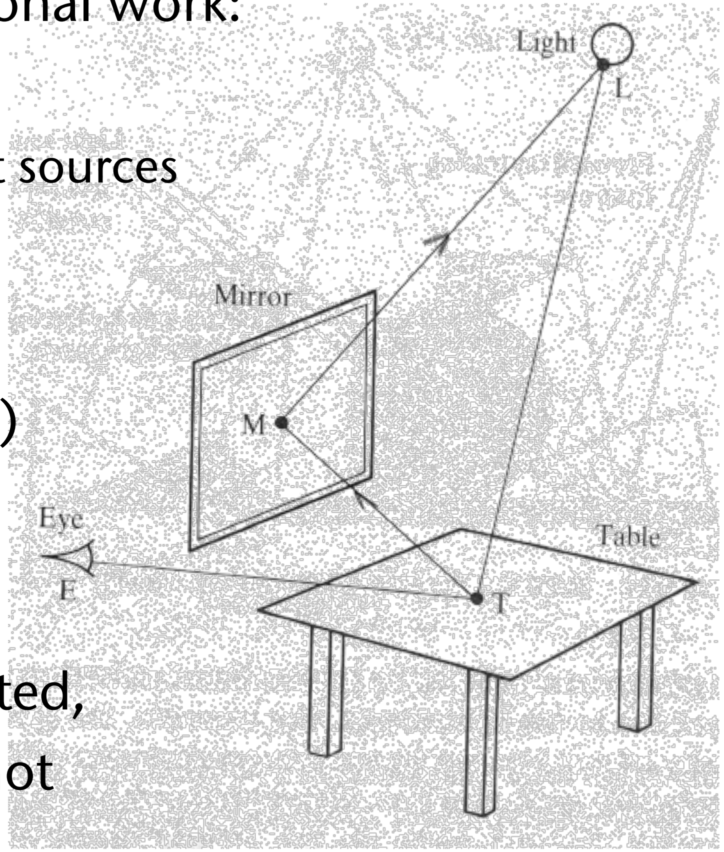
- Scan conversion:
  - Fast (for a number of reasons)
  - Supported by all graphics hardware
  - Well-suited for real-time graphics
  - Only heuristic solutions for shadows and transparent objects
  - No interreflections
- Raytracing:
  - Offers general and simple (in principle) solution for global effects, such as shadows, interreflection, transparent objects, etc.
  - Much slower (unless you cast only primary rays)
  - Not supported by most graphics hardware
  - Difficult to achieve real-time rendering

Other Advantages of Ray-Tracing

- Shines with scenes that contain lots of glossy/shiny surfaces and transparent objects
- Fairly easy to incorporate other object representations (e.g., CSG, smoke, fluids, …)
  - Only prerequisite: must be possible to compute the itersection between ray and object, and to compute the normal at the point of intersection
- No separate clipping step necessary

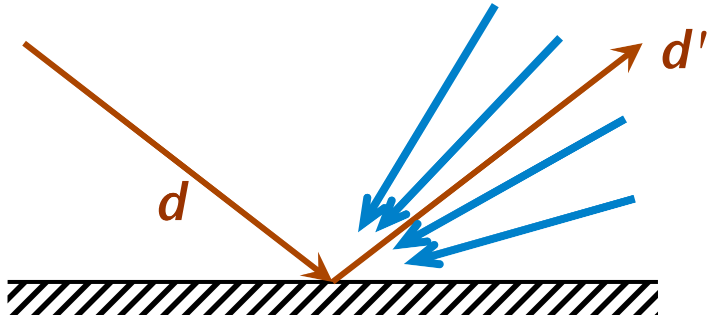
Disadvantages of (Simple) Ray-Tracing

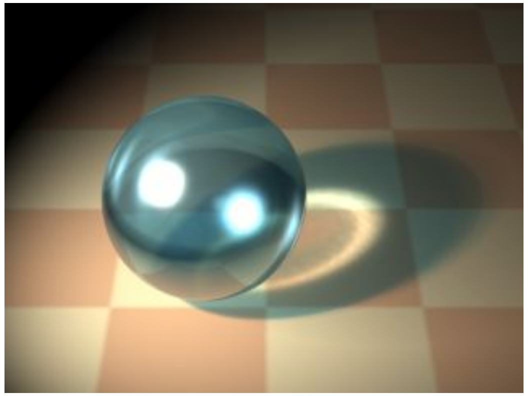
- Needs a huge amount of computational work:
  - Just for primary rays: $O(\, p \cdot (n+l) \,)$,
    p = # pixels, n = # polygons, l = # light sources
  - Number of rays grows exponentially with number of recursions!
- No indirect lighting (e.g., by mirrors)
- No soft shadows
- With each camera movement, the complete ray tree must be recomputed, although an object's shading does not depend on the camera's position
- For all of these disadvantages, a number of remedies have been proposed …

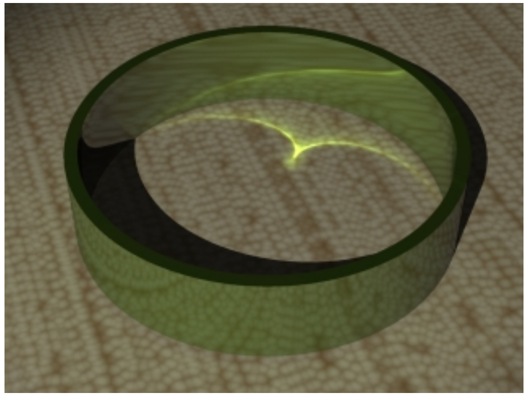


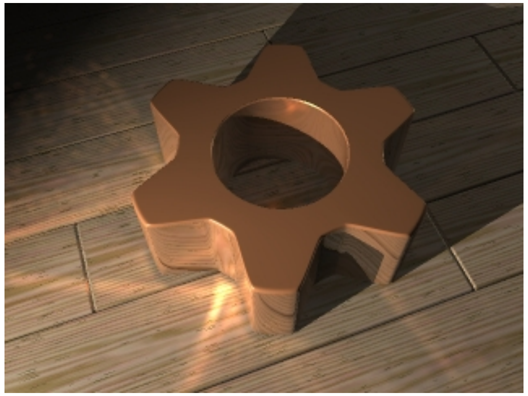
Example for the Problem with (Missing) Indirect Lighting: Caustics

- Caustic = reflected / transmitted light is concentrated in a point or, possibly curved, line on the surface of another object
- Problem:
  - Ray-tracing follows light paths "in reverse"
  - Simple ray-tracing follows only one path







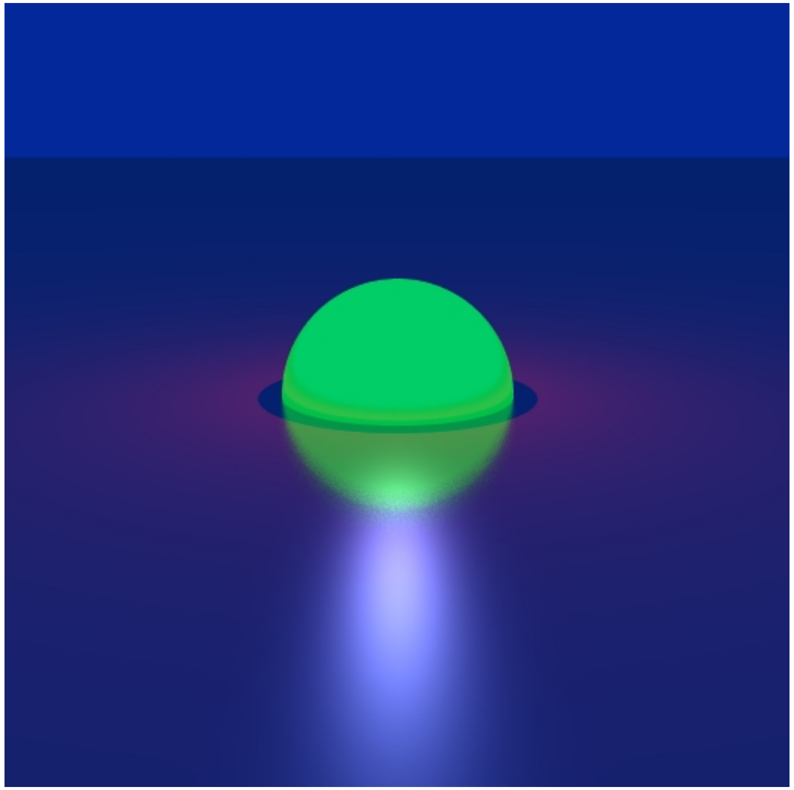


Aliasing

- One ray per pixel → causes typical aliasing artefacts:
  - "Jaggies"
  - Moiré effects

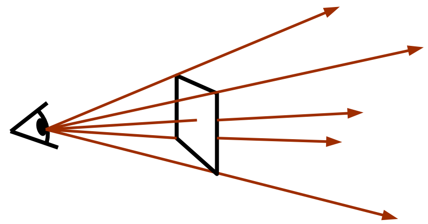
Distribution Ray Tracing

- Simple modification of ray-tracing to achieve
  - Anti-alising
  - Soft shadows
  - Depth-of-field
  - Shiny/glossy surfaces
  - Motion blur
- Was proposed under a different name:
  - "Distributed Ray Tracing"
  - Don't use that name ("distributed" = verteilt)

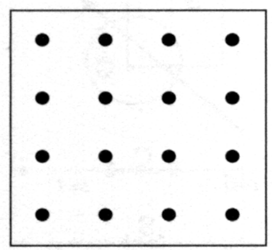


Anti-Aliasing with Ray-Tracing

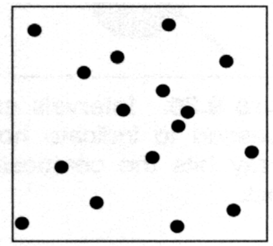
- Shoot many rays per pixel, instead of just one, and average retrieved colors



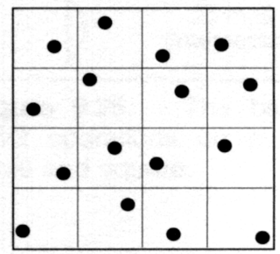
- Methods for constructing the rays:
  - Regular sampling (perhaps problems with Moiré patterns)



  - Random sampling (problems because of noise)

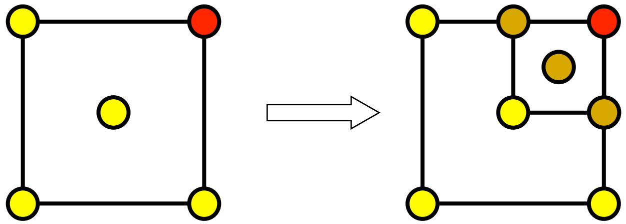


  - Stratification: combination of regular and random sampling, e.g., by placing a grid over the pixel, and picknig one random position per cell

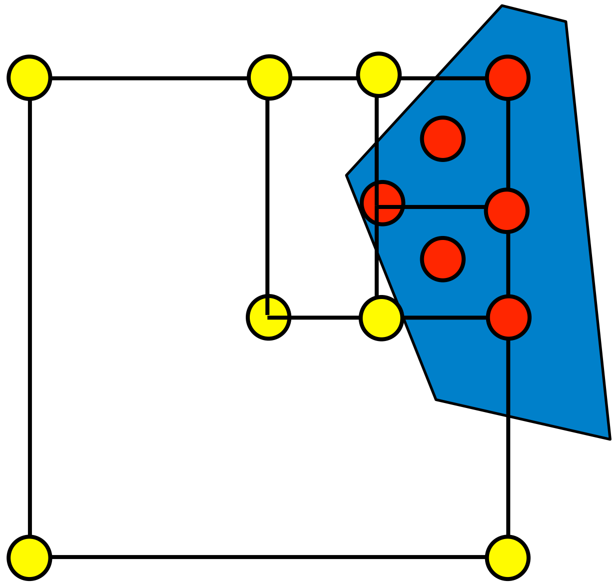


Adaptive Supersampling

- Idea: shoot more rays only in case of large differences in color



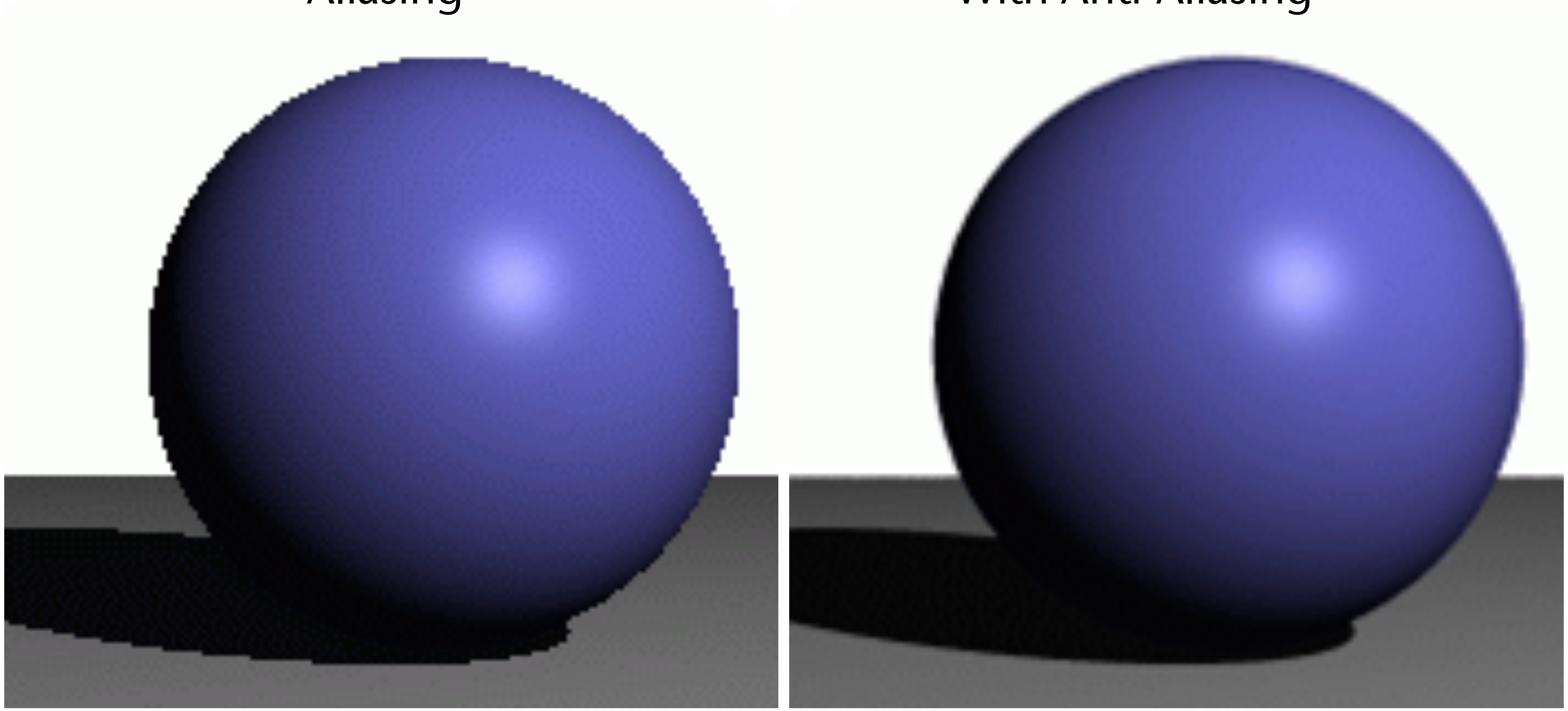
- Example:



- Resulting color = weighted average of all samples, weighted by the area each sample covers

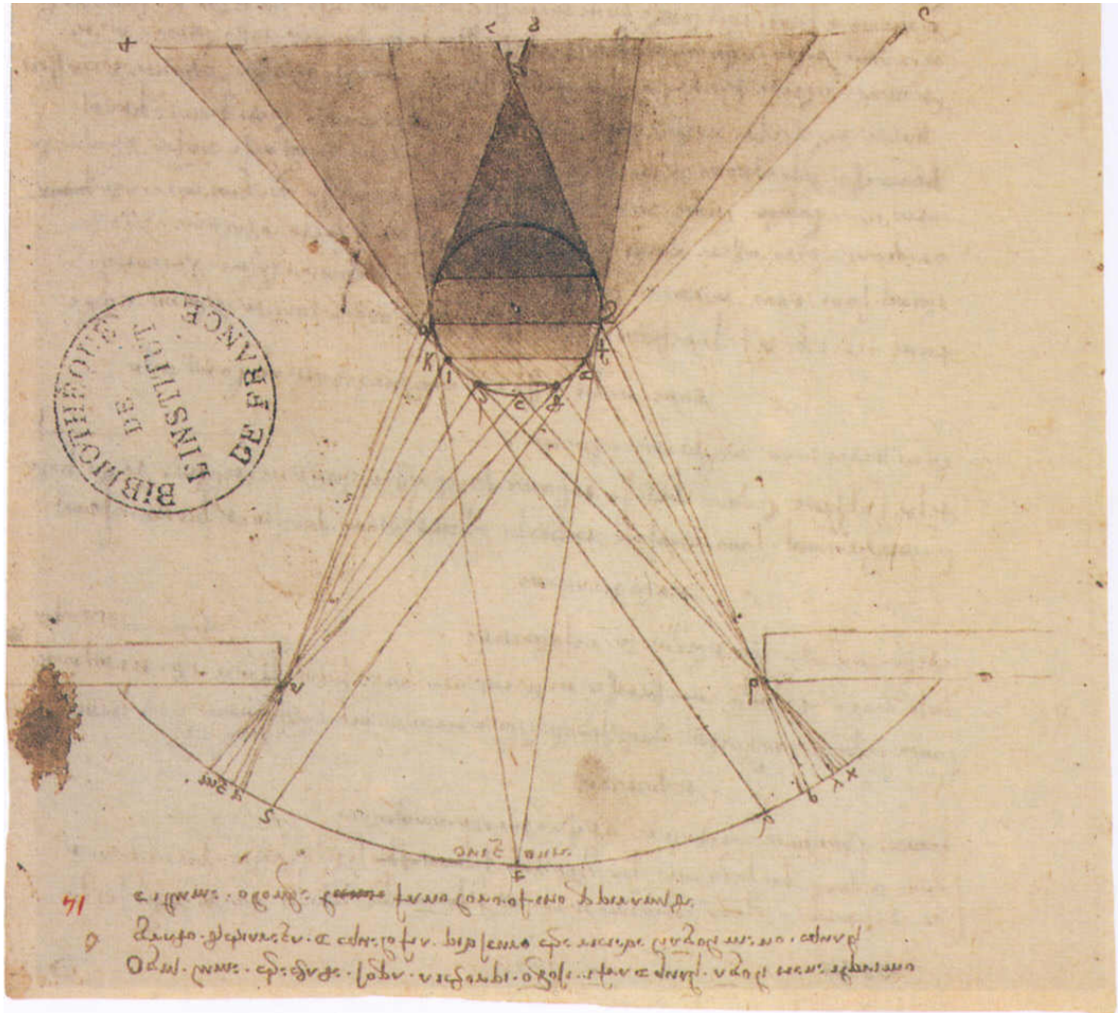
Result

Aliasing | With Anti-Aliasing



Soft Shadows, Penumbra

- Behind a lighted object, there are 3 regions:
  - Completely in shadow (*umbra*)
  - Partially in shadow (*penumbra*)
  - Completely lighted

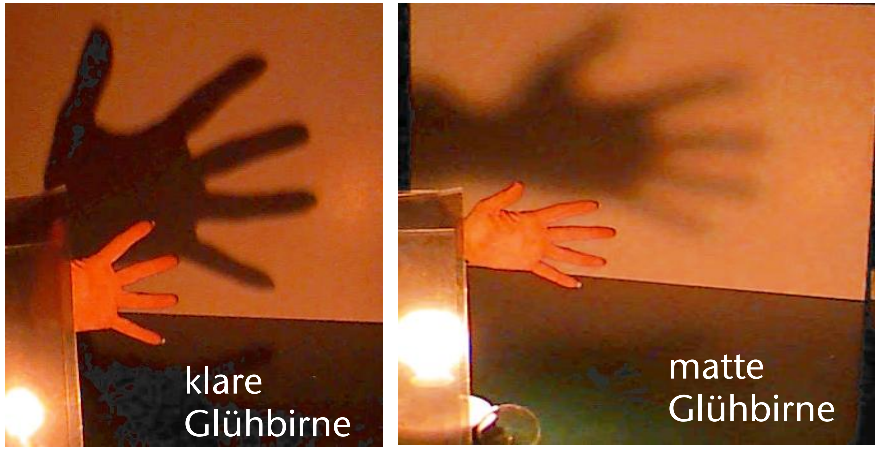


XVI. Léonard de Vinci (1452-1519). Lumière d'une fenêtre sur une sphère ombreuse avec (en partant du haut) ombre intermédiaire, primitive, dérivée et (sur la surface, en bas) portée. Plume et lavis sur pointe de métal sur papier. 24 x 38 cm. Paris, Bibliothèque de l'Institut de France (ms. 2185 ; B.N. 2038, f° 14 r°).

In Reality ...

http://3media.initialized.org/photos/2000-10-18/index_gall.htm

http://www.davidfay.com/index.php

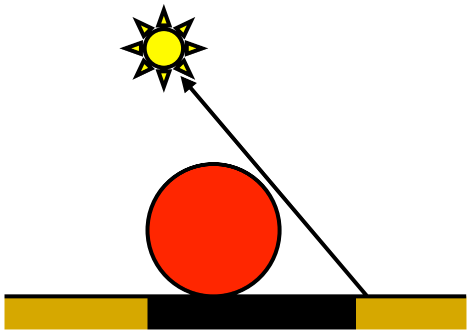


http://www.pa.uky.edu/~sciworks/light/preview/bulb2.htm

… and in Ray-Tracing

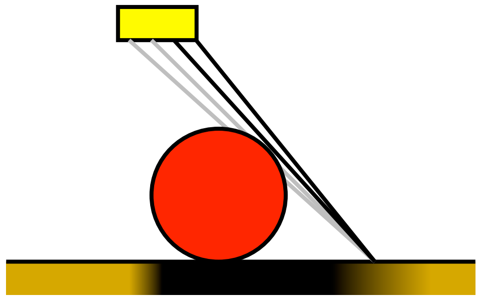
- So far, exactly 1 shadow feeler per light:
  - We add a light source's contribution or not, depending on

$$s_i = \begin{cases} 1\,, & \text{light source visible} \\ 0\,, & \text{invisible} \end{cases}$$

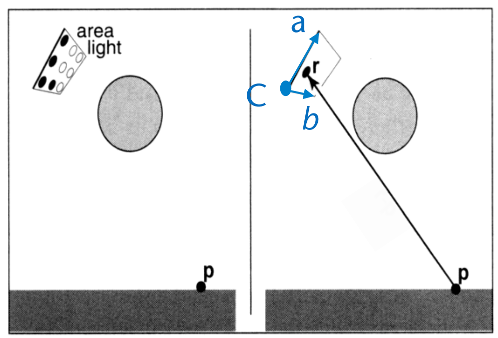


- Now, send many shadow feelers:

$$s_i = \frac{\#\ \text{passing shadow rays}}{\#\ \text{shadow rays sent}}$$

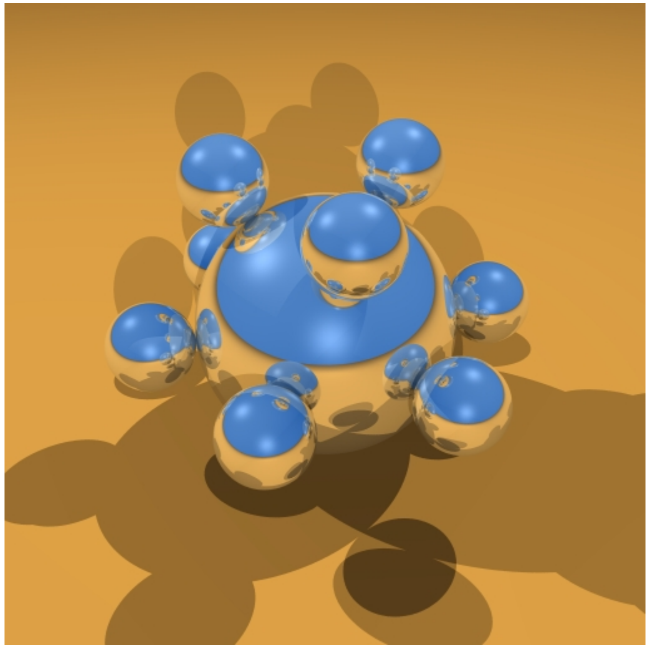


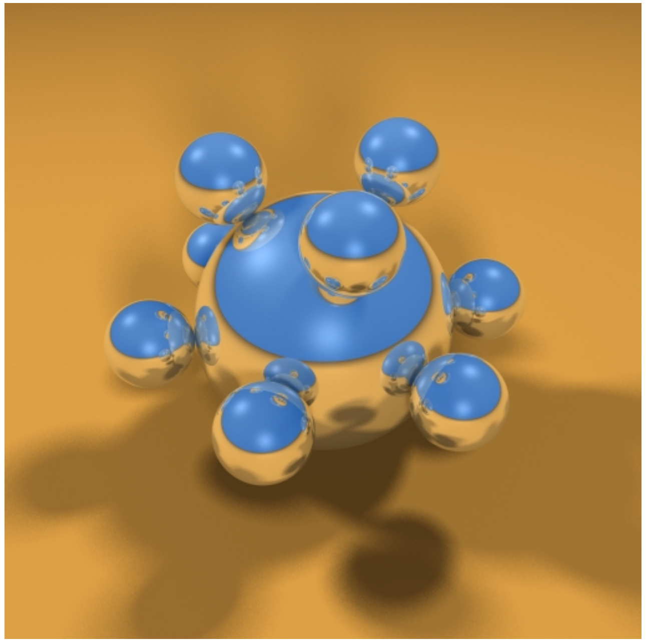
- Drei Arten von Sampling der Lichtquelle:
  - Regelmäßige Abtastung der Lichtquelle
  - Zufällige Abtastung der Lichtquellen
  - Stratifizierte Abtastug

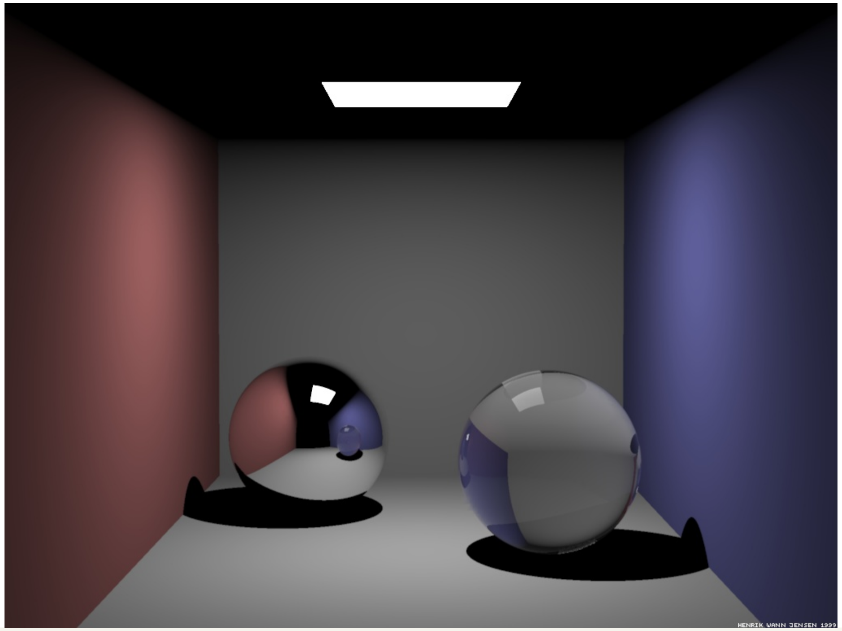


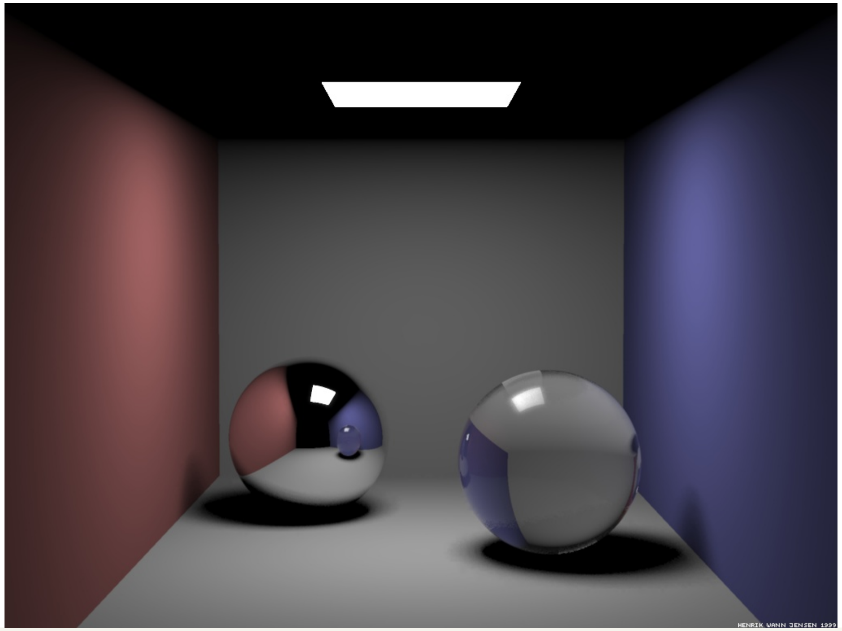
- Modification of the (local) lighting model:

$$L_{\text{Phong}} = \sum_{j=1}^{n} s_j \cdot f(\Phi_j, \Theta_j) \cdot I_j$$









- Geometric construction of the different shadow regions:

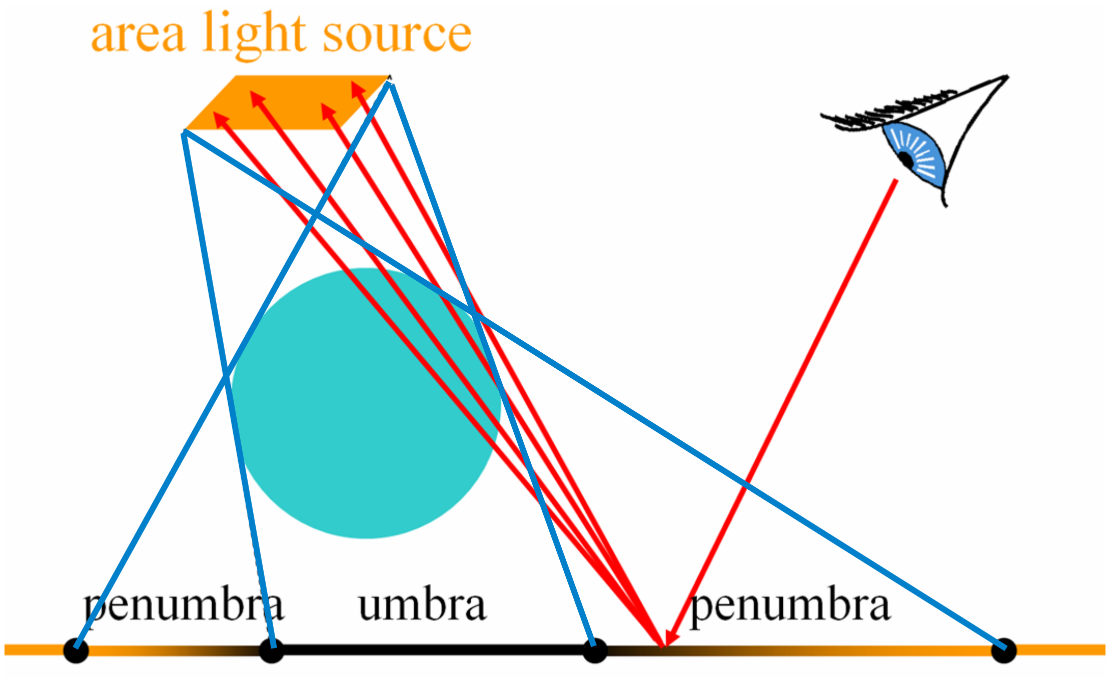
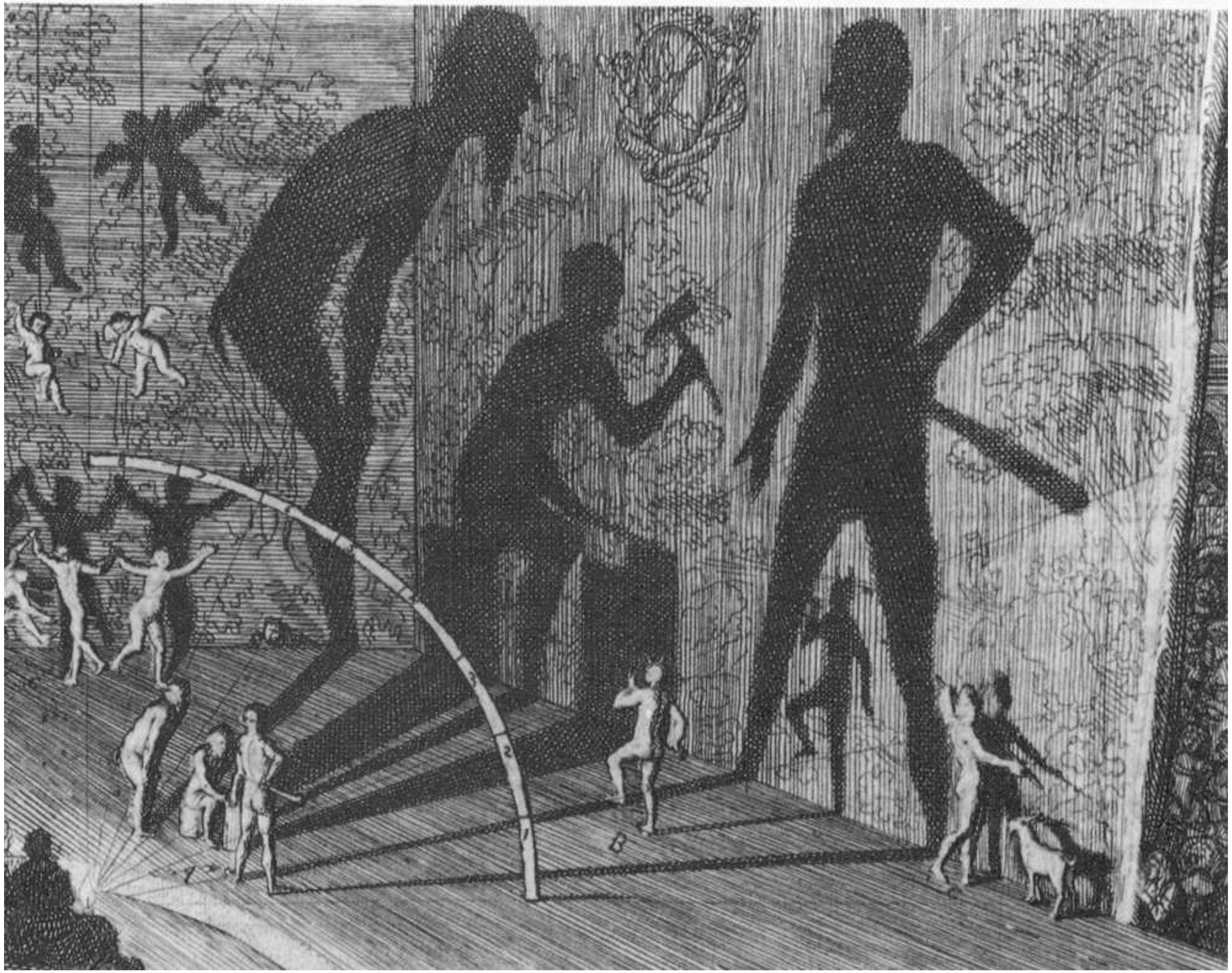
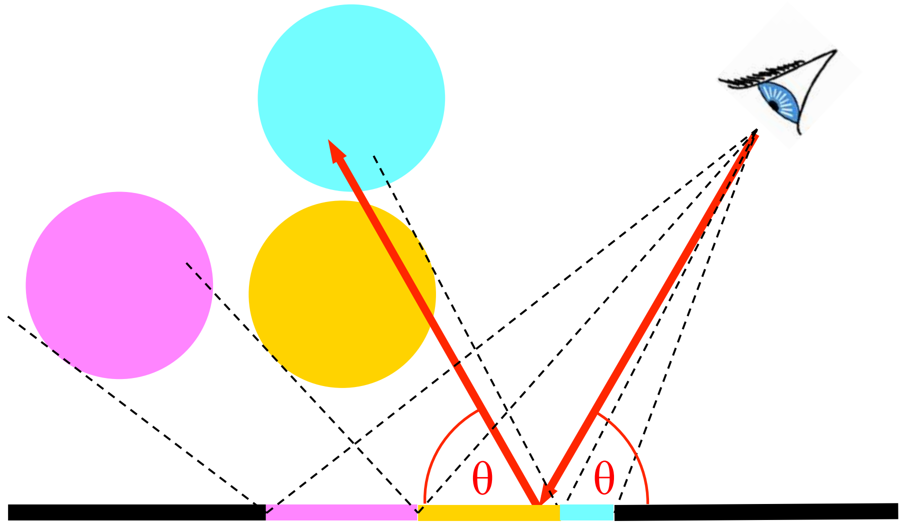


Plate 50 Samuel van Hoogstraten, *Shadow Theatre.* From *Inleyding tot de hooghe schoole der schilderkonst* 1678. (Einführung in die hohe Schule der Gemäldemalerei)



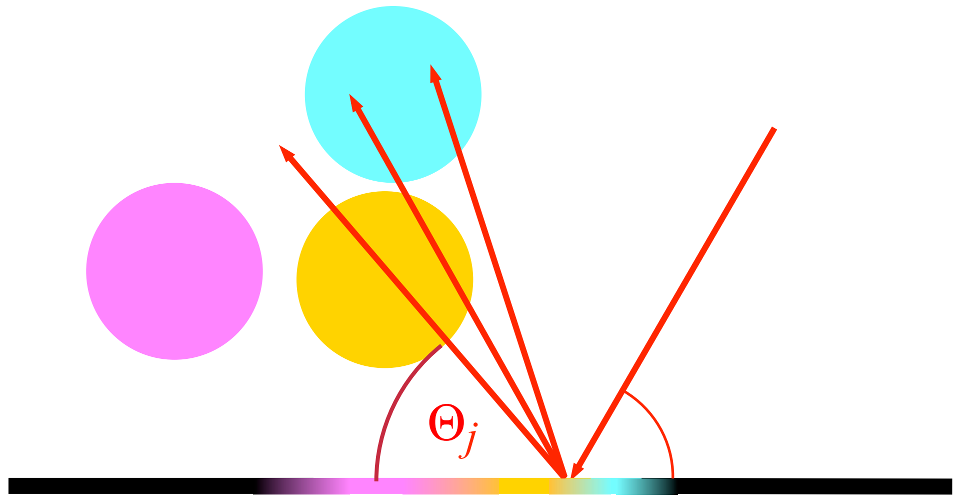
Better Glossy/Specular Reflection

- So far, exactly 1 reflected ray:
  - Problem, if the surface should be matte-glossy …

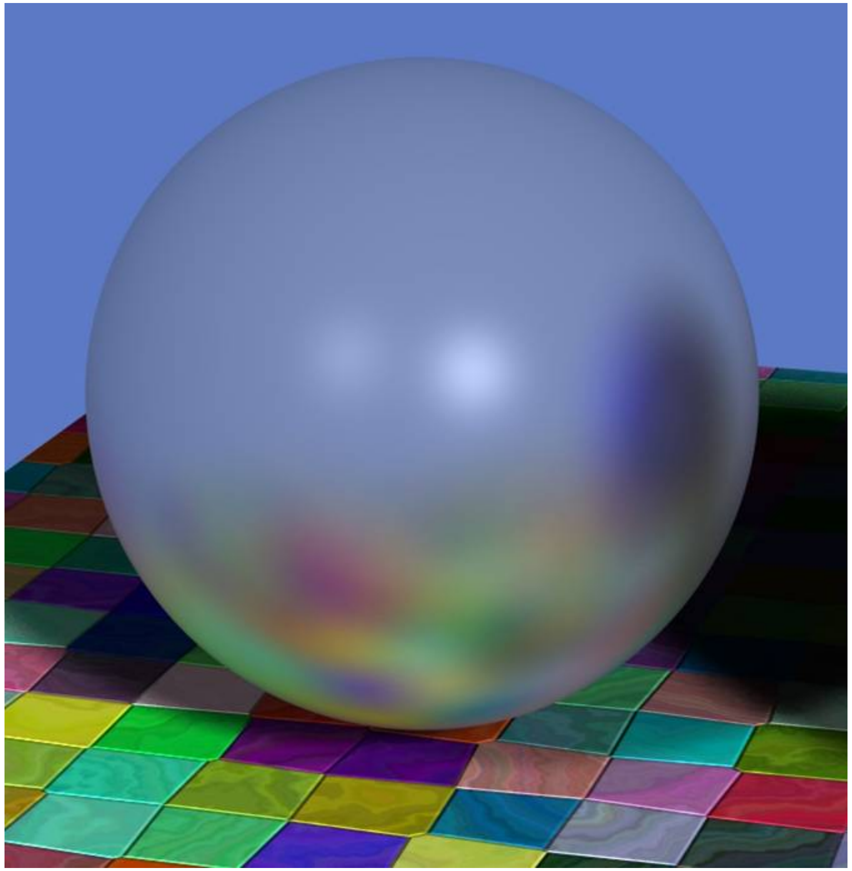


- Solution (somewhat brute-force):
  - Shoot many secondary, "reflected" rays
  - Accumulate according to power-cosine law (Phong)

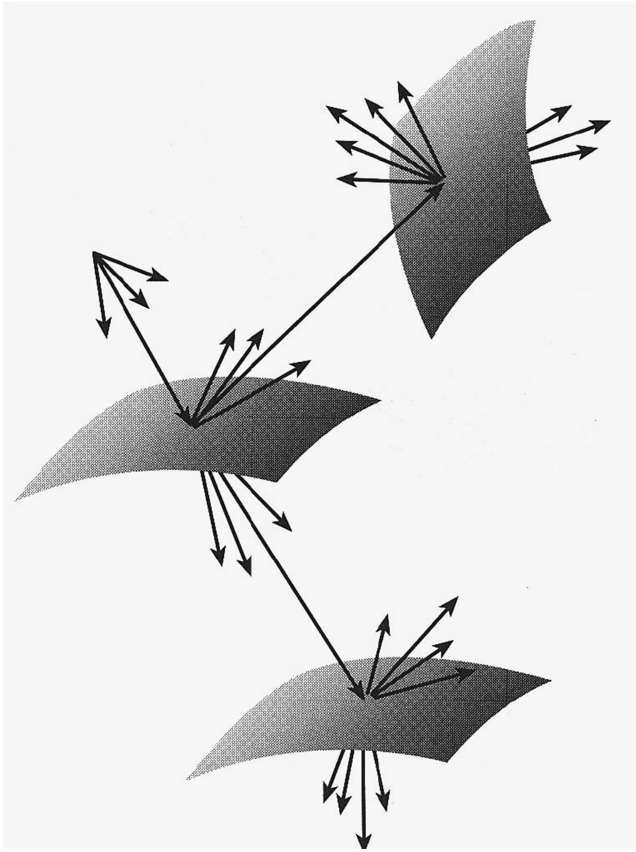
$$\cos^p \Theta_j$$



- Example:

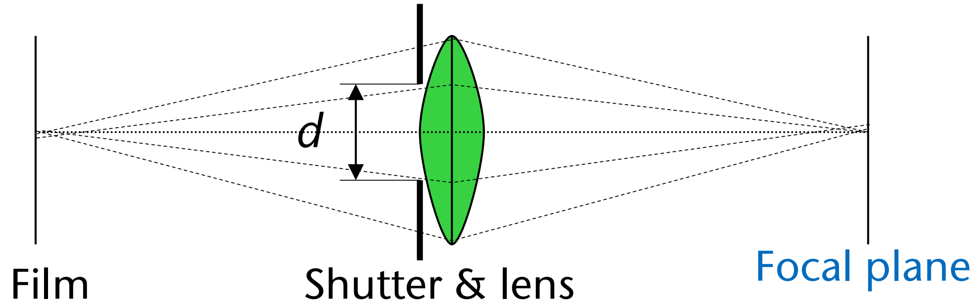


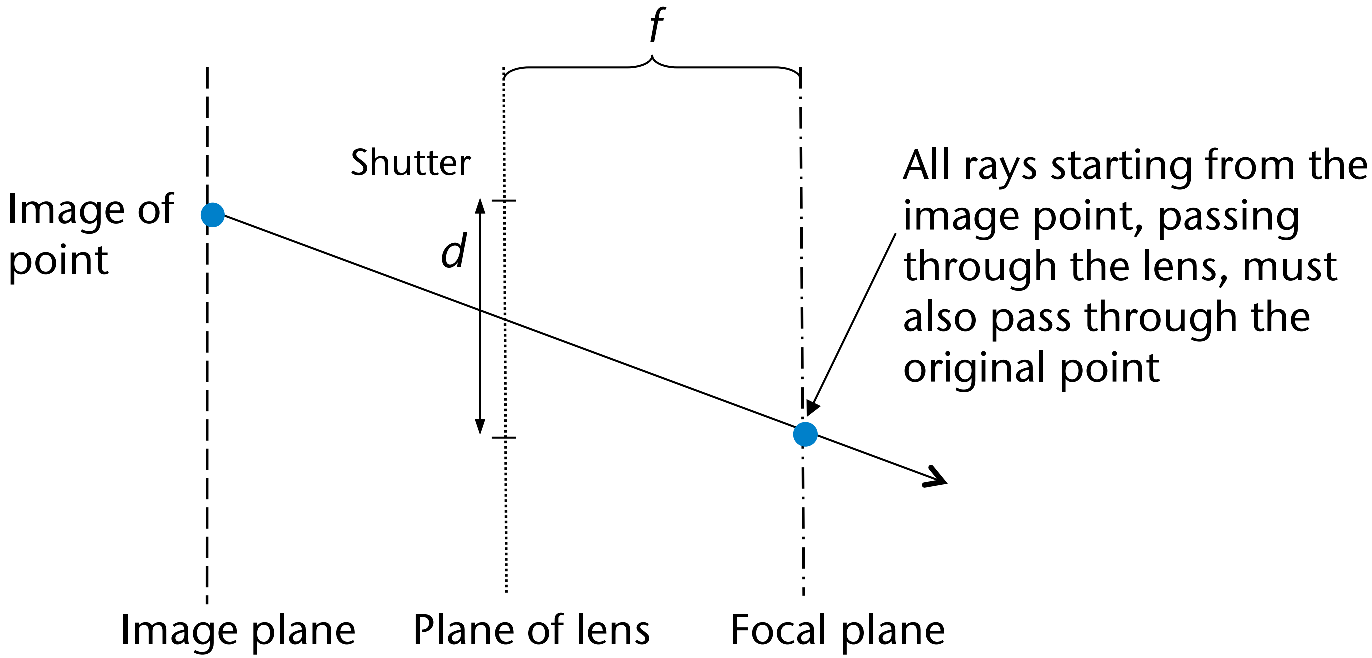
- The ray tree:



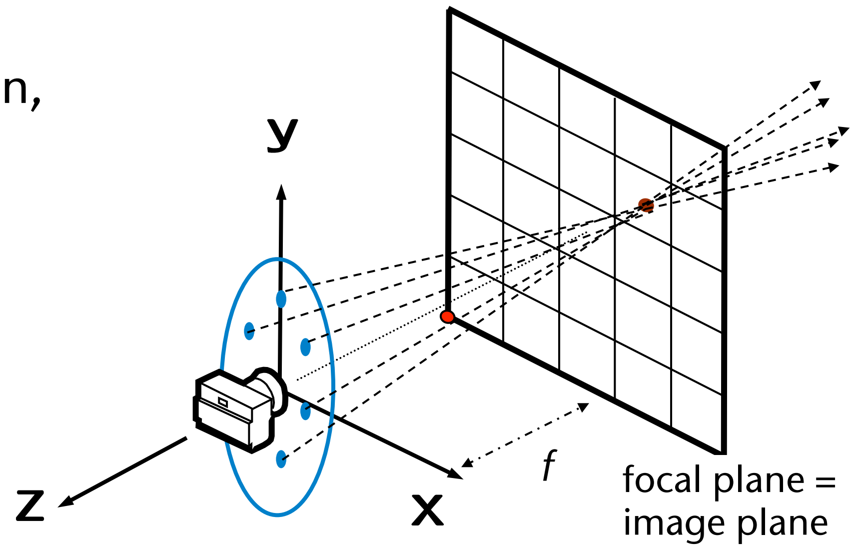
Depth-of-Field (Tiefen(un-)schärfe)

- So far: ideal pin-hole camera model
- For depth-of-field, we need to model real cameras



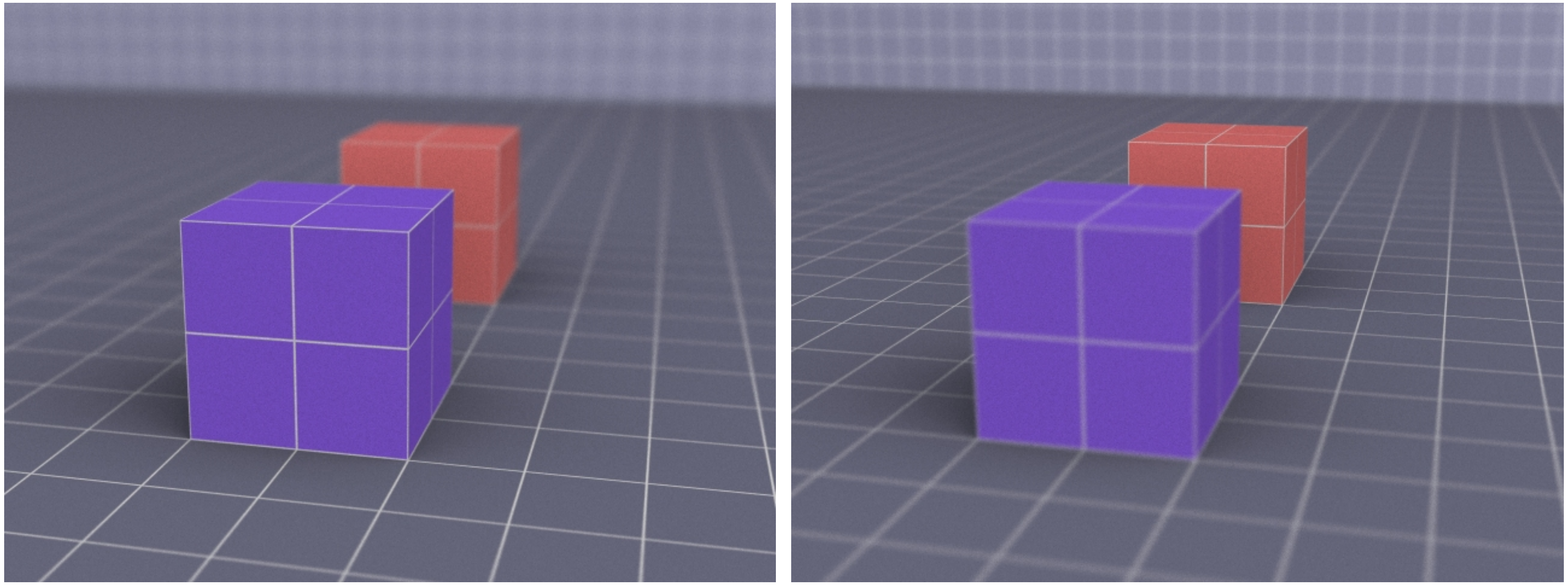


- A class **`LensCamera`** would generate rays like this:
  - Sample the whole shutter opening by some distribution, shoot ray from each sample point through focal plane = image plane



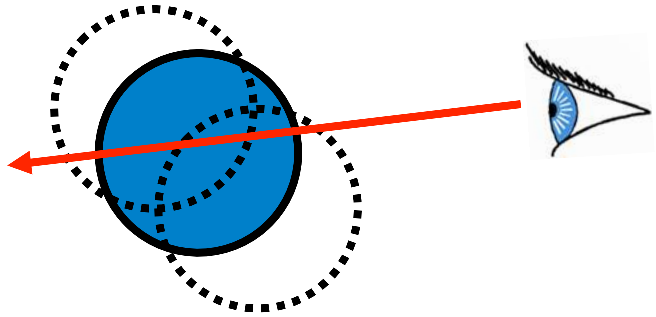
- Remark:
  - Again, use stratified sampling for sampling the disc (= shutter)

Examples



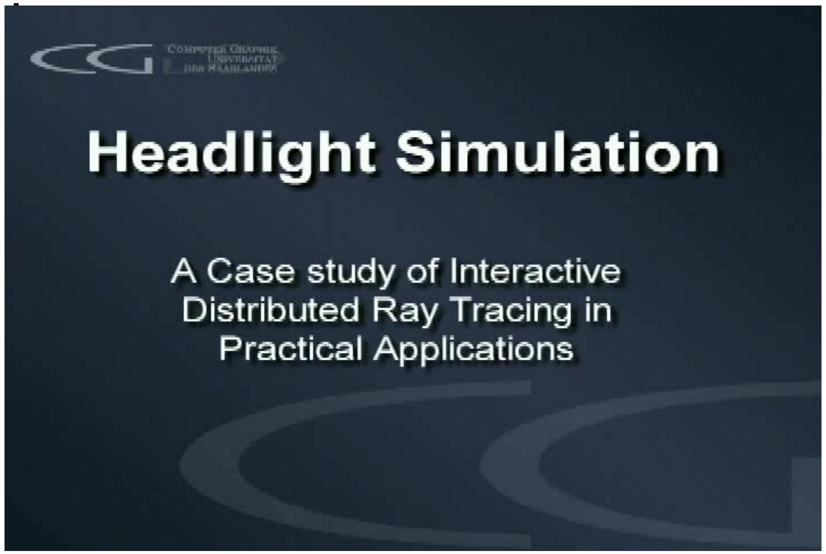
Motion Blur (Bewegungsunschärfe)

- Goal: compute "average" image for time interval $[t_0, t_1]$, during which objects move
- Sample time interval with $t \in [t_0, t_1]$ and shoot one ray per pixel per time t
- When computing ray-object intersections, use positions $P = P(t)$ for all objects
- Average color of all rays for one pixel

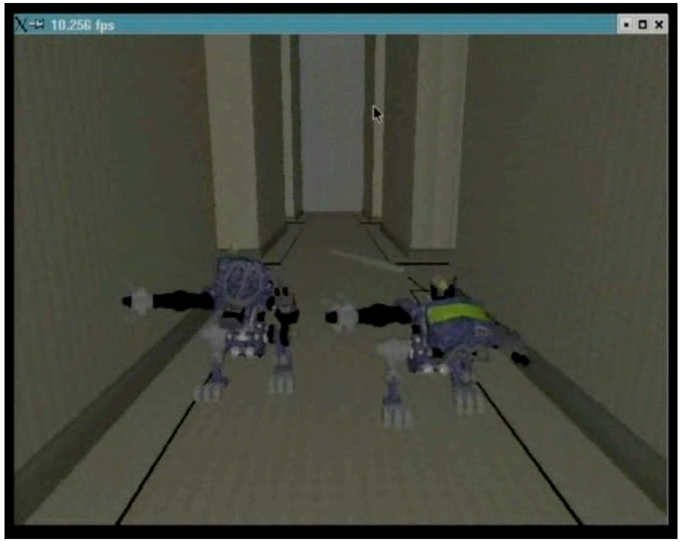


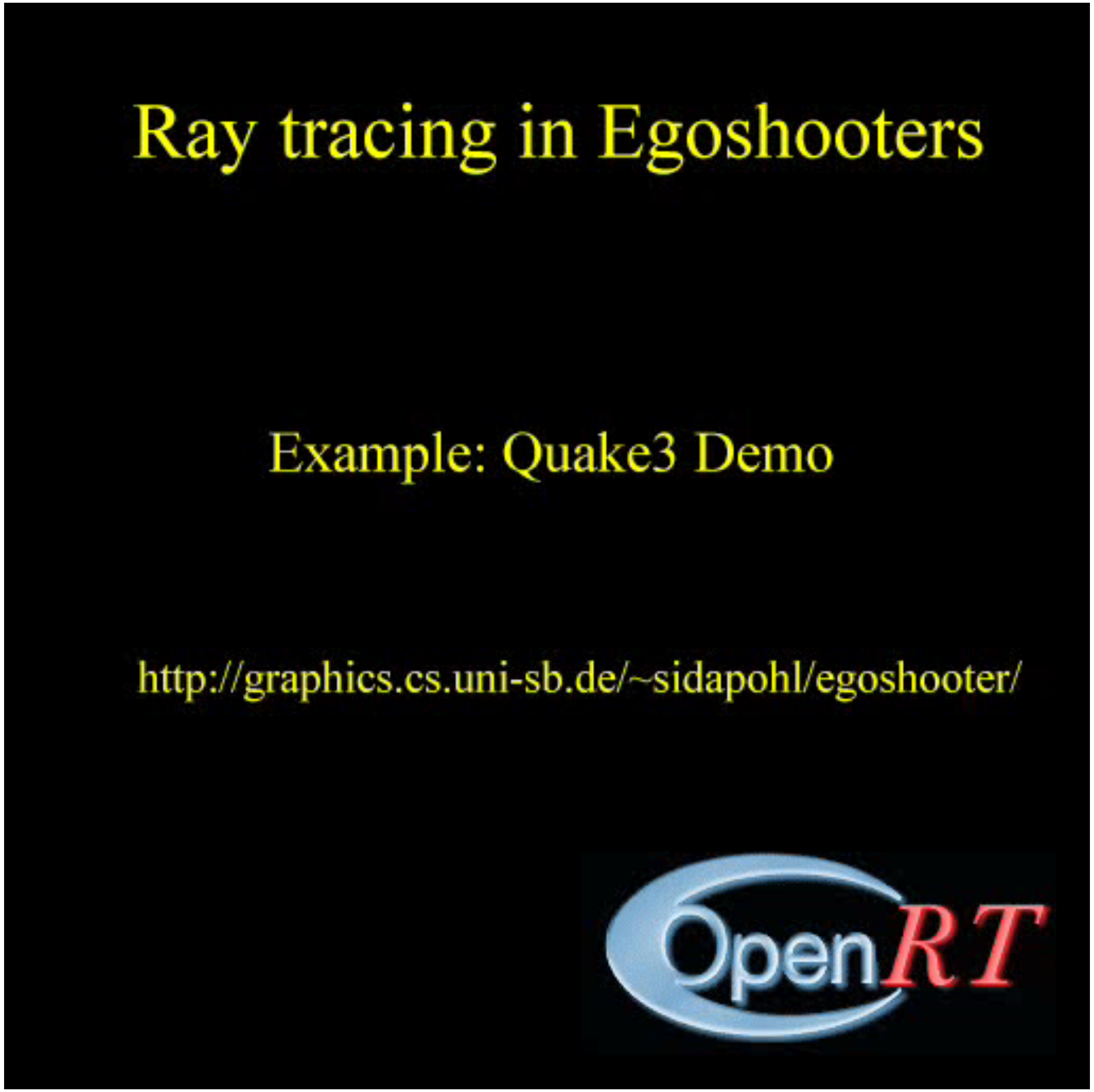
"But is it real-time?"

- Ray Tracing in der Vergangenheit war sehr langsam
- Inzwischen Echtzeit-Fähigkeit für einige Szenen
- OpenRT-Projekt: Real-Time Ray Tracing
  - Siehe http://www.openrt.de
- Special-Purpose-Hardware, PC-Cluster
- Nur eine Frage der Zeit, bis Commodity-Graphics-Hardware es



Uni Saarbrücken





Quake 3 mit Ray-Tracing. Plattform: Cluster mit 20 AMD XP1800. 2004
http://graphics.cs.uni-sb.de/~sidapohl/egoshooter/

Eine Anmerkung zu Typos

- Typos passieren auch auf den Folien
  - Keine Angst haben zu fragen!
  - Bitte teilen Sie mir Fehler mit
- Typos passieren sogar in Lehrbüchern
  - Ich selbst habe 2 nicht-triviale Fehler im Shirley-Buch, 2-te Auflage gefunden [WS 05/06]
  - Fazit: mitdenken, nicht einfach direkt kopieren