

Ray Tracing I: Switching gears...



For your convenience

- Half-Time Summary Slides

Anmälningsskod (för www.studera.nu): CTH-23630
 Examiner: Ulf Assarsson
 ulffe@ce.chalmers.se

[Home](#) | [Schedule](#) | [Literature](#) | [Tutorials](#) | [Exam](#)

SCHEDULE:

- Link to always up-to-date schedule.
- All lectures are in ED, located in the EDIT-building at Campus Johanneberg
- [MAP](#) for lecture hall and tutorial rooms

Lectures

Mondays 13-15 in HB1, week 5-7

Wednesdays 13-15 in HB1, week 1-7.

Fridays 13-15 in HC3, week 1-7.

The following plan may change during the course. The links for the Bonus-OH are located under the table.

Bonus material is simply non-compulsory additional material that is fun or highlighting for the interested reader.

The column "Tutorial" states when it is wise to start working on the corresponding tutorial.

Lösenordsskyddade bonusfiler packas upp med lösenord "datografik".

All self-studies below are **non-compulsory**

NOTE: If you are using the 2:nd edition of Real-Time Rendering, for the compulsory RTR-chapter hints, see last year's schedule [here](#).

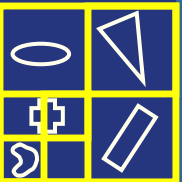
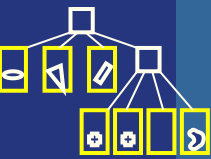
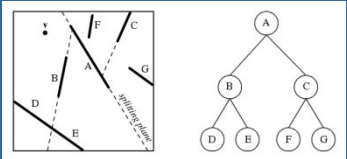


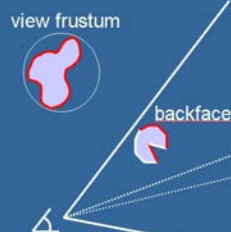
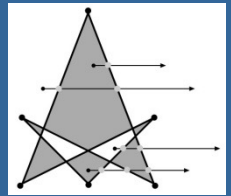
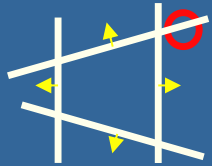
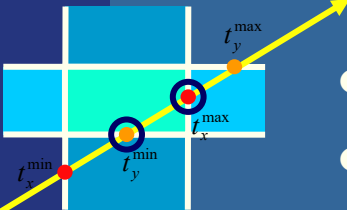
Lecture	Readings/Läsanvisningar	Tutorial	Deadlines
week 1			
Wed. lecture 1 - Introduction + Pipeline and OpenGL	Compulsory: RTR chapter 2, ch 15.2. pipeline.pdf , Bonus: OH 1-16, simpleapp.zip - the test application shown at lecture, crash course in VC++ .zip . Also, see A Quick Introduction to C++ with example code .		
Self studies - Languages (non-compulsory)	Languages.pdf (in Swedish) - Read briefly in and only if you find it interesting		

...

Fri. lecture 6 - Intersections and Spatial data structures	Compulsory: spatial.pdf , isect.pdf , RTR ch: 14: 14.1 (skip 14.1.4), 14.2, 14.3, 14.4, 14.5, 14.6 (skip 14.6.1, 14.6.2, 14.6.3), 14.7 RTR ch: 16 (skip: 16.1, 16.4, 16.7.2, 16.7.3, 16.10.2, 16.11, 16.12, 16.13.4, for 16.13.5 – see Sep Axis Theorem in slides, 16.14.1, 16.15, 16.17), 16.14 briefly. Bonus: OH 309-320		
Self studies	Bonus: Postscript , OH 17-26, OH 65-79 och OH 281-282.		
Half-Time Wrap-Up:	Half-time-wrapup slides. These slides correspond to the most important issues of each lecture so far in the course.		
week 4			
Wed lecture 7 - Ray Tracing 1		mini-tutorial 2	Tutorial 2
Self studies: Textures in Art of Illusion and Perlin-noise	Bonus: OH 175-200		
Fri. lecture 8 - Ray Tracing 2			
Self studies			

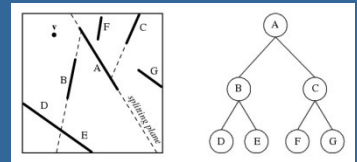
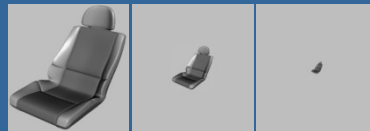
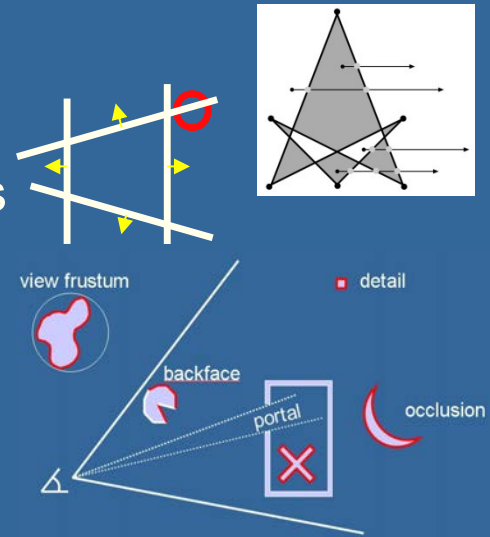
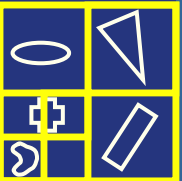
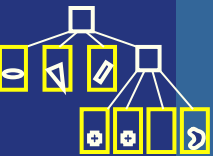
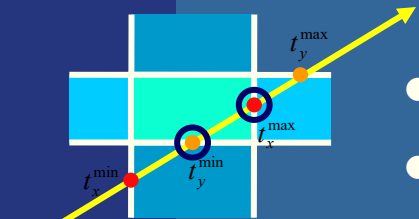
Typical Exam Questions

- Prev Lecture:
 - Describe **one** intersection test for
 - ray/triangle – (e.g. analytically, Jordans Cross theorem or summing angles)
 - Ray/box (slabs)
 - View Frustum Culling using spheres
 - Culling – VFC, Portal, Detail, Backface, Occlusion
 - What is LODs
 - Describe how to build and use BVHs, AABSP-tree, Polygon aligned BSP-tree.
 - Describe the octree/quadtree.



- Prev Lecture:

- Describe **one** intersection test for
 - ray/triangle – (e.g. analytically, Jordans Cross theorem or summing angles)
 - Ray/box (slabs)
 - View Frustum Culling using spheres
- Culling – VFC, Portal, Detail, Backface, Occlusion
- What is LODs
- Describe how to build and use BVHs, AABSP-tree, Polygon aligned BSP-tree.
- Describe the octree/quadtree.



What is ray tracing?

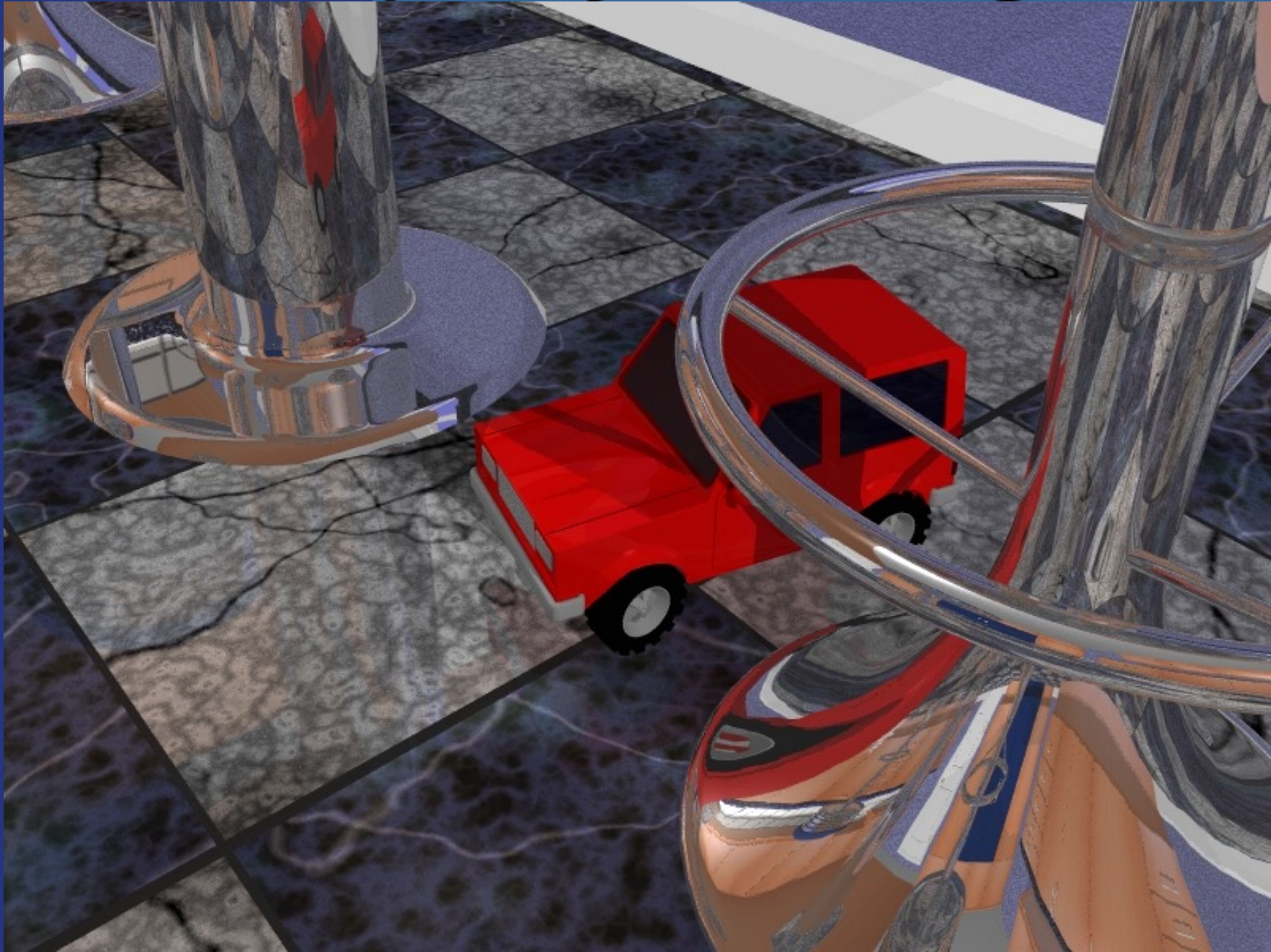
- Another rendering algorithm
 - Fundamentally different from polygon rendering (using e.g., OpenGL)
 - OpenGL
 - renders one triangle at a time
 - Z-buffer sees to it that triangles appear "sorted" from viewpoint
 - Local lighting --- per vertex
 - Ray tracing
 - Gives correct reflections!
 - Renders one pixel at a time
 - Sorts per pixel
 - Global lighting equation (reflections, shadows)

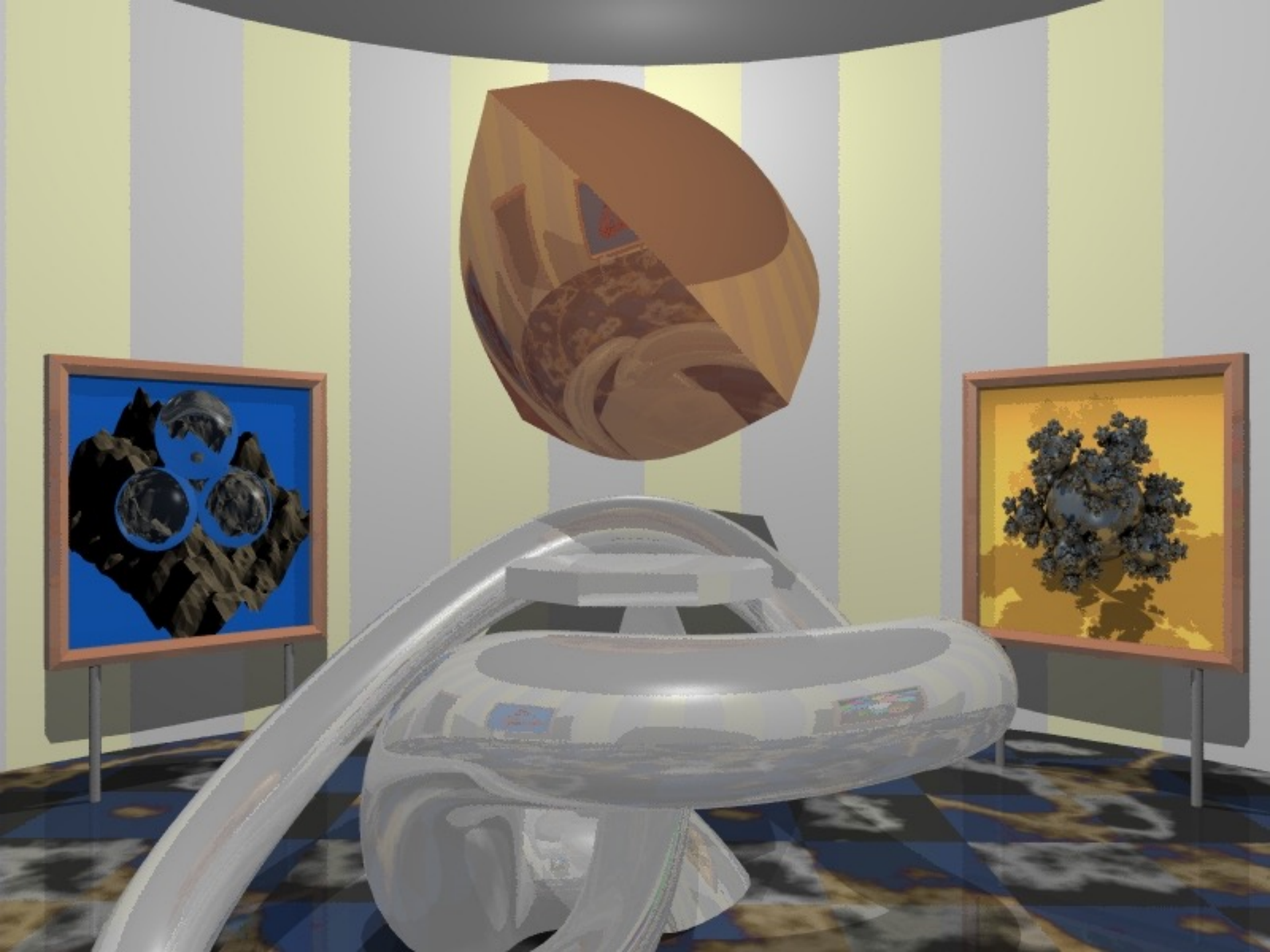


What is the point of ray tracing?

- Higher quality rendering
 - Global lighting equation (shadows, reflections, refraction)
 - Accurate shadows, reflections, refraction
 - More accurate lighting equations
- Is the base for more advanced algorithms
 - Global illumination, e.g., path tracing, photon mapping
- It is extremely simple to write a (naive) ray tracer
- A disadvantage: it is inherently slow!

Some simple, ray traced images...

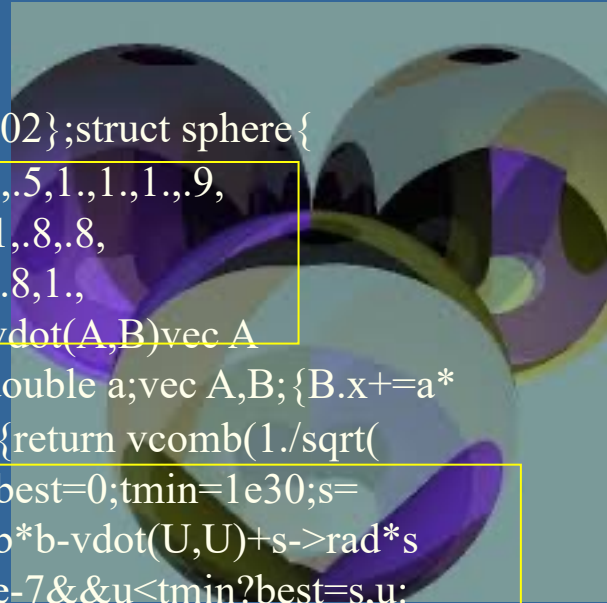




Again: it is simple to write a ray tracer!

– A la Paul Heckbert:

```
typedef struct {double x,y,z} vec; vec U, black, amb={.02,.02,.02}; struct sphere {
vec cen,color; double rad,kd,ks,kt,kl,ir} *s,*best,sph[]={0.,6.,.5,1.,1.,1.,.9,
.05,.2,.85,0.,1.7,-1.,8.,-.5,1.,.5,2,1.,.7,3,0.,.05,1.2,1.,8.,-.5,1.,8.,8,
1.,.3,.7,0.,0.,1.2,3.,-6.,15.,1.,8,1.,7,0.,0.,0.,.6,1.5,-3.,-3.,12.,.8,1.,
1.,5,0.,0.,.5,1.5,}; yx; double u,b,tmin,sqrt(),tan(); double vdot(A,B) vec A
,B; {return A.x*B.x+A.y*B.y+A.z*B.z;} vec vcomb(a,A,B) double a; vec A,B; {B.x+=a*
A.x; B.y+=a*A.y; B.z+=a*A.z; return B;} vec vunit(A) vec A; {return vcomb(1./sqrt(
vdot(A,A)),A,black);} struct sphere* intersect(P,D) vec P,D; {best=0;tmin=1e30;s=
sph+5; while(s-->sph) b=vdot(D,U=vcomb(-1.,P,s->cen)), u=b*b-vdot(U,U)+s->rad*s
->rad, u=u>0?sqrt(u):1e31, u=b-u>1e-7?b-u:b+u, tmin=u>=1e-7&&u<tmin?best=s,u:
tmin; return best;} vec trace(level,P,D) vec P,D; {double d,eta,e; vec N,color;
struct sphere*s,*l;if(!level--) return black;if(s=intersect(P,D)); else return
amb;color=amb; eta=s->ir;d=-vdot(D,N=vunit(vcomb(-1.,P=vcomb(tmin,D,P),s->cen
)));if(d<0)N=vcomb(-1.,N,black),eta=1/eta,d=-d;l=sph+5; while(l-->sph)if((e=l
->kl*vdot(N,U=vunit(vcomb(-1.,P,l->cen))))>0&&intersect(P,U)==l)color=vcomb(e
,l->color,color);U=s->color;color.x*=U.x;color.y*=U.y;color.z*=U.z;e=1-eta*
eta*(1-d*d);return vcomb(s->kt,e>0?trace(level,P,vcomb(eta,D,vcomb(eta*d-sqrt
(e),N,black))):black,vcomb(s->ks,trace(level,P,vcomb(2*d,N,D)),vcomb(s->kd,
color,vcomb(s->kl,U,black))));} main() {printf("%d %d\n",32,32);while(yx<32*32)
U.x=yx%32-32/2,U.z=32/2-yx++/32,U.y=32/2/tan(25/114.5915590261),U=vcomb(255.,
trace(3,black,vunit(U)),black),printf("%.0f %.0f %.0f\n",U);} /*minray!*/
```

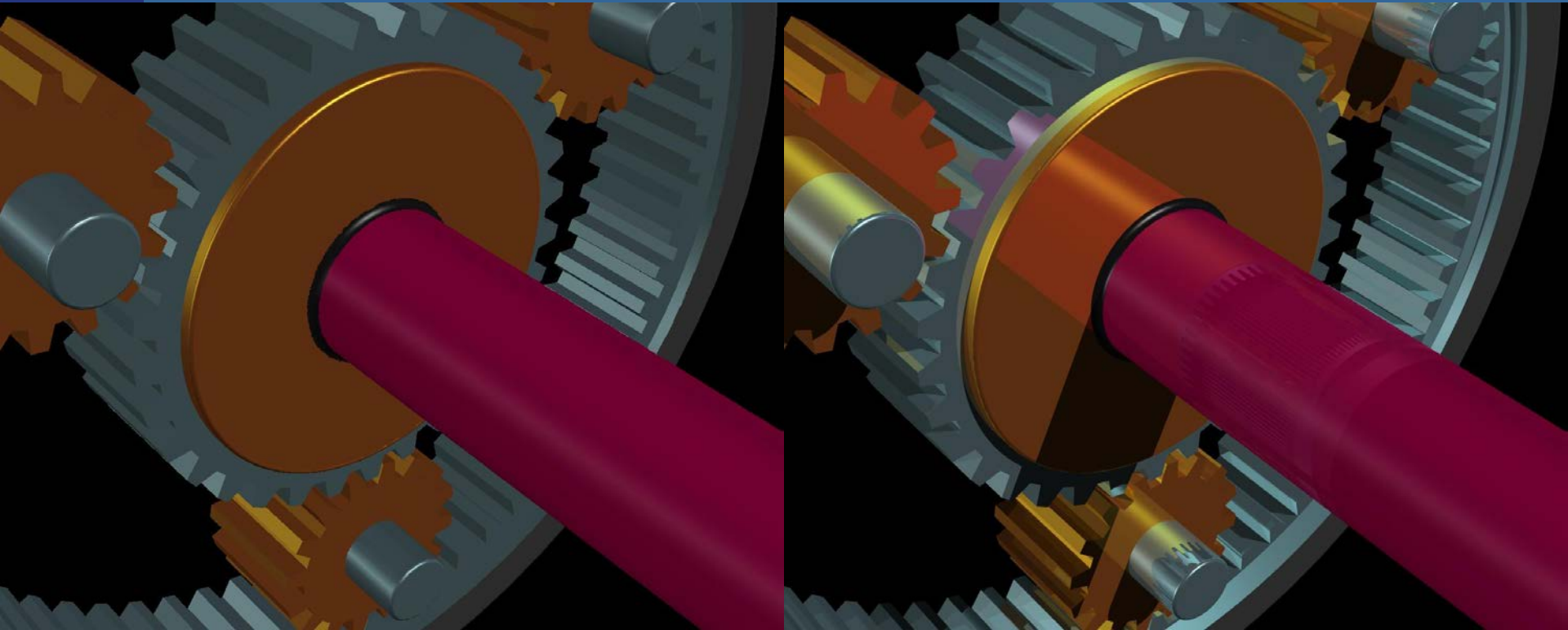


Which rendering algorithm will win at the end of the day?

- Ray tracing or polygon rendering?
- Ray tracing is:
 - Slow
 - But realistic
 - Therefore, focus is on creating faster algorithms, and possible hardware acceleration (GPU, RPU)
- Polygon rendering (OpenGL) is:
 - Fast (simple to build hardware)
 - Not that realistic
 - Therefore, focus is on creating more realistic images using graphics hardware
- Answer: right now, it depends on what you want, but for the future, no one really knows

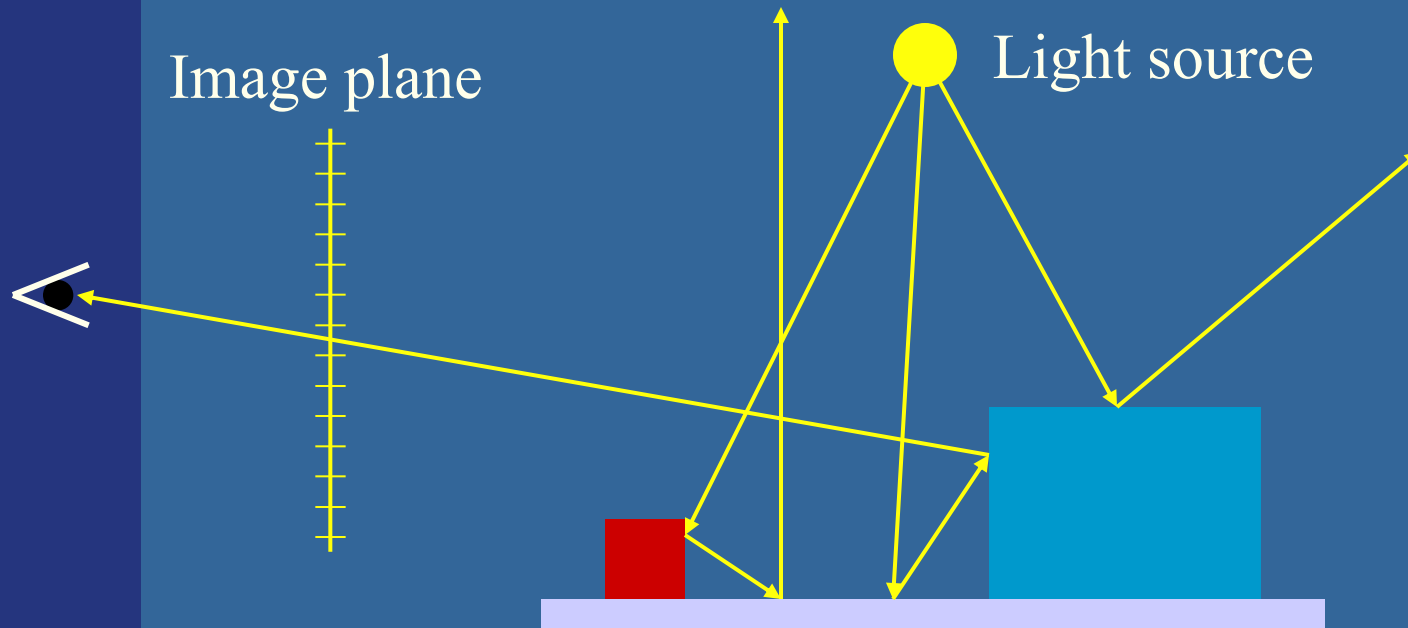
Side by side comparison

Images courtesy of Eric Haines



To be physically correct, follow photons from light sources...

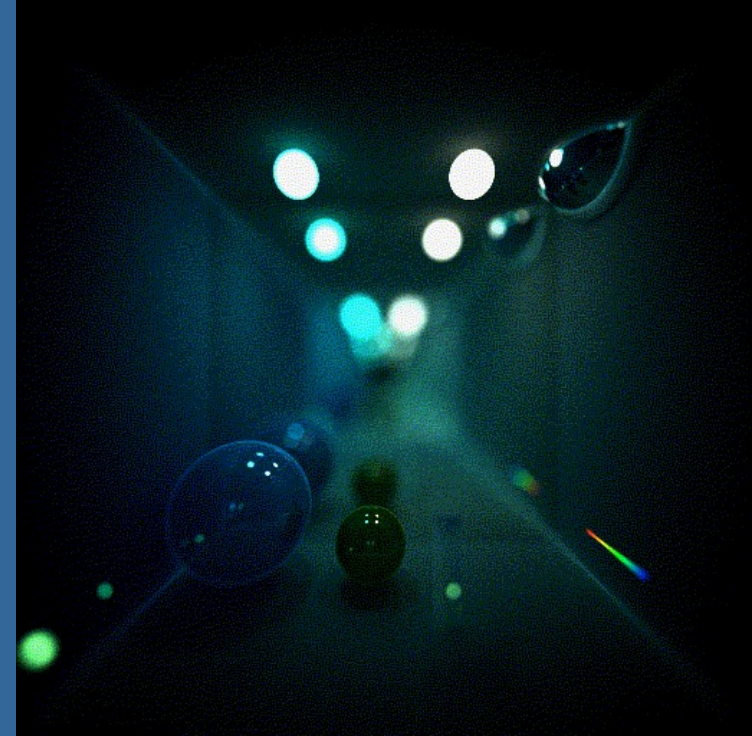
- Not what we do for a simple ray tracer
 - Though this is almost what we do for more advanced techniques (photon mapping)



- Not effective, not many rays will arrive at the eye

This image was generated in 1991 by simulating the motion of 29.8 Billion photons in a room. The room is 2 meters cubed with a 30 cm aperture in one wall. The opposite and adjacent walls are mirrors, so this is a 'tunnel of mirrors'. The depth of field is very shallow. In the foreground is a prism, resting on the floor. A beam of light emerges from the left wall, goes through the prism and makes a spectrum on the right wall. About 1 in 177 photons made it through the aperture.

The image took 100 Sun SparcStation1s 1 month to generate using background processing time. This represents 10 CPU years of processing time. If the lights are 25 watt bulbs this represents a few picoseconds of time.



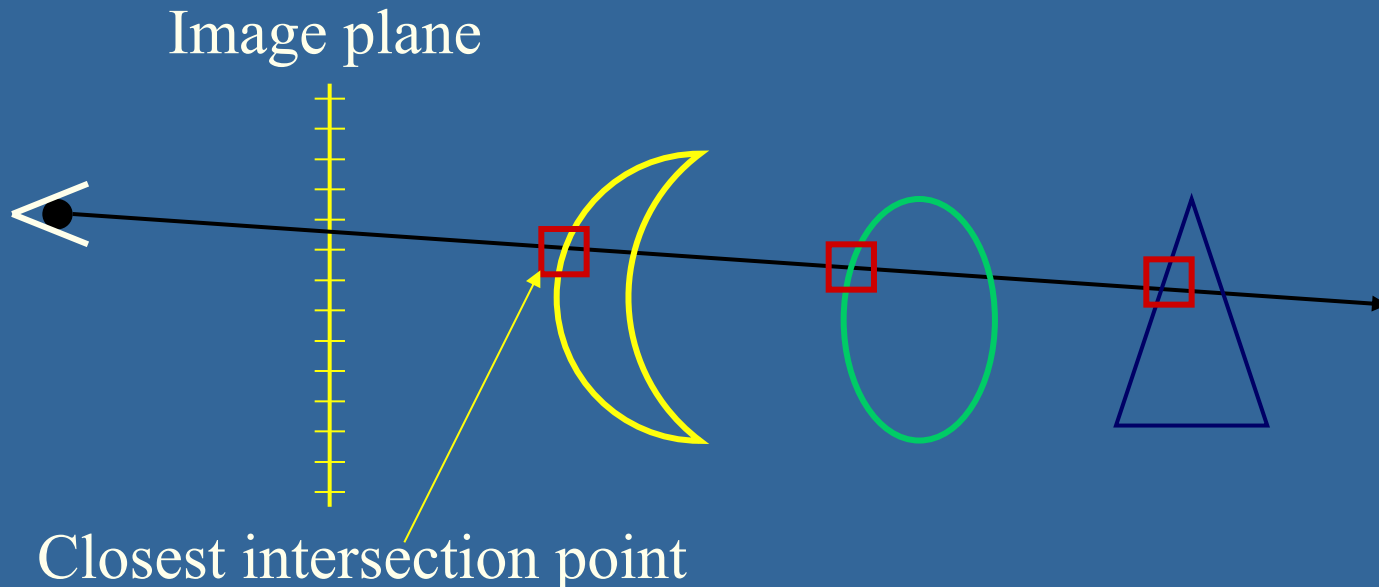
29.8 Billion photons

Same image but with 382 Billion Photons



Follow photons backwards from the eye: treat one pixel at a time

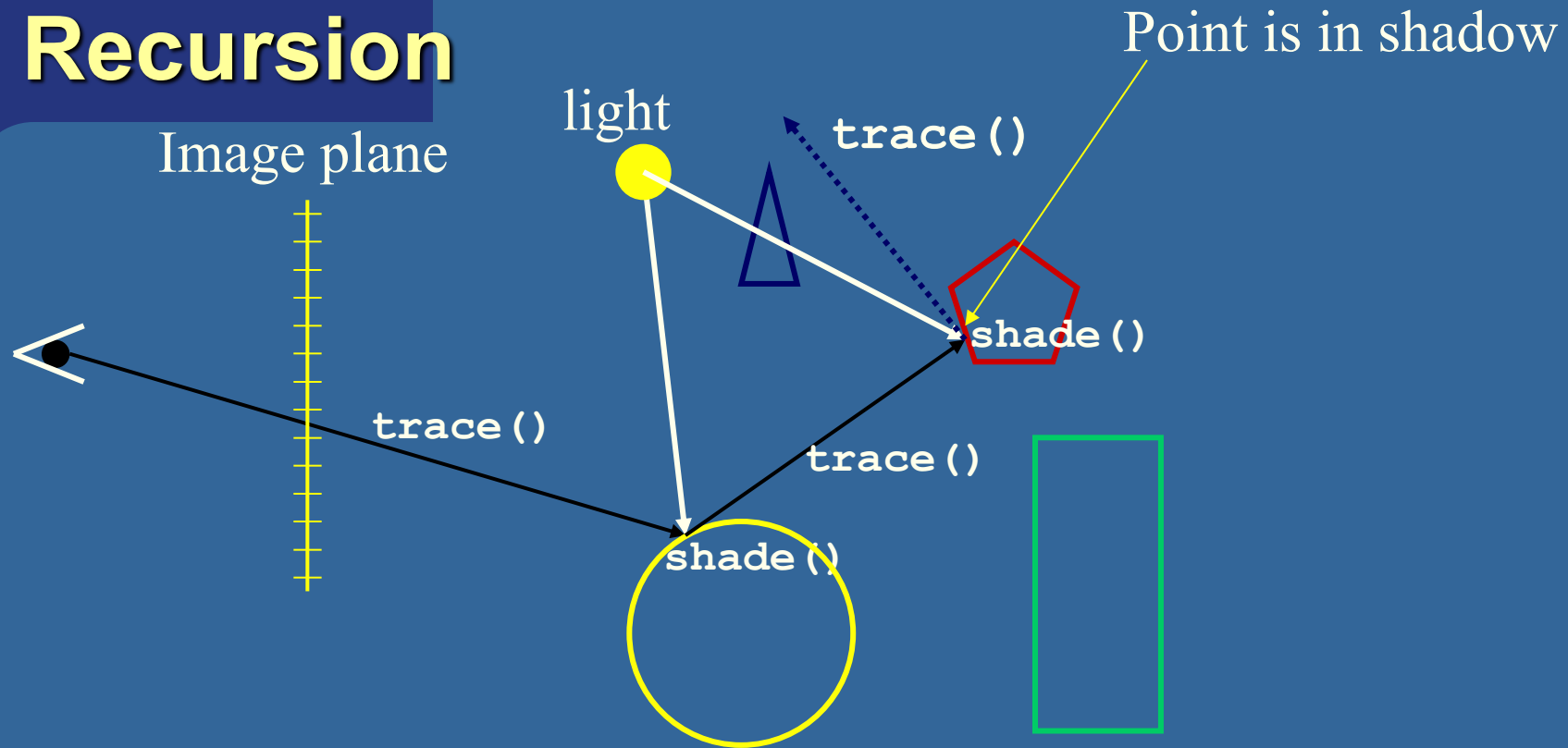
- Rationale: find photons that arrive at each pixel
- How do one find the visible object at a pixel?
- With intersection testing
 - Ray, $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$, against geometrical objects
 - Use object that is closest to camera!
 - Valid intersections have $t > 0$
 - t is a signed distance



Finding closest point of intersection

- Naively: test all geometrical objects in the scene against each ray, use closest point
 - Very very slow!
- Be smarter:
 - Use spatial data structures, e.g.:
 - Bounding volume hierarchies (BVH)
 - (Octrees), Sparse Voxel Octrees
 - kd trees
 - Grids (not yet treated)
 - Or a combination (hierarchies) of those
- We will return to this topic a little later

`trace()` and `shade()` : Recursion



- First call `trace()` to find first intersection
- `trace()` then calls `shade()` to compute lighting
- `shade()` then calls `trace()` for reflection and refraction directions

trace() in detail



```
Color trace(Ray R)
```

```
{
```

```
    float t;
```

```
    Object O; // typically a triangle
```

```
    Color col;
```

```
    bool hit=findClosestIntersection(R,&t,&O);
```

```
    if(hit)
```

```
    {
```

```
        // Compute intersection point P
```

```
        Vec3f P = R.origin() + t*R.direction();
```

```
        // Compute normal at intersection point
```

```
        Vec3f N = computeNormal(P,O);
```

```
        // flip normal if pointing in wrong dir.
```

```
        if(dot(N,R.direction()) > 0.0) N=-N;
```

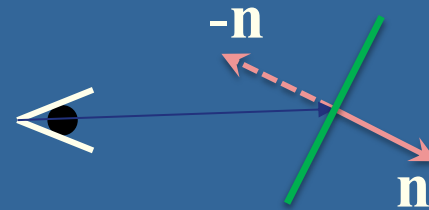
```
        col=shade(R,O,P,N);
```

```
    }
```

```
    else col=background_color;
```

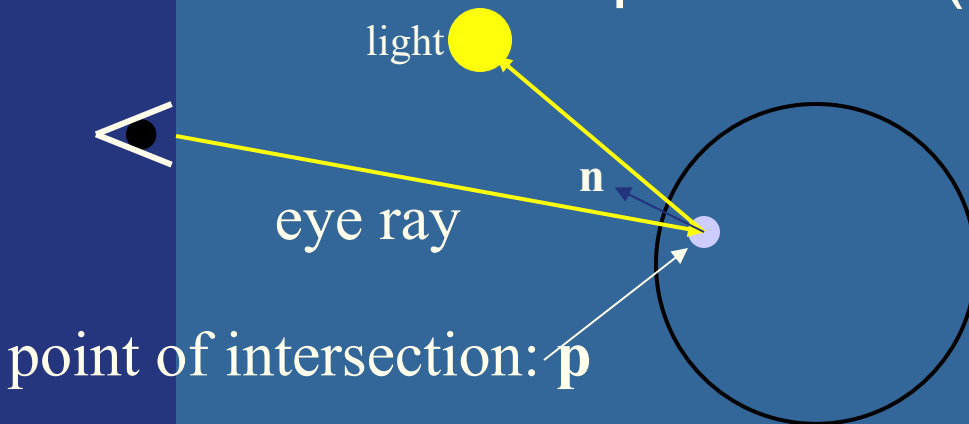
```
    return col;
```

```
}
```



In `trace()`, we need a function `findClosestIntersection()`

- Use intersection testing (from a previous lecture) for rays against objects
- Intersection testing returns signed distance(s), t , to the object
- Use the t that is smallest, but > 0
- Naive: test all objects against each ray
 - Better: use spatial data structures (more later)
- Precision problems (exaggerated):



The point, \mathbf{p} , can be incorrectly self-shadowed, due to imprecision

Solution: after \mathbf{p} has been computed, update as: $\mathbf{p}' = \mathbf{p} + \epsilon \mathbf{n}$

(\mathbf{n} is normal at \mathbf{p} , ϵ is small number > 0)

Example of Surface Acne

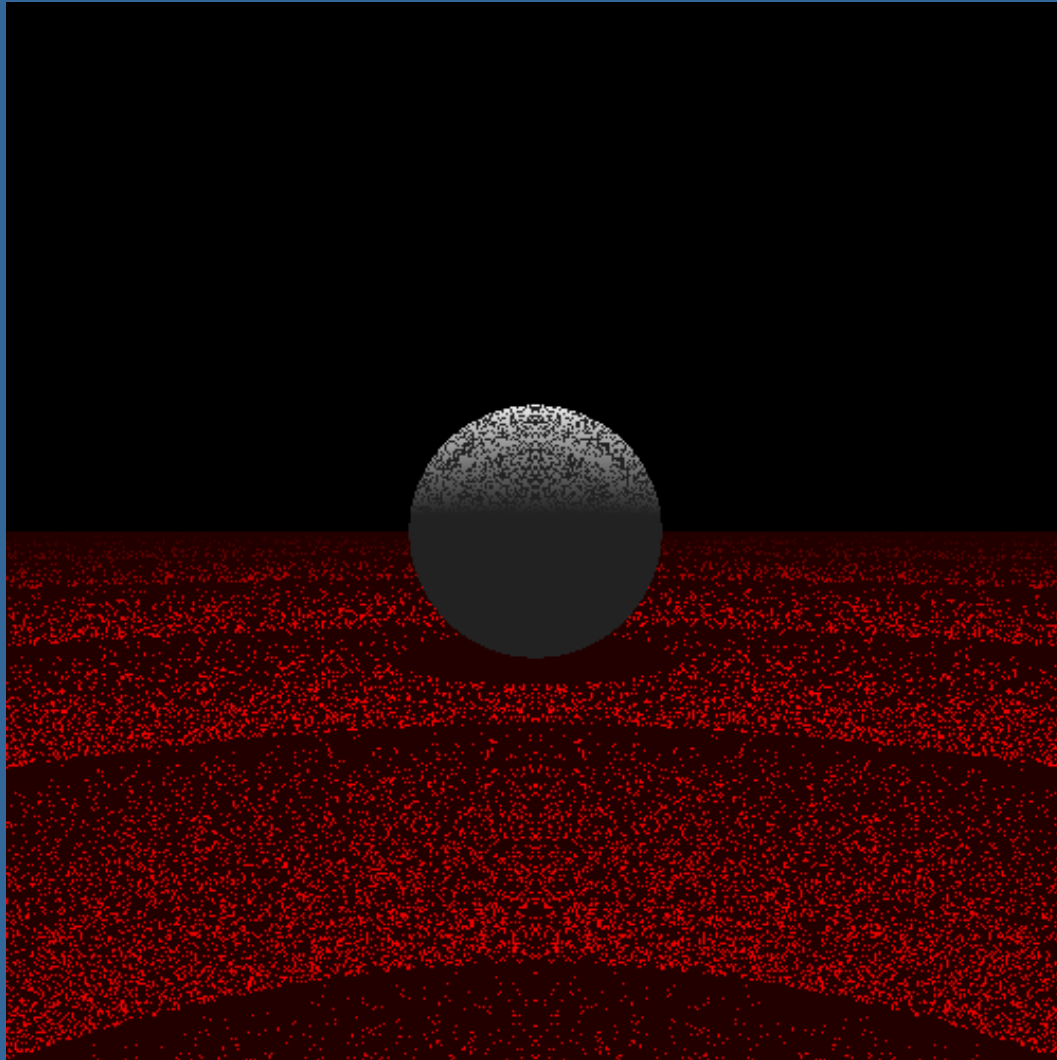


Image from Joe Doliner

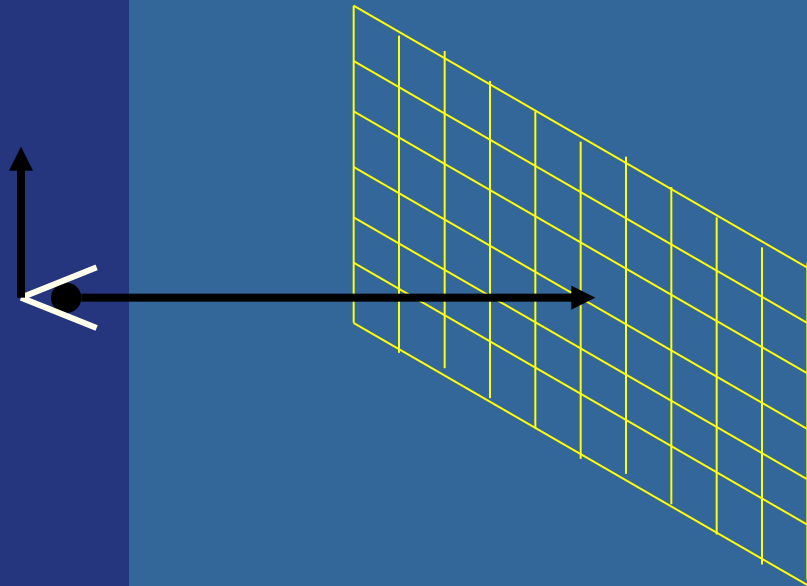
shade () in detail

```
Color shade(Ray R, Mtrl &m, Vector P,N)
{
    Color col;
    Vector refl,refr;
    for each light L
    {
        if(not inShadow(L,P))
            col+=DiffuseAndSpecular();

    }
    col+=AmbientTerm();
    if(recursed_too_many_times()) return col;
    refl=reflectionVector(R,N);
    col+=m.specular_color()*trace(refl);
    refr=computeRefractionVector(R,N,m);
    col+=m.transmission_color()*trace(refr);
    return col;
}
```


Who calls `trace()` or `shade()`?

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



- Use camera parameters to compute rays
 - Resolution, fov, camera direction & position & up

When does recursion stop?

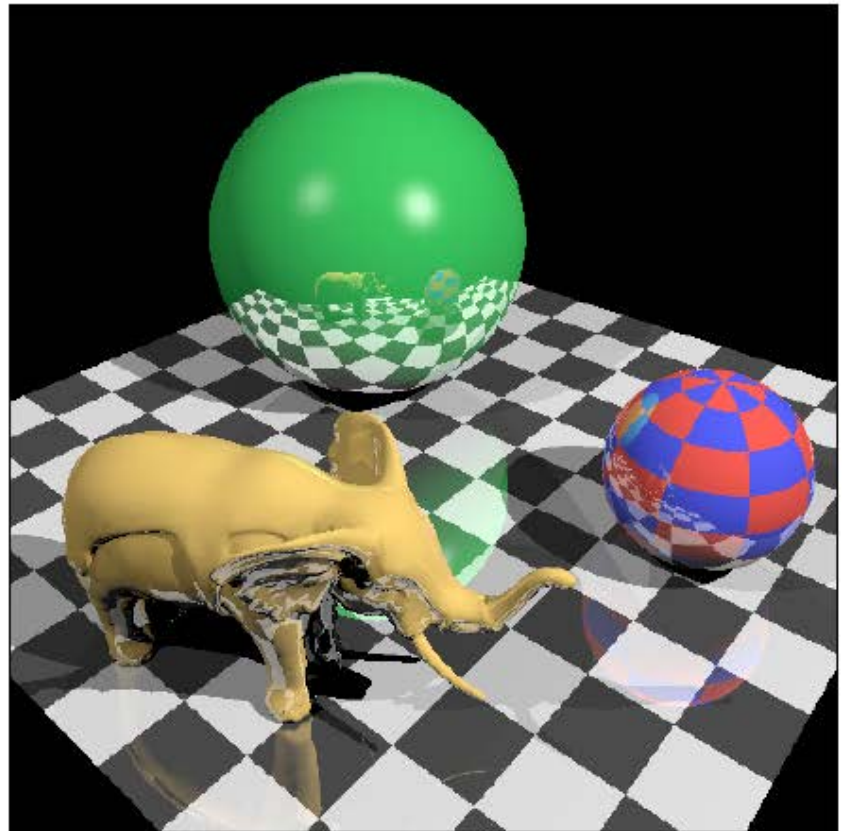
- Recurse until ray does not hit something?
 - Does not work for closed models
- One solution is to allow for max N levels of recursion
 - N=3 is often sufficient (sometimes 10 is sufficient)
- Another is to look at material parameters
 - E.g., if specular material color is (0,0,0), then the object is not reflective, and we don't need to spawn a reflection ray
 - More systematic: send a weight, w , with recursion
 - Initially $w=1$, and after each bounce,
 $w = w * O.\text{specular_color}()$; and so on.
 - Will give faster rendering, if we terminate recursion when weight is too small (say <0.05).
 - Or use a weight per rgb, $\mathbf{w}=(1,1,1)$ and stop when $\max(\mathbf{w}_r, \mathbf{w}_g, \mathbf{w}_b) < 0.05$ or when $\|\mathbf{w}\| < 0.05$.

When to stop recursion



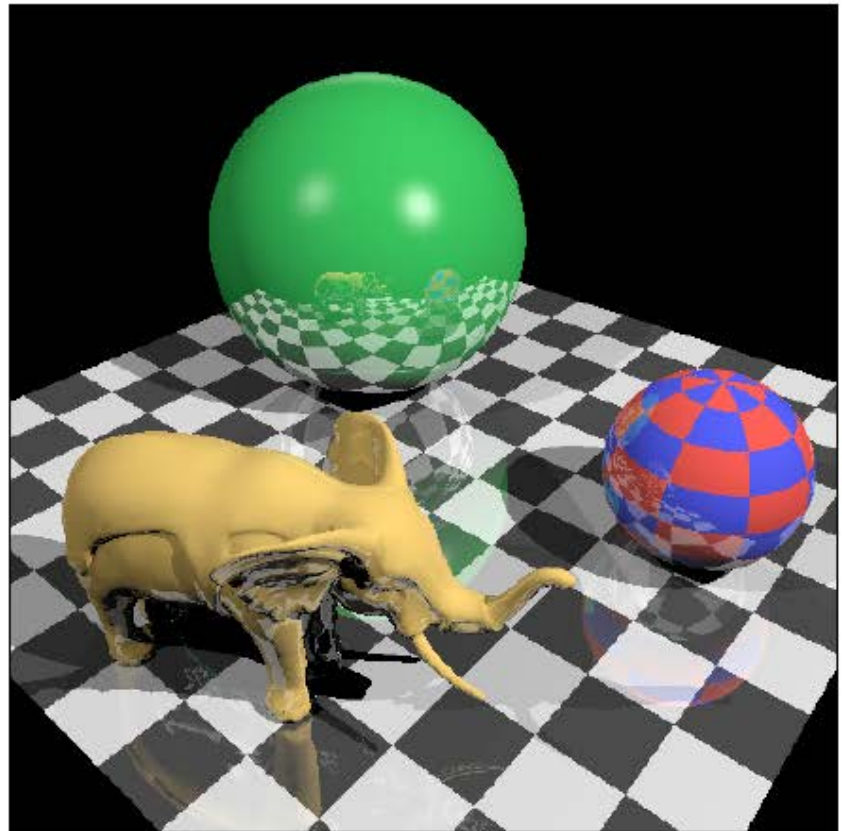
Reflections - Result

- Direct illumination with shadows + reflections
- Depth cutoff = 1



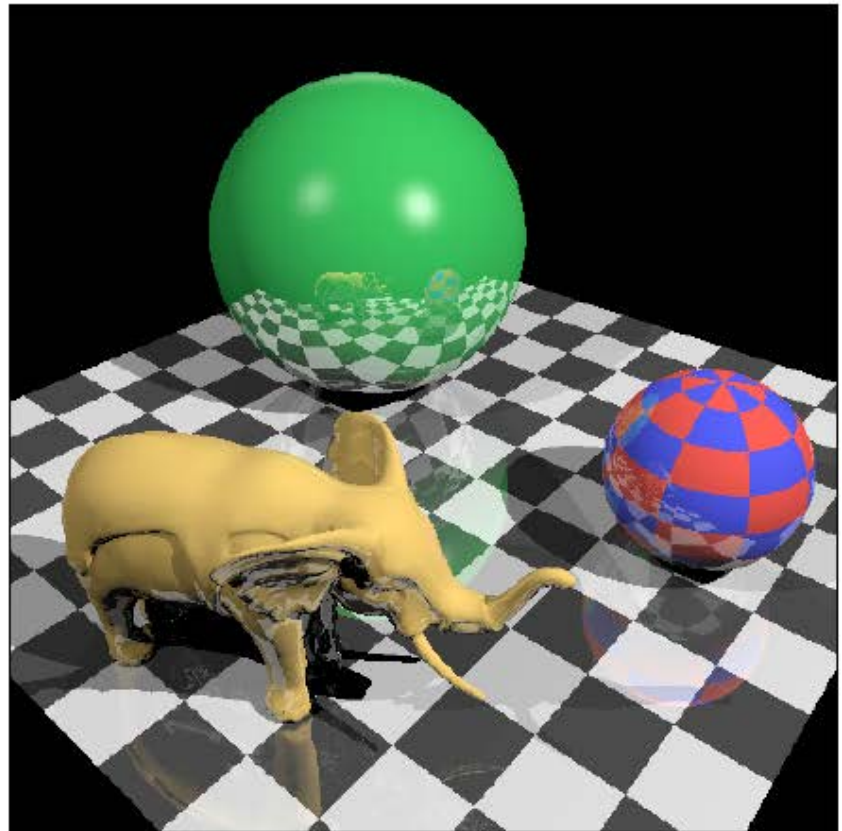
Reflections - Result

- Direct illumination with shadows + reflections
- Depth cutoff = 2



Reflections - Result

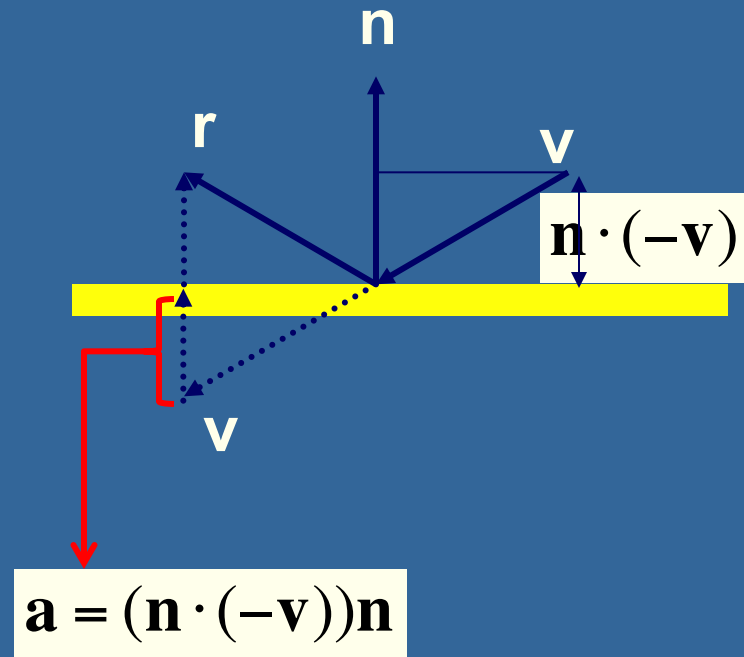
- Direct illumination with shadows + reflections
- Depth cutoff = 3



Reflection vector (recap)

- Reflecting the incoming ray \mathbf{v} around \mathbf{n} :
- Note that the incoming ray is sometimes called $-\mathbf{v}$ depending on the direction of the vector.
- \mathbf{r} can be computed as $\mathbf{v} + (2\mathbf{a})$. I.e.,

$$\mathbf{r} = \mathbf{v} - 2(\mathbf{n} \cdot \mathbf{v})\hat{\mathbf{n}}$$



Refraction:

Need a transmission direction vector, \mathbf{t}

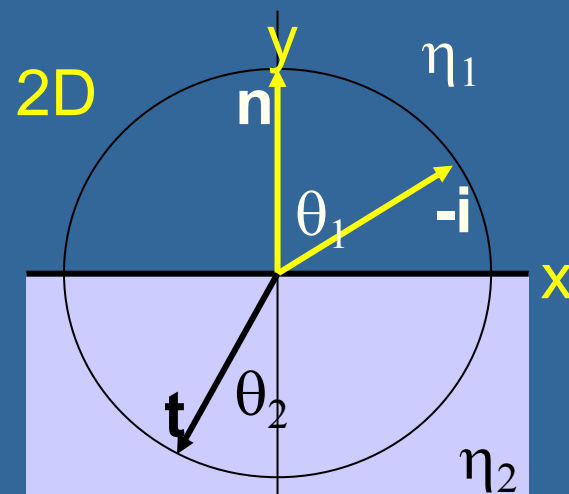
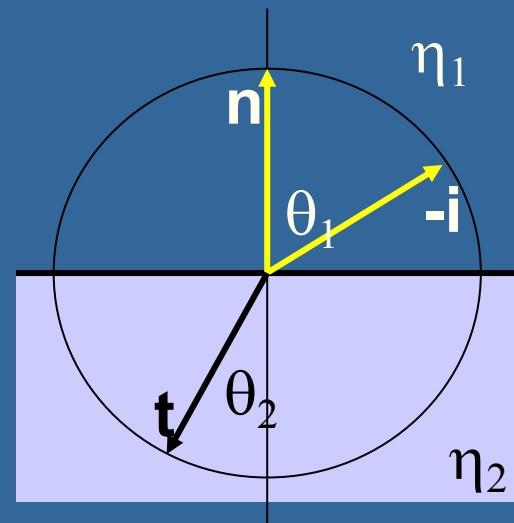
- \mathbf{n} , \mathbf{i} , \mathbf{t} are unit vectors
- η_1 & η_2 are refraction indices
- Snell's law says that:
 - $\sin(\theta_2)/\sin(\theta_1) = \eta_1/\eta_2 = \eta$, where η is relative refraction index.
- How can we compute the refraction vector \mathbf{t} ?

- This would be easy in 2D:

- $t_x = -\sin(\theta_2)$

- $t_y = -\cos(\theta_2)$

- I.e., $\mathbf{t} = -\sin(\theta_2)\hat{\mathbf{x}} - \cos(\theta_2)\hat{\mathbf{y}}$



Refraction:

$$\mathbf{t} = -\sin(\theta_2)\hat{\mathbf{x}} - \cos(\theta_2)\hat{\mathbf{y}}$$

- But we are in 3D, not in 2D!
- So, the solution will look like:

$$\mathbf{t} = -\sin(\theta_2)\hat{\mathbf{v}}_1 - \cos(\theta_2)\hat{\mathbf{v}}_2$$

$$\mathbf{v}_2 = \mathbf{n}$$

$$\mathbf{v}_1 = \text{normalize}(-\mathbf{i} + (\mathbf{i} \cdot \mathbf{n})\mathbf{n})$$

So we could consider us done. But let's continue simplifying to avoid expensive trigonometric functions (sin, cos, arcsin). Only use cheap $\cos(\theta_1) = (-\mathbf{i} \cdot \mathbf{n})$.

1. Remove normalization to use Snell's law in step 3 below:

$$-\mathbf{v}_1 = (-\mathbf{i} + (\mathbf{i} \cdot \mathbf{n})\mathbf{n}) / \sin(\theta_1)$$

2. Plugin \mathbf{v}_1 into \mathbf{t}

$$\Rightarrow \mathbf{t} = \sin(\theta_2) (-\mathbf{i} + (\mathbf{i} \cdot \mathbf{n})\mathbf{n}) / \sin(\theta_1) - \cos(\theta_2)\mathbf{n}$$

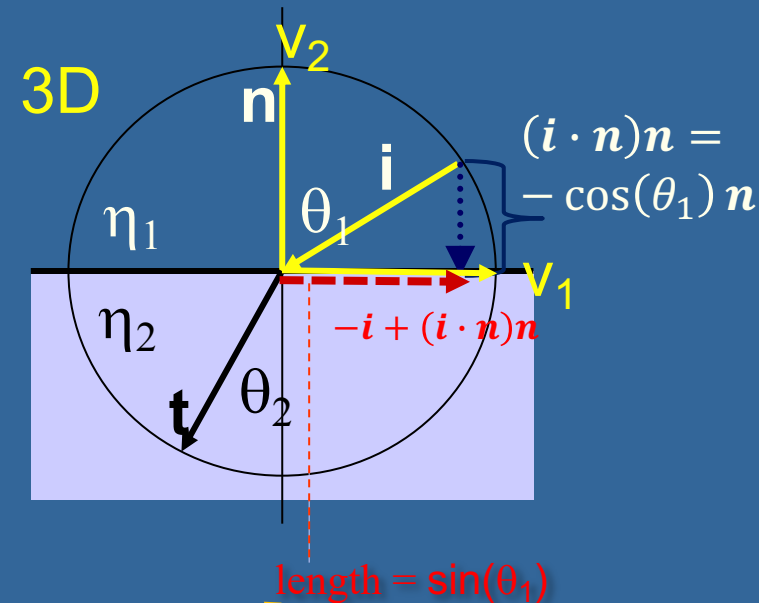
3. Use Snell: $\sin(\theta_2)/\sin(\theta_1) = \eta$

$$\Rightarrow \mathbf{t} = \eta(-\mathbf{i} + (\mathbf{i} \cdot \mathbf{n})\mathbf{n}) - \cos(\theta_2)\mathbf{n}$$

4. Simplify $\cos(\theta_2)$ using Trig1: $\cos(\theta_2)^2 = 1 - \sin(\theta_2)^2$ and Snell: $\sin(\theta_2) = \eta \sin(\theta_1)$:

$$\Rightarrow \cos(\theta_2) = \sqrt{1 - \sin(\theta_2)^2} = [\text{Snell}] = \sqrt{1 - \eta^2 \sin(\theta_1)^2} = [\text{Trig1}] = \sqrt{1 - \eta^2 (1 - \cos(\theta_1)^2)}$$

$$\Rightarrow \mathbf{t} = \eta(-\mathbf{i} + (\mathbf{i} \cdot \mathbf{n})\mathbf{n}) - \sqrt{1 - \eta^2 (1 - \cos(\theta_1)^2)} \mathbf{n} \quad // \text{ replacing } \cos(\theta_2) \text{ with an expression of } \cos(\theta_1) \\ // \text{ which is fast to compute since } \cos(\theta_1) = (-\mathbf{i} \cdot \mathbf{n})$$



Refraction

- Thus:

$$\mathbf{t} = \eta \mathbf{i} + (\eta \cos(\theta_1) - \sqrt{1 - \eta^2(1 - (\cos(\theta_1))^2)}) \mathbf{n}$$

This is fast to compute since

$$\cos(\theta_1) = -\mathbf{n} \cdot \mathbf{i}$$

which only requires a simple dot product

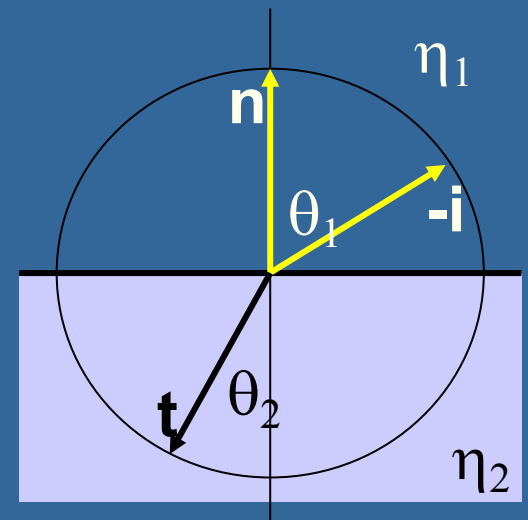
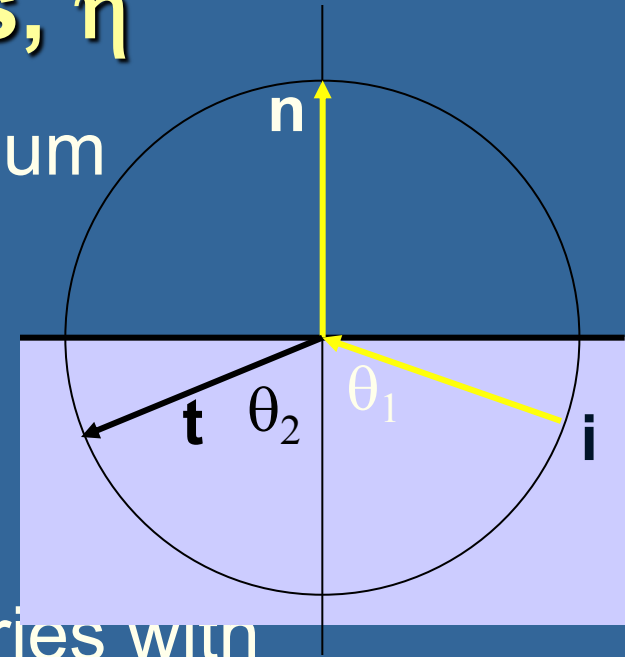


Image with a refractive object

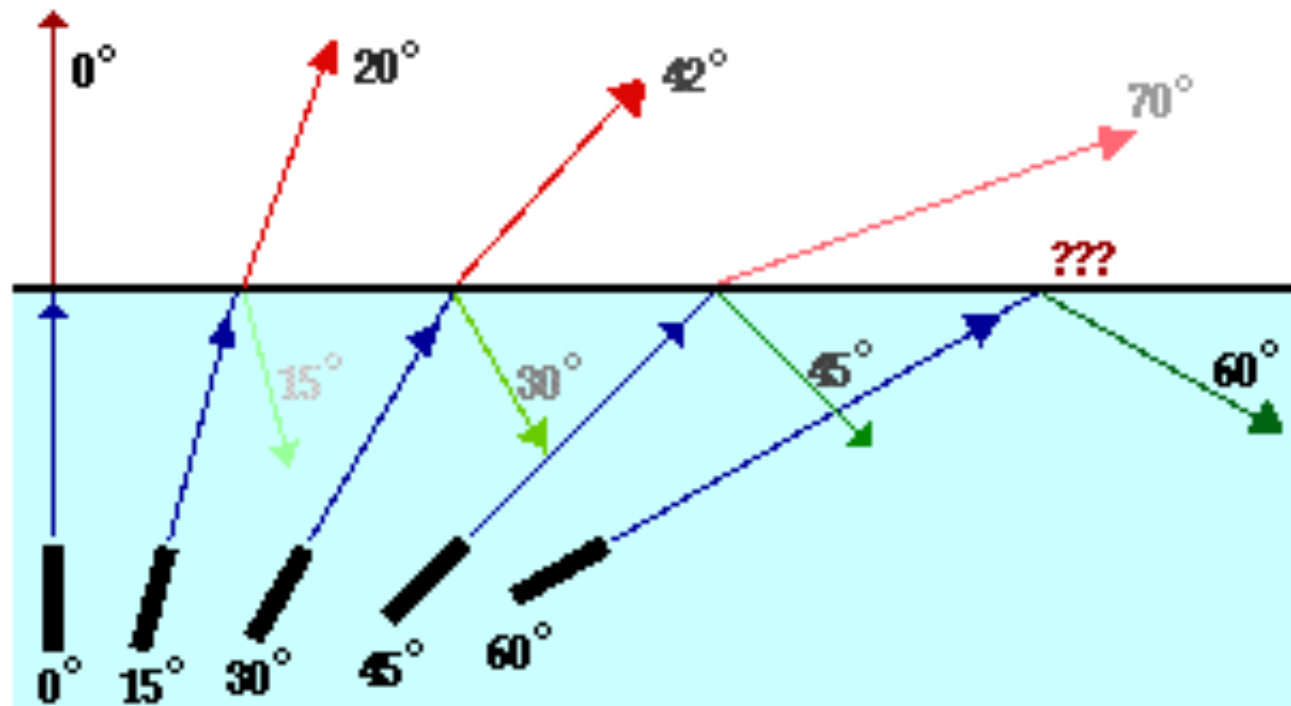


Some refraction indices, η

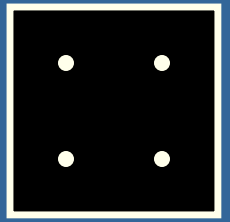
- Measured with respect to vacuum
 - Air: 1.0003
 - Water: 1.33
 - Glass: around 1.45 – 1.65
 - Diamond: 2.42
 - Salt: 1.54
- Note 1: the refraction index varies with wavelength for metals, i.e., one index per color channel, RGB.
- Note 2: can get Total Internal Reflection (TIR)
 - Means no transmission, only reflection
 - TIR occurs when the square root has an imaginary solution.
 - Or put differently:
 - $\theta_2 = \arcsin(\eta \sin(\theta_1))$
 - TIR occurs when $|\eta \sin(\theta_1)| > 1$, i.e., $\arcsin()$ undefined



As the angle of incidence increases from 0 to greater angles ...

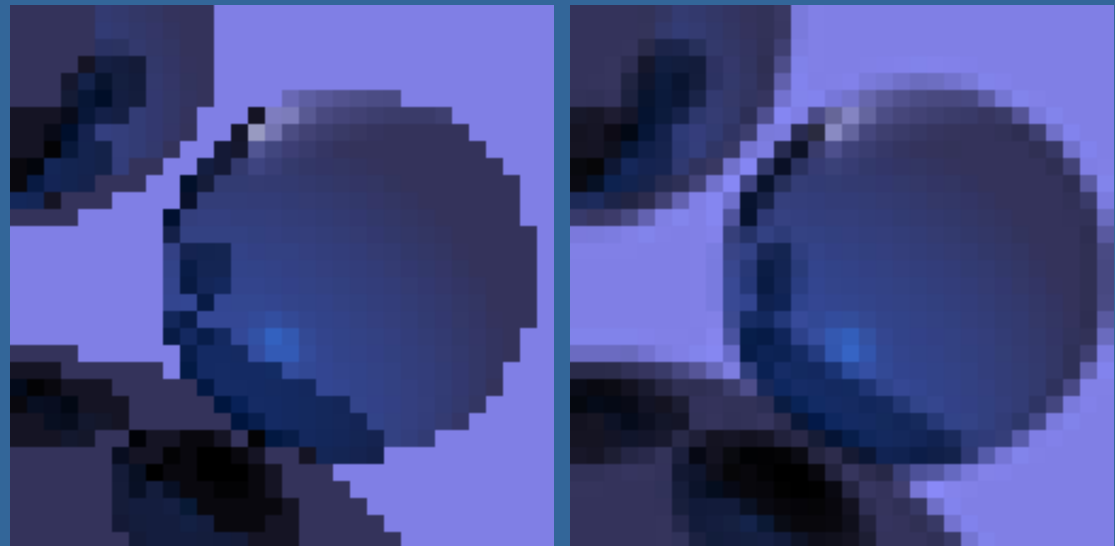


- ...the refracted ray becomes dimmer (there is less refraction)
- ...the reflected ray becomes brighter (there is more reflection)
- ...the angle of refraction approaches 90 degrees until finally a refracted ray can no longer be seen.



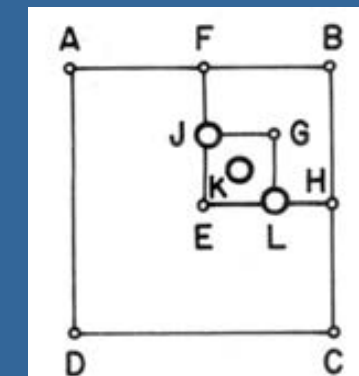
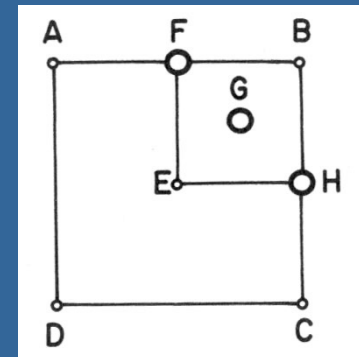
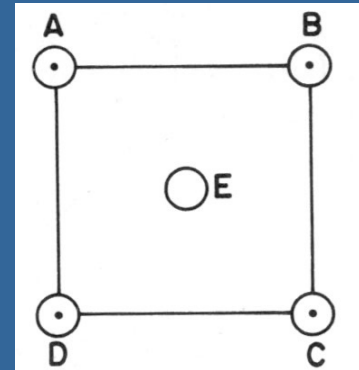
Supersampling

- Evenly distribute ray samples over pixel
- Use box (or tent filter) to find pixel color
- More samples gives better quality
 - Costs more time to render
- Example of 4x4 samples against 1 sample:



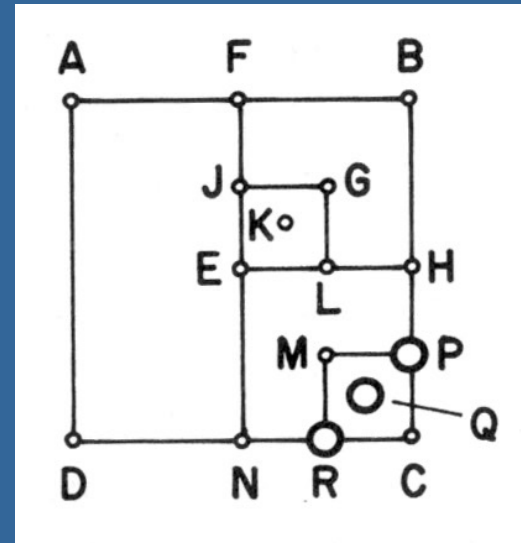
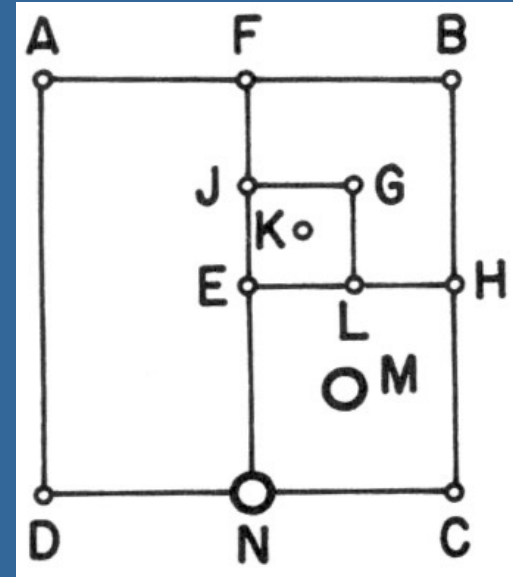
Be a bit smarter, make it cheaper: Adaptive supersampling (1)

- Quincunx sampling pattern to start with
 - 2 samples per pixel, 1 in center, 1 in upper-left
 - Note: adaptive sampling is not feasible in graphics hardware, but simple in a ray tracer
- Colors of AE, DE are quite similar, so don't waste more time on those.
- The colors of B & E are different, so add more samples there with the same sampling pattern
- Same thing again, check FG, BG, HG, EG: only EG needs more sampling
- So, add rays for J, K, and L



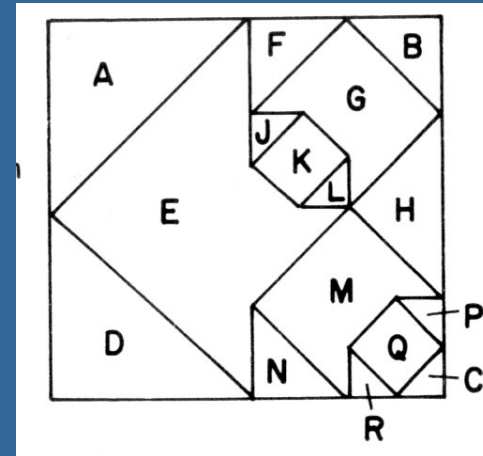
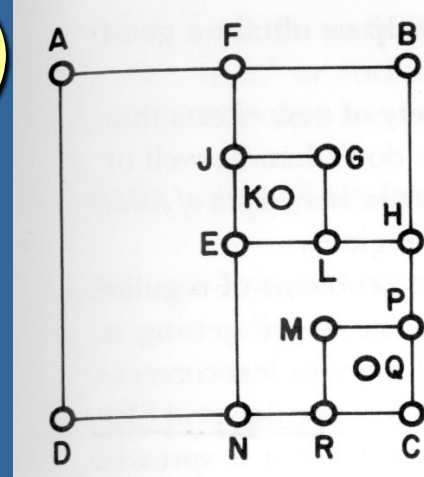
Adaptive supersampling (2)

- C & E were different too
- Add N & M
- Compare EM, HM, CM, NM
- C & M are too different
- So add rays at P, Q, and R
- At this point, we consider the entire pixel to be sufficiently sampled
- Time to weigh (filter) the colors of all rays



Adaptive supersampling (3)

- Final sample pattern for pixel:
- How filter the colors of the rays?
- Think of the pattern differently:
- And use the area of each ray sample as its weight:



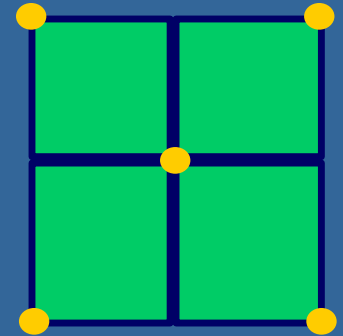
$$\frac{1}{4} \left(\frac{A+E}{2} + \frac{D+E}{2} + \frac{1}{4} \left[\frac{F+G}{2} + \frac{B+G}{2} + \frac{H+G}{2} + \frac{1}{4} \left\{ \frac{J+K}{2} + \frac{G+K}{2} + \frac{L+K}{2} + \frac{E+K}{2} \right\} \right] \right. \\ \left. + \frac{1}{4} \left[\frac{E+M}{2} + \frac{H+M}{2} + \frac{N+M}{2} + \frac{1}{4} \left\{ \frac{M+Q}{2} + \frac{P+Q}{2} + \frac{C+Q}{2} + \frac{R+Q}{2} \right\} \right] \right)$$

Adaptive Supersampling

Pseudo code:

```
Color AdaptiveSuperSampling() {
```

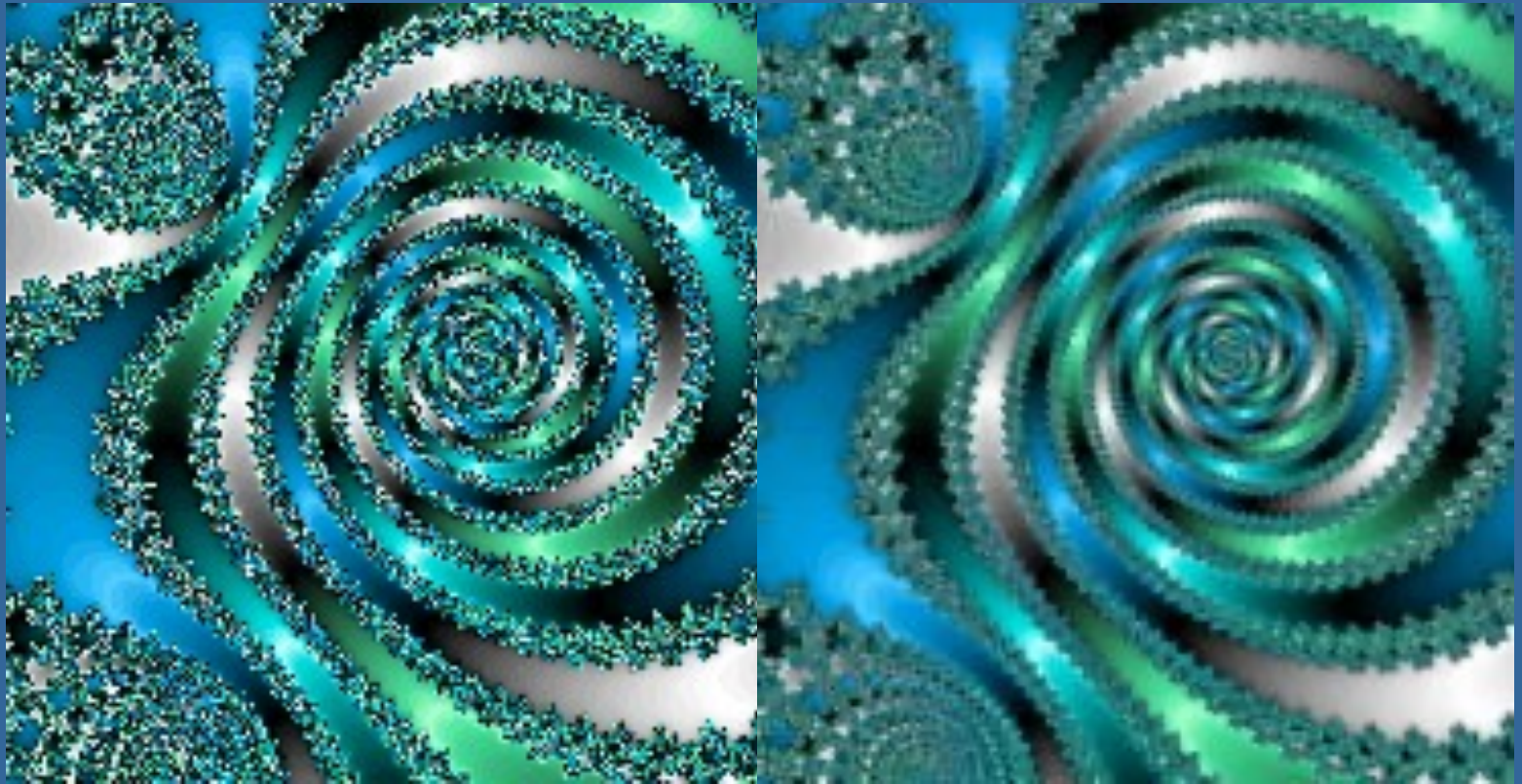
- Make sure all 5 samples exist
 - (Shoot new rays along diagonal if necessary)
 - Color col = black;
 - For each quad i
 - If the colors of the 2 samples are fairly similar
 - $col += (1/4) * (\text{average of the two colors})$
 - Else
 - $col += (1/4) * \text{adaptiveSuperSampling}(\text{quad}[i])$
 - return col;
- ```
}
```



# Caveats with adaptive supersampling (4)

- May miss really small objects anyway
- It's still supersampling, but smart supersampling
  - Cannot fool Nyquist!
  - Only reduce aliasing – does not eliminate it

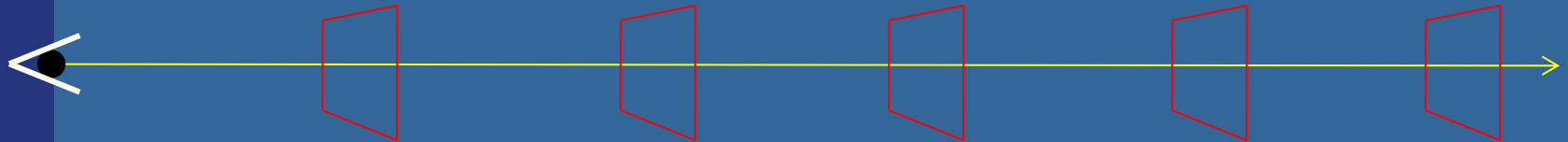
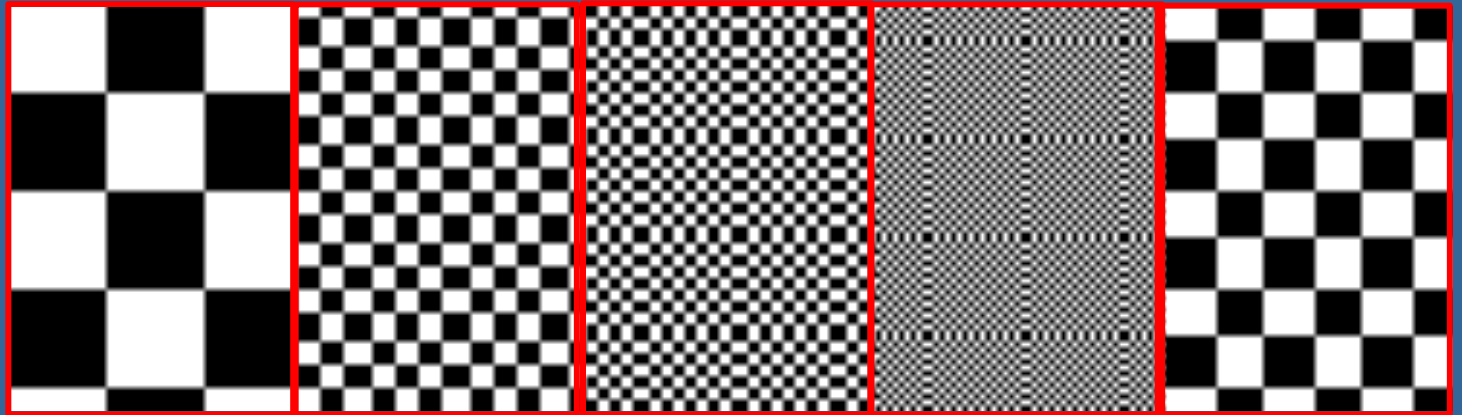
# Antialiasing - example



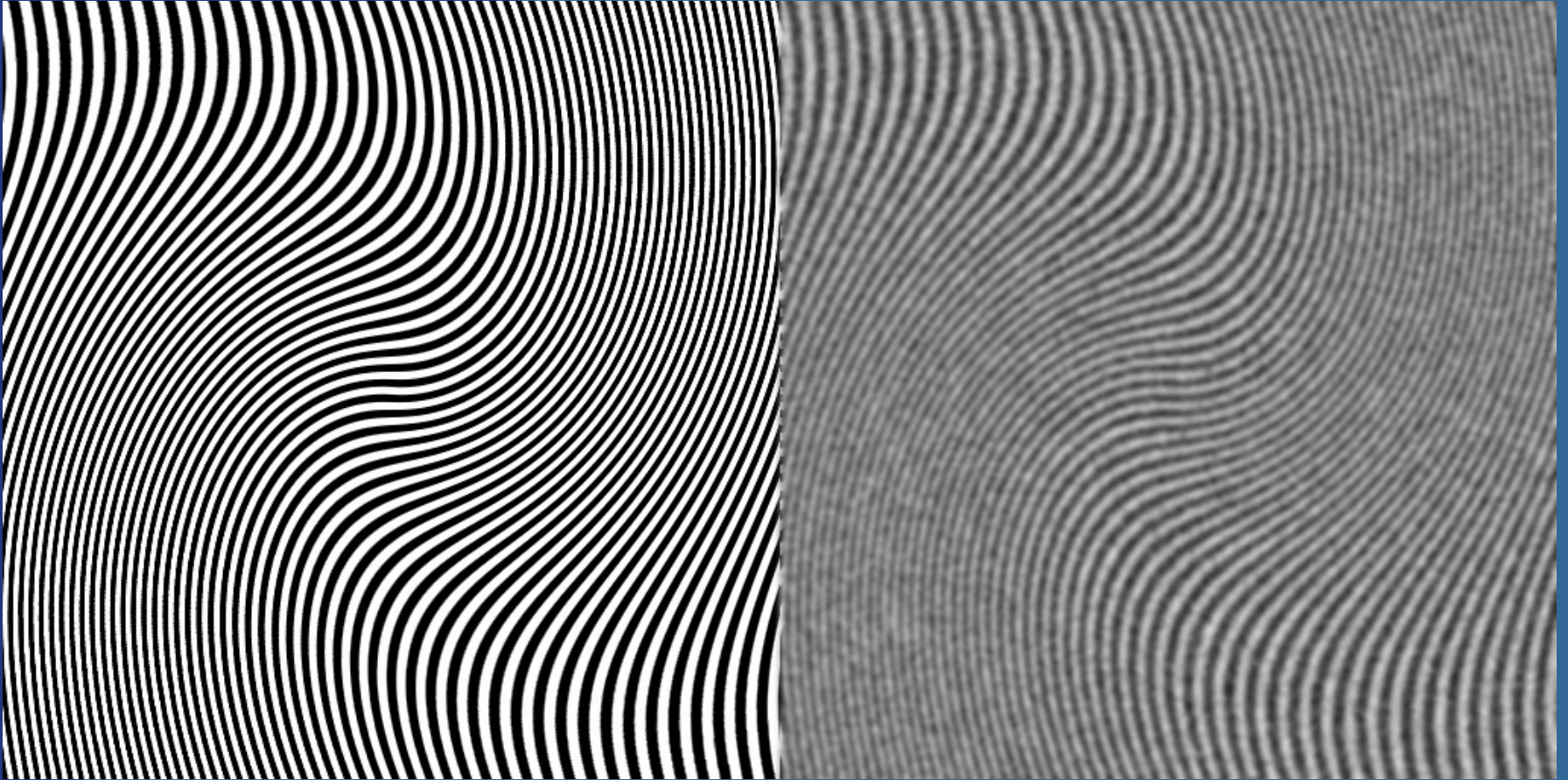


# Patterns

- Texture zoomed out until square  $< 1$  pixel



# Moire example



Moire patterns

Noise + gaussian blur  
(no moire patterns)

# Why

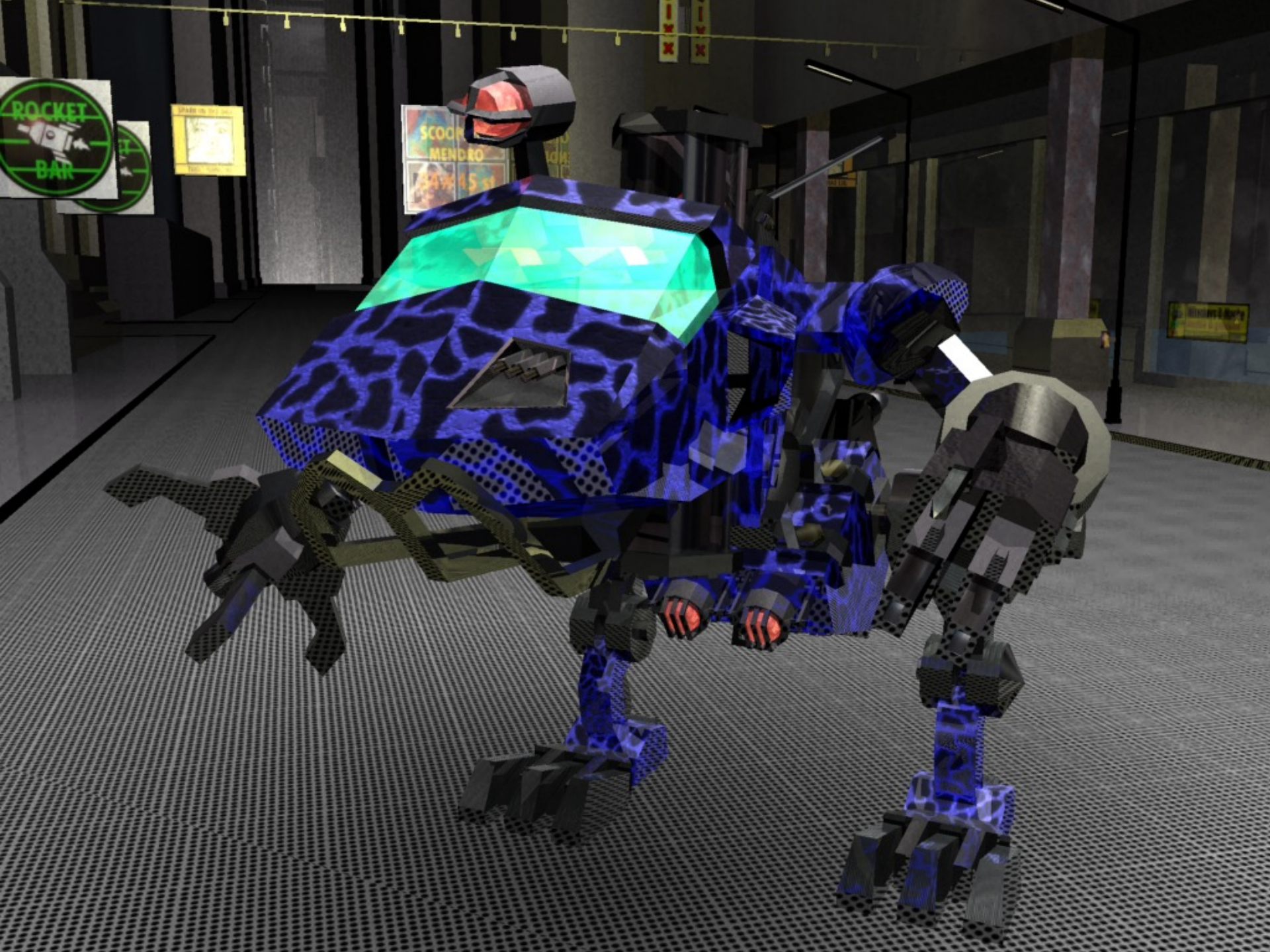
Moiré Effekt



“Moiré effects occur whenever tiny image structures (like the pattern on a shirt) can not be resolved sufficiently by the resolution of the image sensor. According to the Nyquist theorem, each period of an image structure must be covered with at least two pixels. When this is not the case, Moiré effects are the consequence. To avoid Moiré Effects the manufacturers of CCD camera systems use a filter that diffuses the light hitting the sensor area in such a way that it corresponds to the resolution of the ccd. “

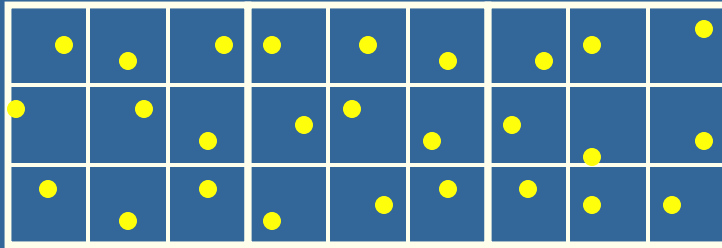








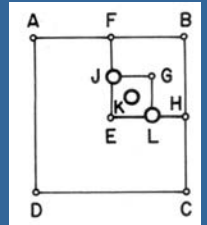
# Jittered sampling



- Works as before
  - Replaces aliasing with noise
  - Our visual system likes that better
- This is often a preferred solution
- Can use adaptive strategies as well

# Typical Exam Questions

- Describe the basic ray tracing algorithm (see next slide)
- Compute the reflection + refraction vector
  - You do not need to use Heckbert's method
- Describe an adaptive super sampling scheme
  - Including recursively computing weights
- What is jittering?

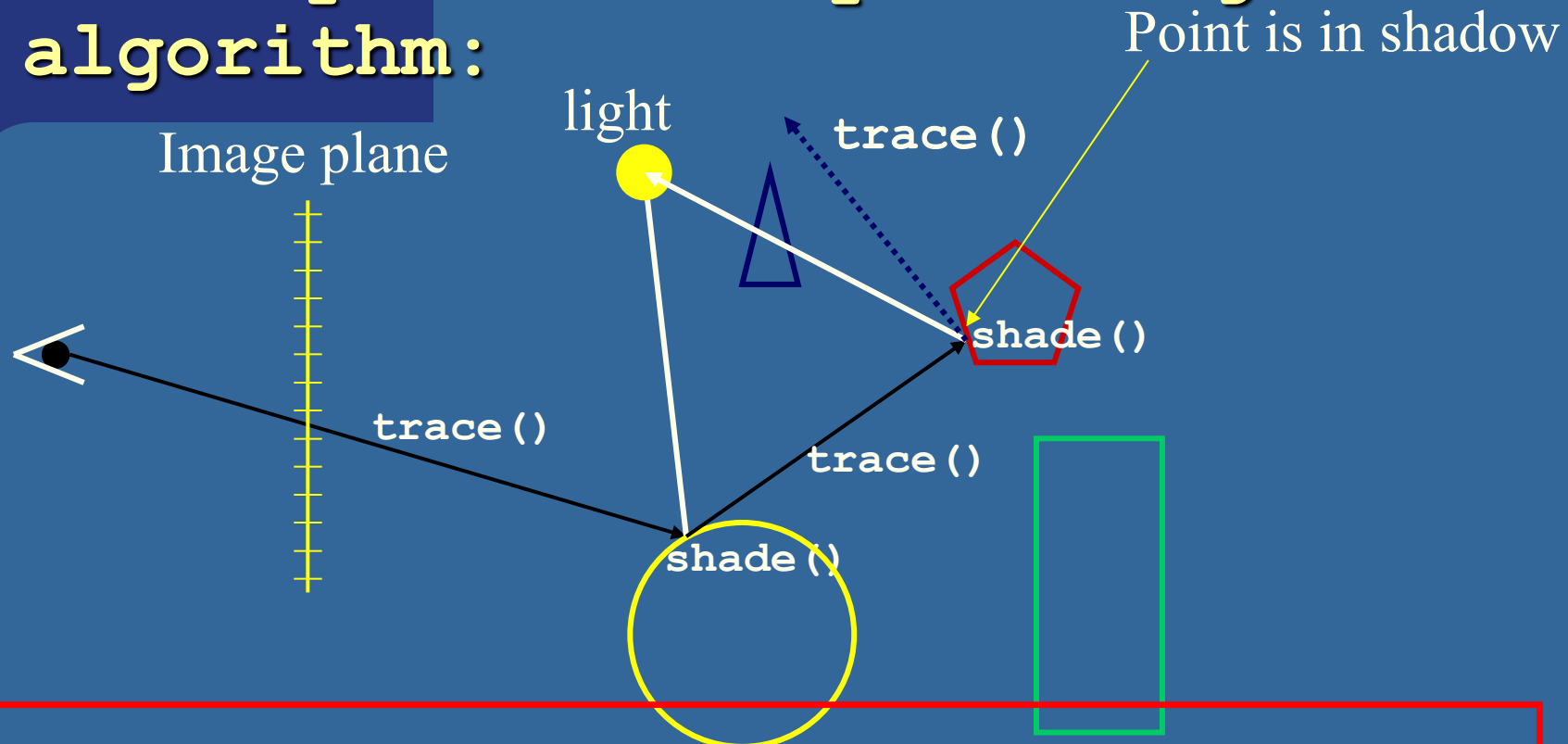


Pseudo code:

```
Color AdaptiveSuperSampling() {
 – Make sure all 5 samples exist
 • (Shoot new rays along diagonal if necessary)

 – Color col = black;
 – For each quad i
 • If the colors of the 2 samples are fairly similar
 – $col += (1/4) * (\text{average of the two colors})$
 • Else
 – $col += (1/4) * \text{adaptiveSuperSampling}(\text{quad}[i])$
 – return col;
}
```

# Summary of the Ray tracing- algorithm:



- **main()-calls trace()** for each pixel
- **trace():** should return color of closest hit point along ray.
  1. calls `findClosestIntersection()`
  2. If any object intersected → call `shade()`.
- **Shade():** should compute color at hit point
  1. For each light source, shoot shadow ray to determine if light source is visible  
If not in shadow, compute diffuse + specular contribution.
  2. Compute ambient contribution
  3. Call `trace()` recursively for the reflection- and refraction ray.

# Real-Time Ray Tracing

- Hardware:
  - SSE, GPU,
  - NVIDIA RTX - GPU accelerated:
    - AABB-hierarchy construction and
    - ray vs AABBH intersection tests.
- Low level optimizations
  - Precomputation of constants per frame, e.q., ray-AABB test.
- Rasterize primary rays.
- Adaptive sub sampling
- Frameless rendering (motion blur)
- Temporal Reprojection

# Ray-AABB hierarchy test, optimized

$$t = (-(\mathbf{n} \cdot \mathbf{o}) - d) / (\mathbf{n} \cdot \mathbf{d})$$

Compute constants per ray and slab axis (x,y, or z). With  $\mathbf{o}$ ,  $\mathbf{d}$  and  $\mathbf{n}$  constant, we can precompute:

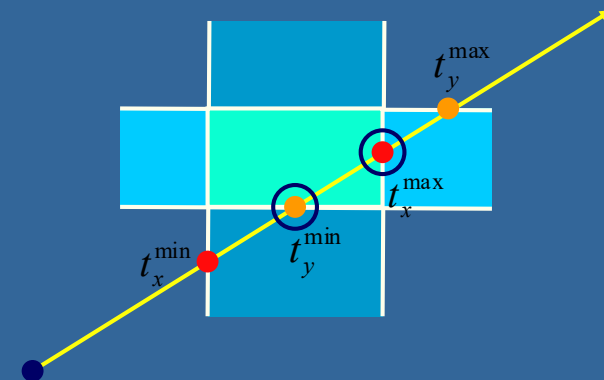
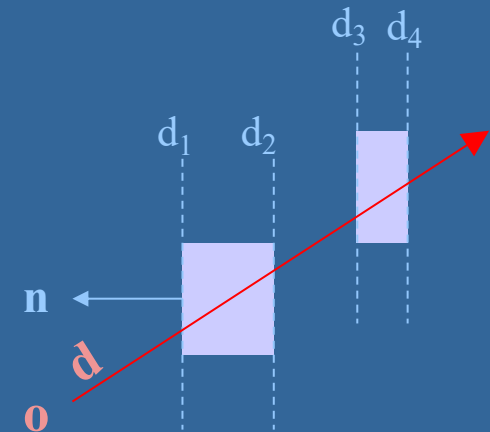
$$\text{set } a = -(\mathbf{n} \cdot \mathbf{o}) / (\mathbf{n} \cdot \mathbf{d})$$

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

$$\Rightarrow t = a + \mathbf{d} \cdot \mathbf{b} \quad // \text{ just 1 madd instr. per plane}$$

Per AABB:

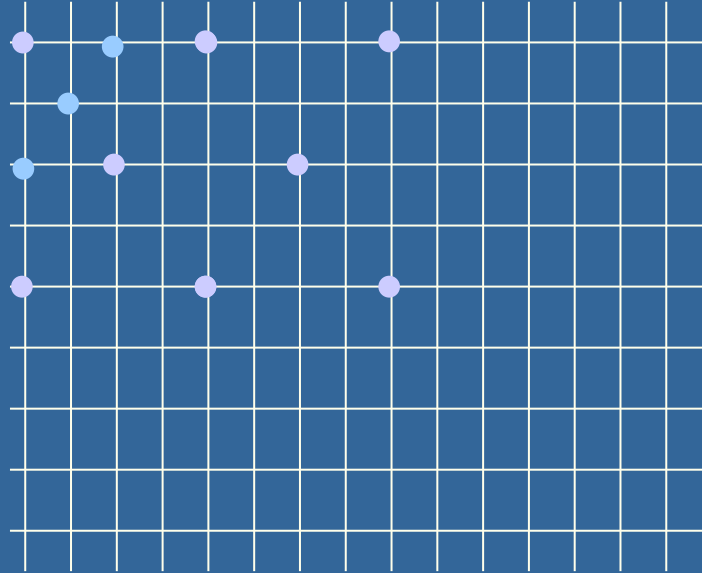
- 6 ray/plane computations à 1 *madd*.
  - ~5 *max* instr.
- + a few comparison instructions.  
VERY FAST



- Keep max of  $t^{\min:s}$  and min of  $t^{\max:s}$
- If  $t^{\min} < t^{\max}$  then intersection
- Special case when ray parallel to slab



# Adaptive Sub Sampling



Many versions exist. E.g., quincunx again:

- Start by sampling every 4x4 pixel corners and in the middle. Gives on average 2 samples per 16 pixels.
- If a quadrant's 2 samples are fairly similar,
  - fill in pixel colors by interpolation.
- Else, supersample recursively.

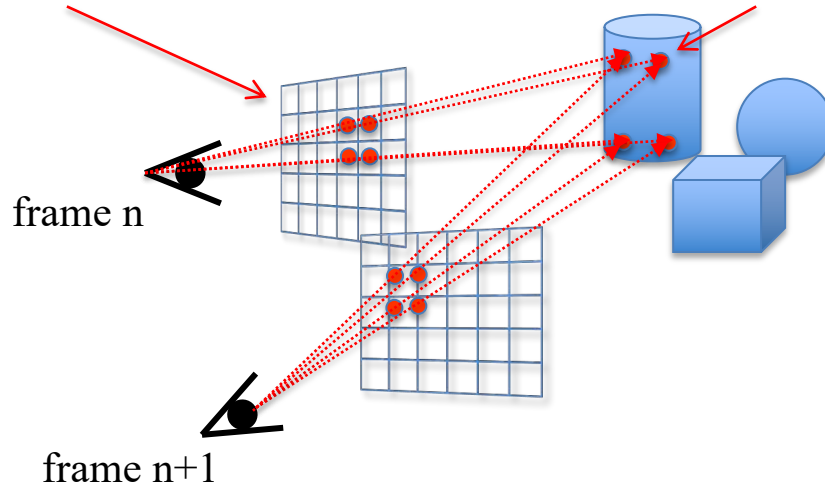
# Frameless rendering



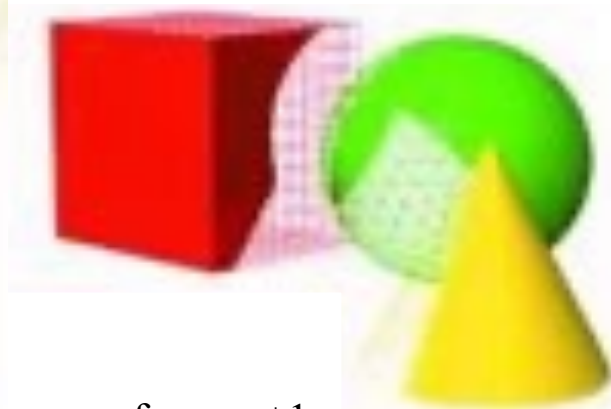
52 Frameless Rendering – updating e.g. only 10% of all pixels each frame

# Temporal Reprojection

Store (r,g,b) color and world space (x,y,z) per pixel



frame n



frame n+1

Reproject samples from frame  $n$  to frame  $n+1$ . Then:

- For pixel with  $<1$  sample
  - trace new ray
- For pixel with  $\geq 1$  sample
  - use closest (smallest  $z$ )
- Does not work as well for spec. mtrl.