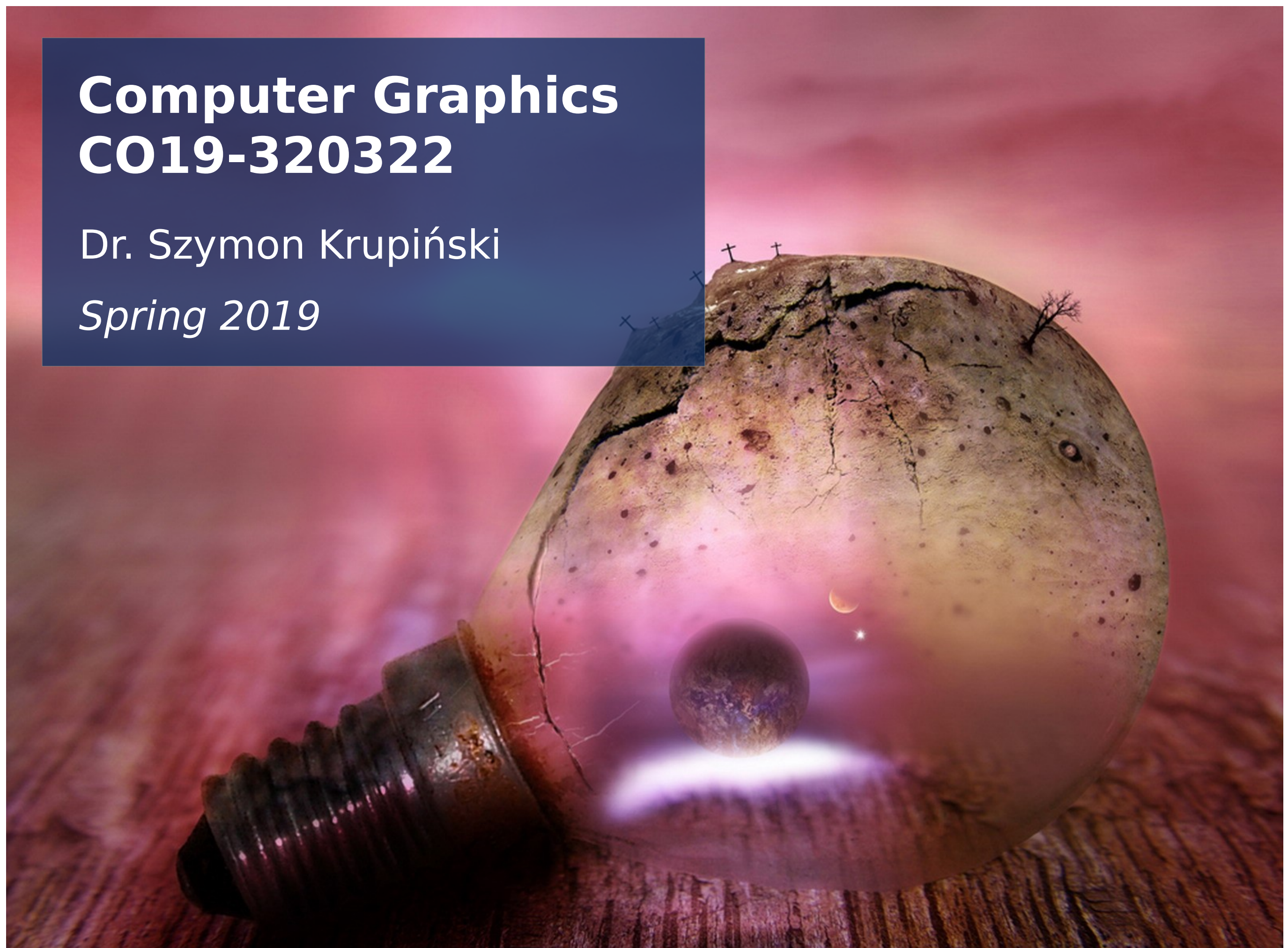
Lecture 10: Rasterisation 4

JACOBS UNIVERSITY



Computer Graphics CO19-320322

Dr. Szymon Krupiński

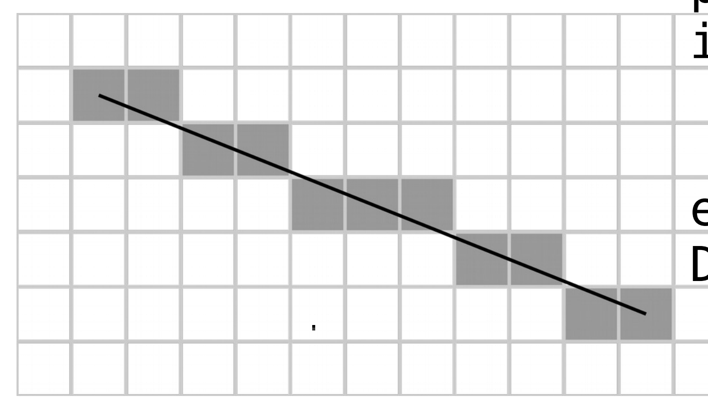
Spring 2019

Other approaches

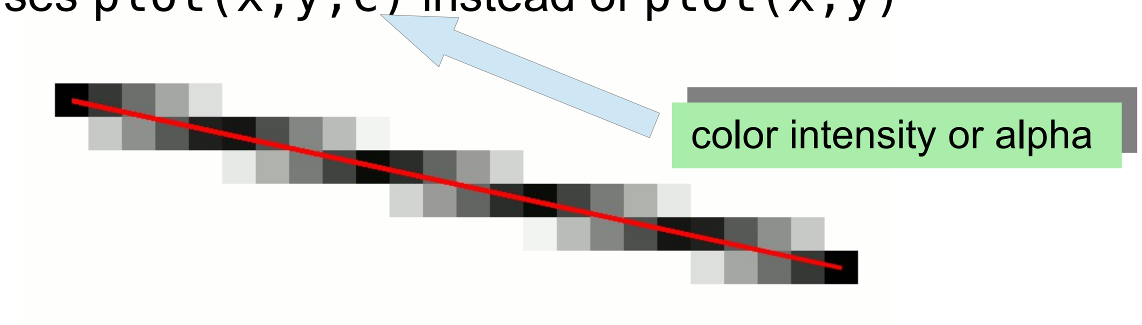
- Simple line tracing (edges)
  - Bresenham's line algorithm
    - uses only integer addition, subtraction and bit shifting
    - Still implemented in some hardware!
  - Xiaolin Wu's line algorithm
    - anti-aliased lines

Bresenham's line alg.:

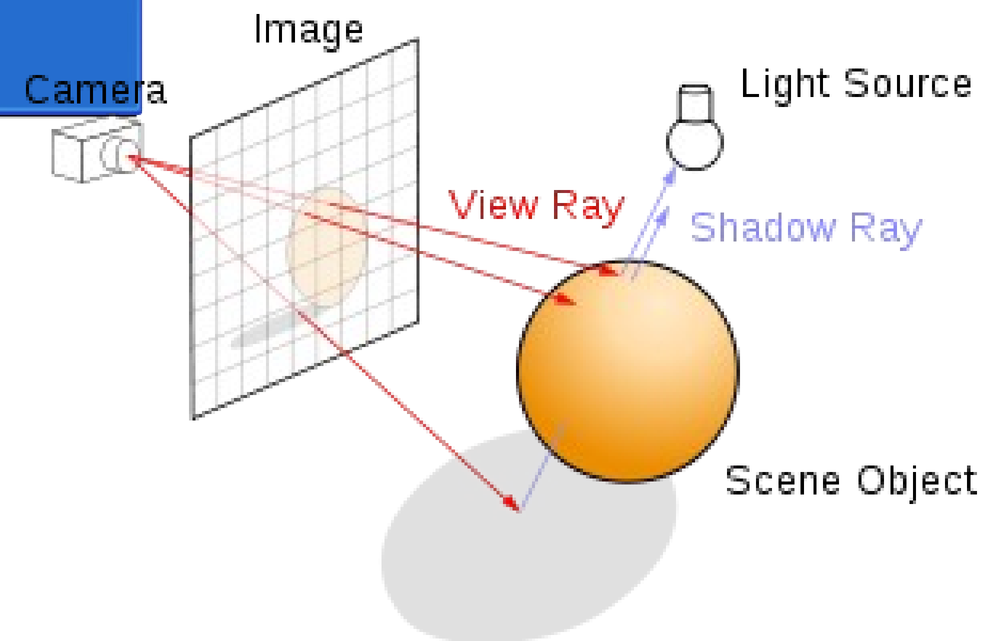
```
plotLine(x0,y0, x1,y1)
  dx = x1 - x0
  dy = y1 - y0
  D = 2*dy - dx
  y = y0

  for x from x0 to x1
    plot(x,y)
    if D > 0
      y = y + 1
      D = D - 2*dx
    end if
    D = D + 2*dy
```



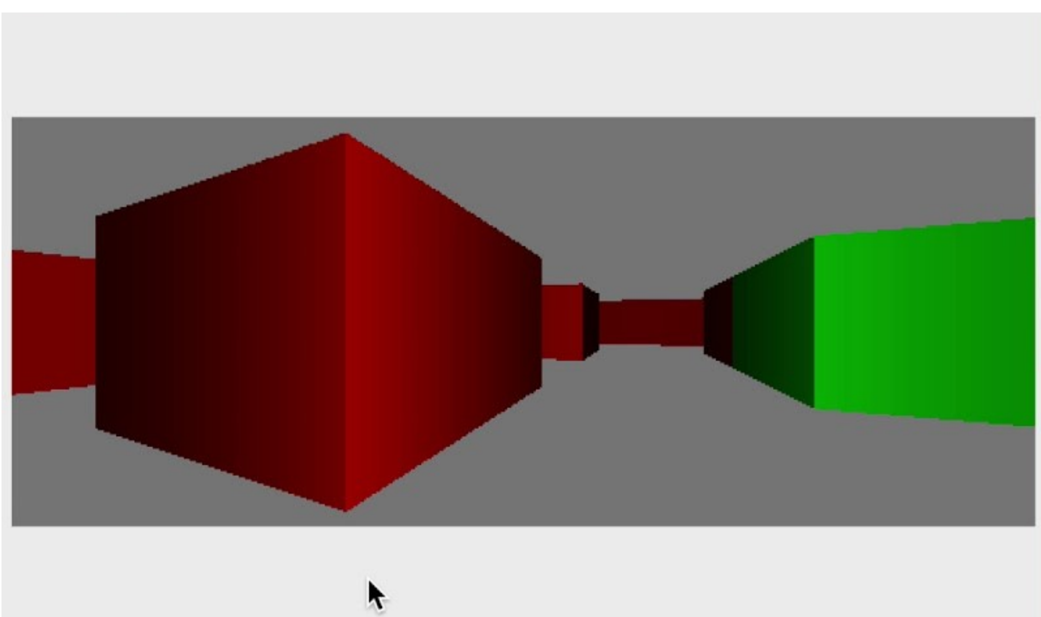
Uses `plot(x,y,c)` instead of `plot(x,y)`

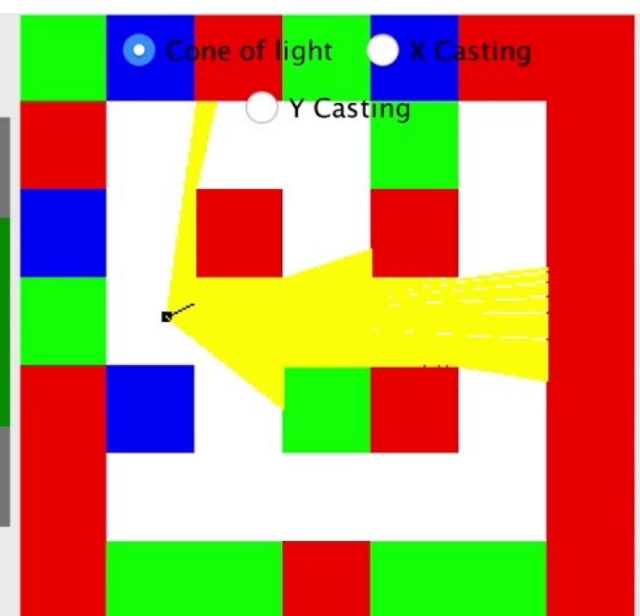


Other approaches



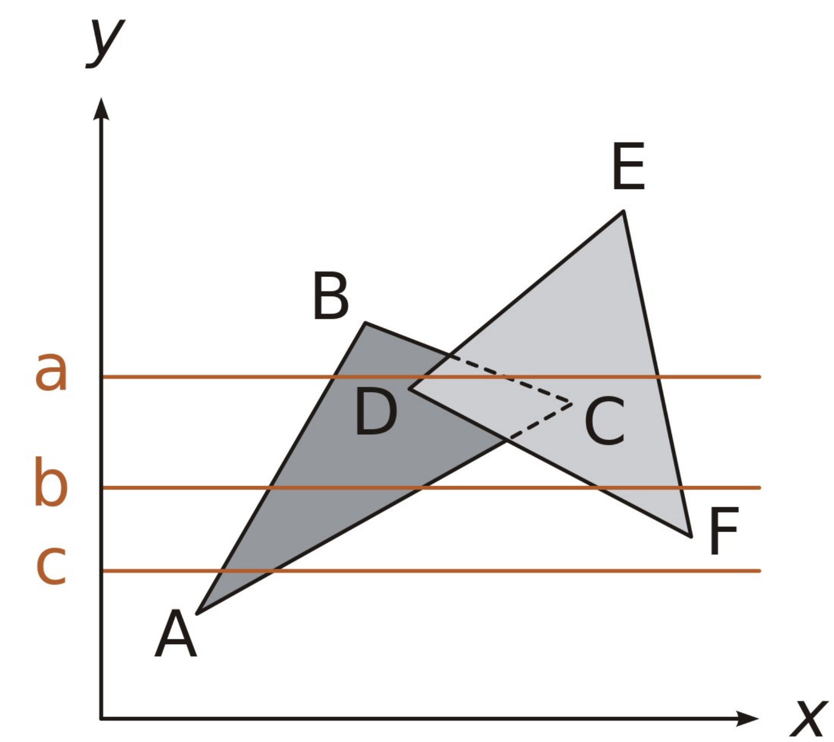
- Ray tracing
  - Big separate topic
- Ray casting
  - Got the early 3-D games started!





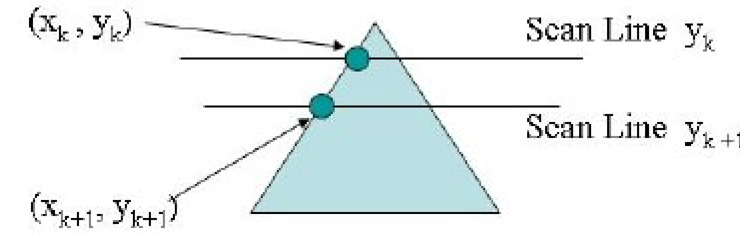
Other approaches

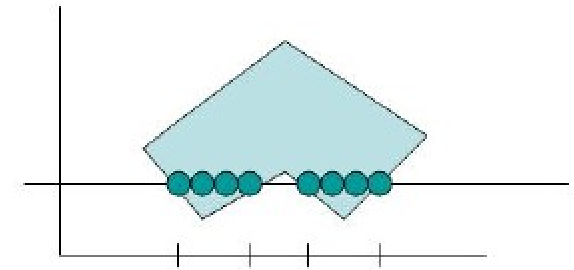
- Scanline rendering
  - The sampling method of coverage and depth checking we have discussed so far is designed to run triangle-by-triangle
  - One can also “scan” the framebuffer line by line and check which geometry should be represented by each pixel

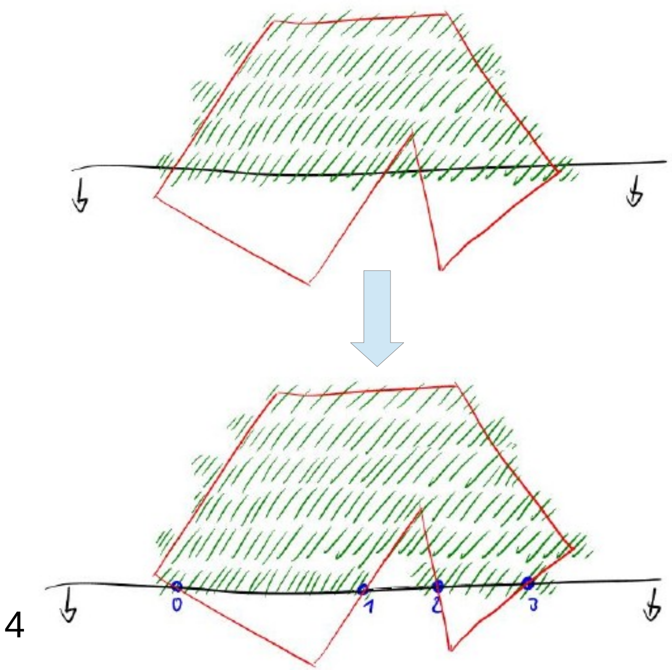


Scanline rendering

- The core of the decision process is to carry out check which edges are affecting given pixel
  - Horizontal “scan line” traverses the scene top-down
  - Intersection of the scan line are checked with all triangle / polygon edges and sorted left to right
  - Simple logic: start filling in the color after encountering the first edge intersection and stop after the second
  - The renderer maintains a list of all active edges in a linked list in order to speed up the process
  - This list is modified by inserting edges at “event points”
    - Start / end of an edge
    - Edge crossing (need to change the order)
  - Horizontal edges can be eliminated







Scanline rendering

- Many different optimisations are ossible with this approach
- Number of times visible pixels are processed is kept to the absolute minimum
- But, in case of a big set of geometries, the active edge list can grow very big
- It’s difficult to tune performance
- Currently the method is little used in real time CG today

Ray tracing

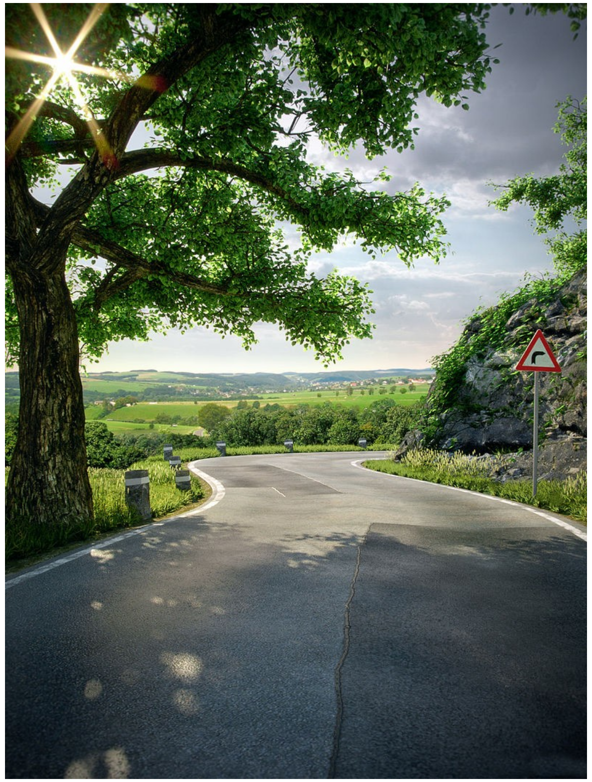
- An intuitive concept: simulate a ray of light
  - Either coming from the light source (as physics dictates)
  - Or leaving the camera (reverse – we only see rays which make it to our eyes)
- Ray tracing is a per-sample-point operation
  - By checking what geometry this ray has met, we determine which color and intensity the pixel should represent

Ray tracing

- Ray tracing is inherently parallel, since the rays for one pixel are independent of the rays for other pixels
- Can take advantage of modern parallel CPU/GPU/Clusters to significantly accelerate a ray tracer
  - Threading (e.g., Pthread, OpenMP) distributes rays among cores
  - Message Passing Interface (MPI) distributes rays among processors across different machines
  - OptiX/CUDA distributes rays on the GPU
- Memory coherency helps when distributing rays to various threads/processors
  - Assign the spatially neighboring rays (on the image plane) to the same core/processor
  - These rays tend to intersect with the same objects in the scene, and thus need access to the same memory

Ray tracing

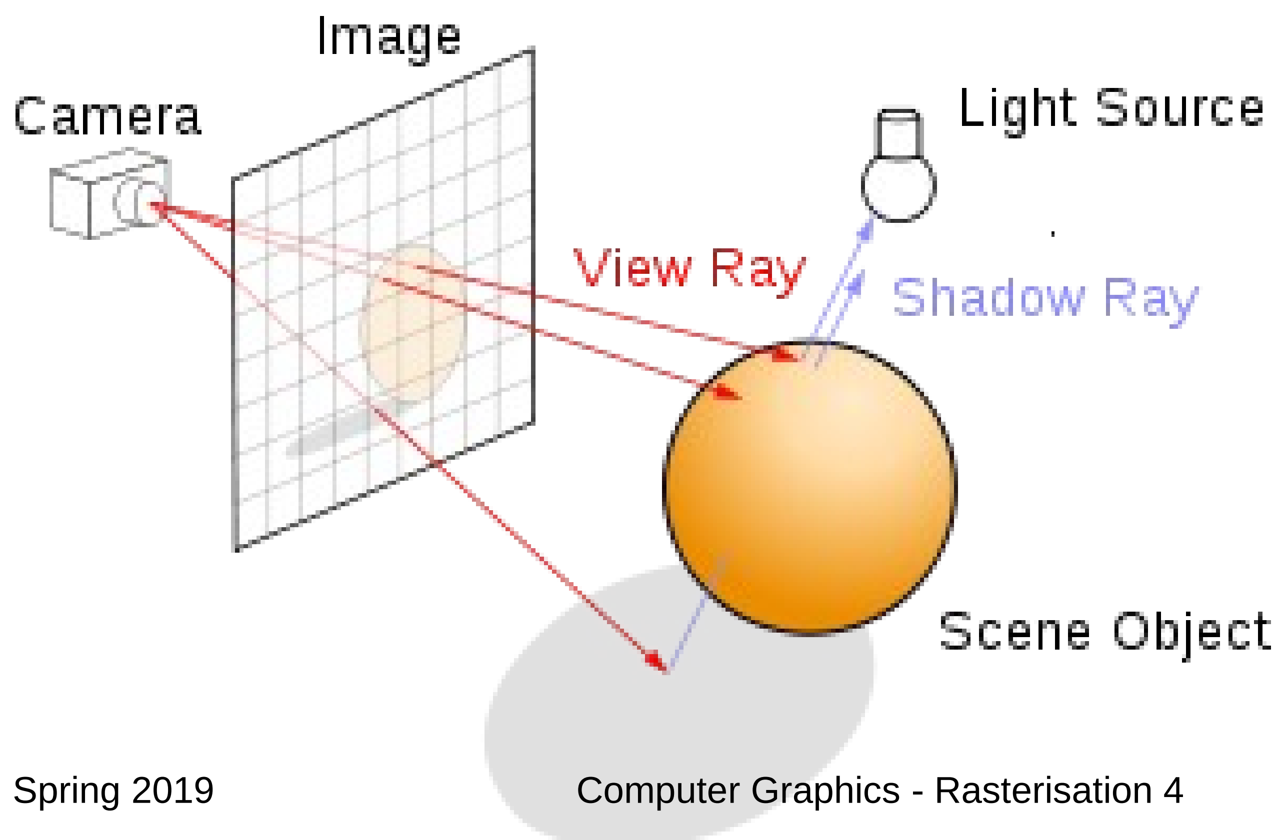
- Typically not used in games
  - could be used in games, but typically considered too expensive
  - However, GPUs are very good at tasks that are easily parallelizable
- For example, NVIDIA Optix real time ray tracer…
- Cycles engine of Blender
- Octane renderer using Brigade 3D technology



Good demos:
https://www.youtube.com/watch?v=tjf-1BxpR9c
https://www.youtube.com/watch?v=BpT6MkCeP7Y

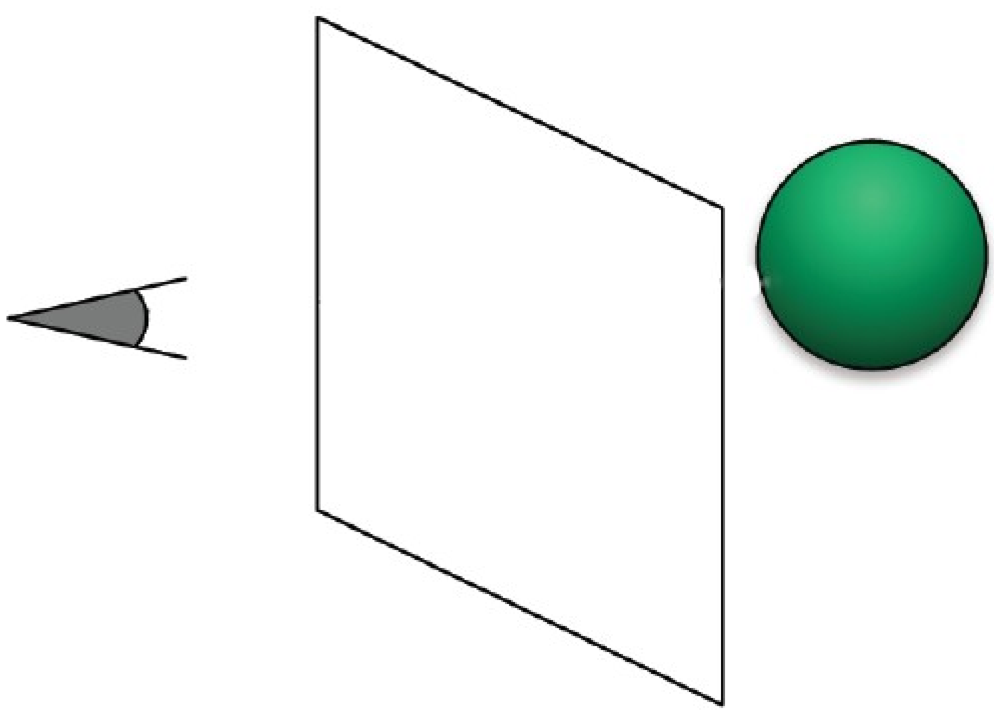
Ray tracing

- Generate an image by backwards tracing the path of light through pixels on the image plane
- Simulate the interaction of light with objects



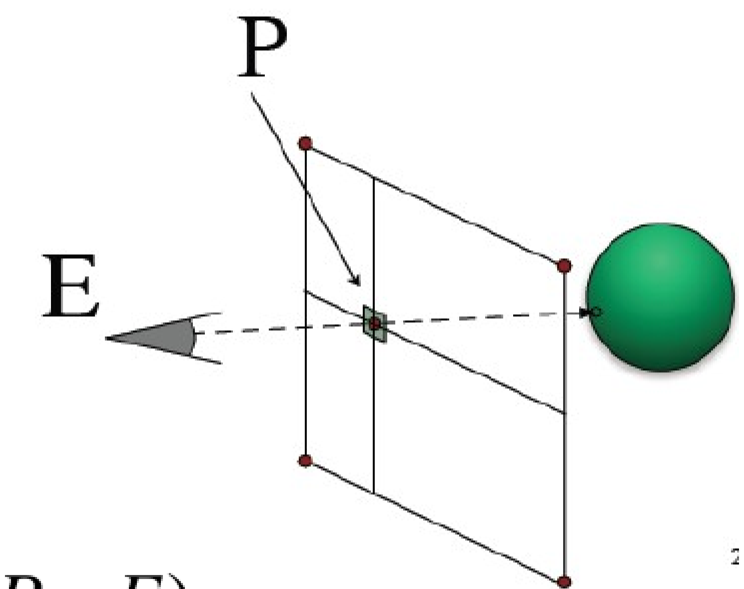
Ray tracing

- Start with an eye pupil or aperture (a focal point), along with an image plane with pixels and cast a ray *R(t)*



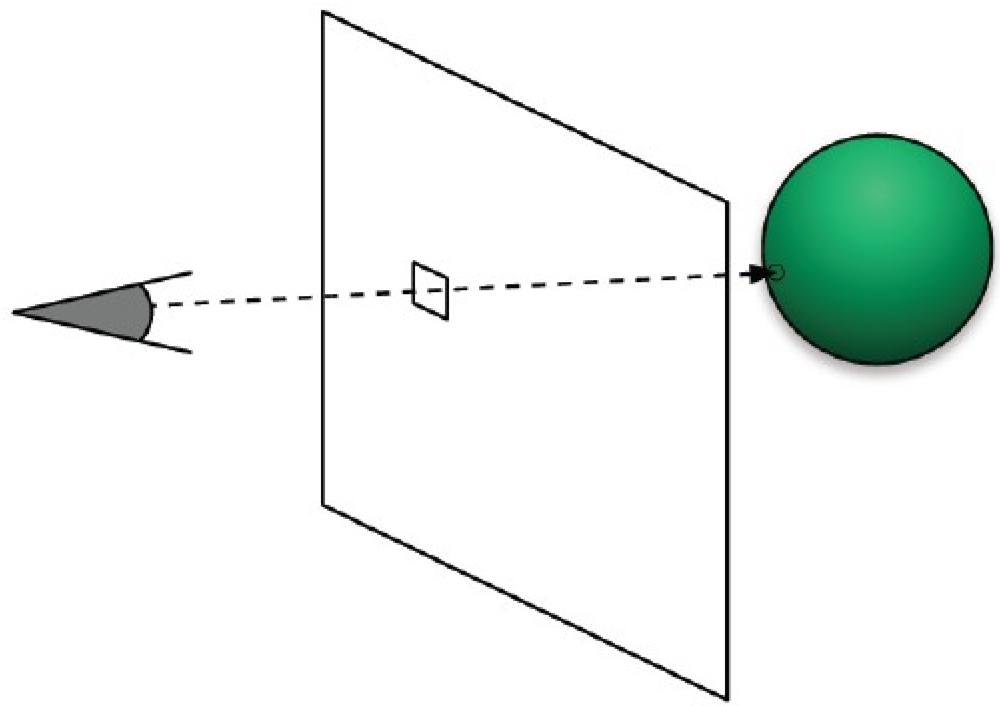
$$R(t) = E + t(P - E)$$

$$t \in [0, +\infty)$$



Ray tracing

- Interesting information: what is the first object the ray will hit?



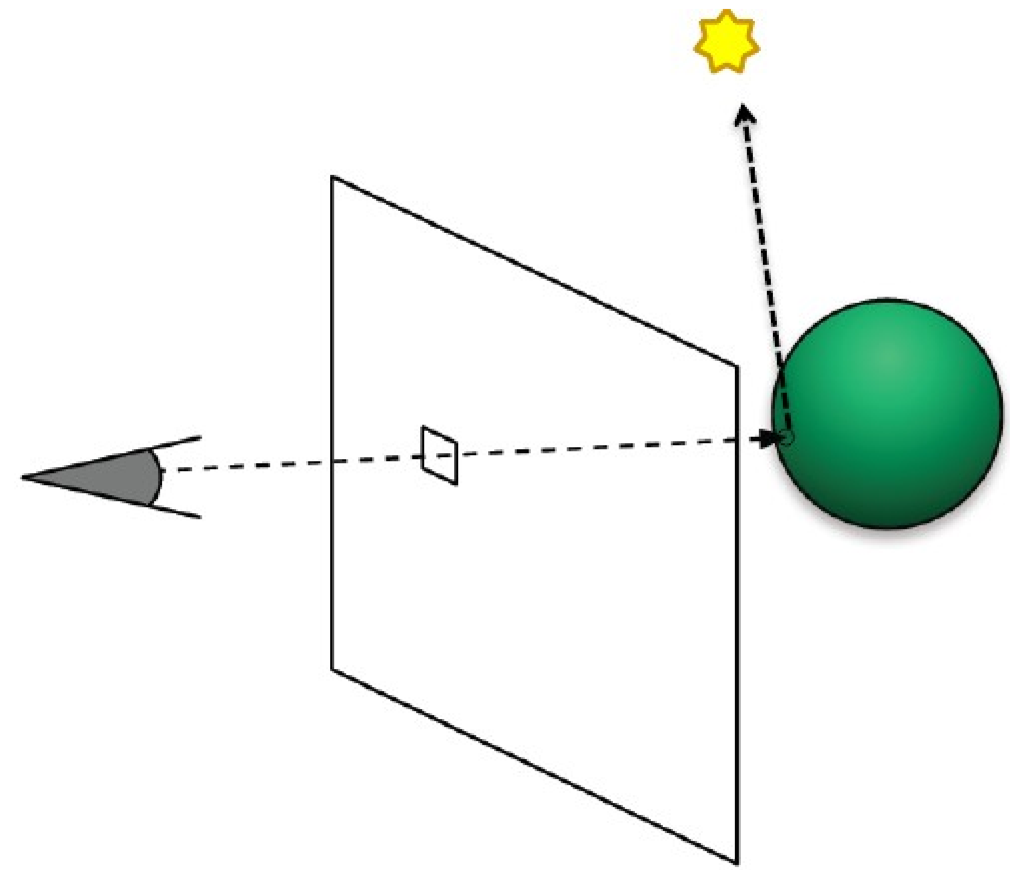
Ray tracing

- Another information we need: what intensity to give to its color? Need to check for lighting conditions
  - (we haven’t covered this for our triangl rendering yet)
  - we’ll cast a “shadow ray” towards the source of light to see if we have direct view of it (if not = we are in shadow)
  - Actually, towards ALL sources of light...
  - This works in a very similar way as OpenGL shading but it’s not the same

$$R(t) = S + t(L - S)$$

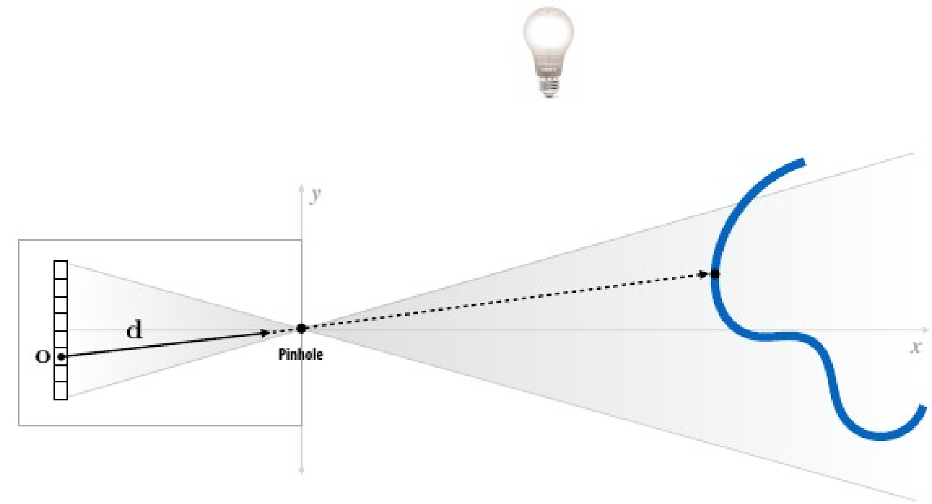
$$t \in [\varepsilon, 1)$$

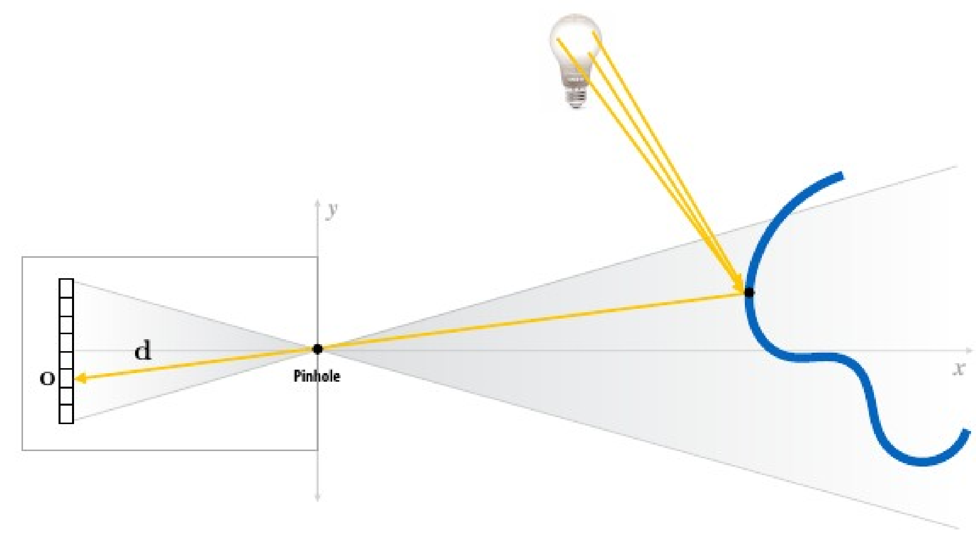
S – light ray interception point vector
L – light source vector



Ray tracing

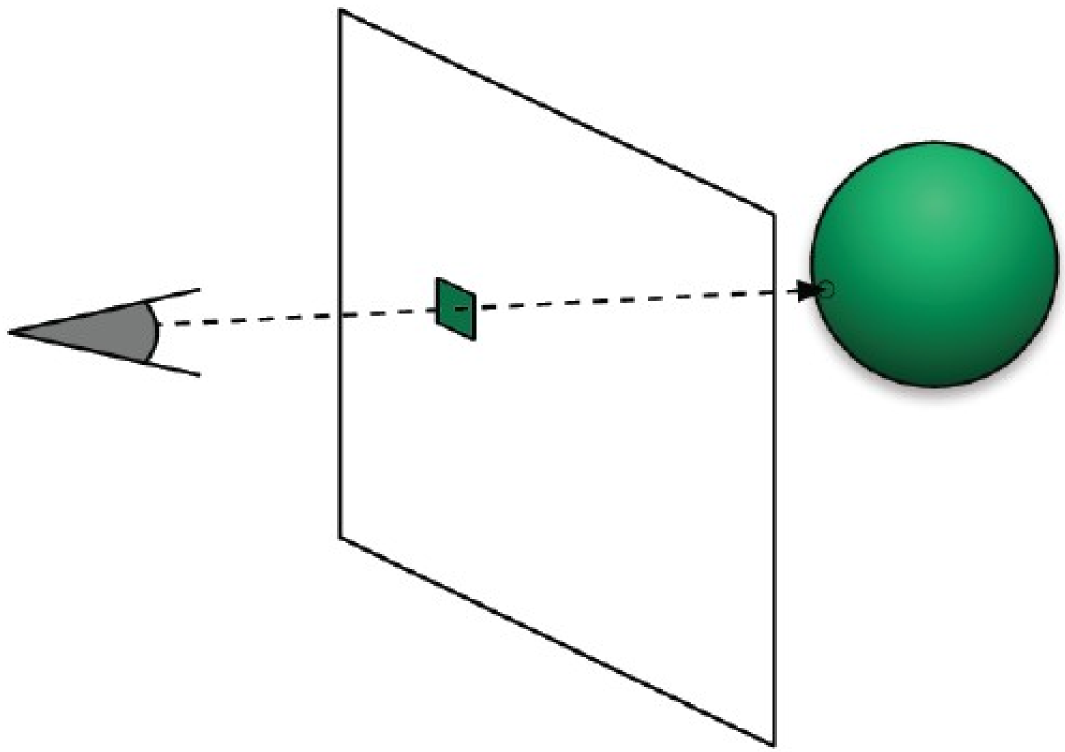
- The concept is simple but the reality is pretty complex!
- We will need to have a fairly comprehensive model to take care of even just the reflection





Ray tracing

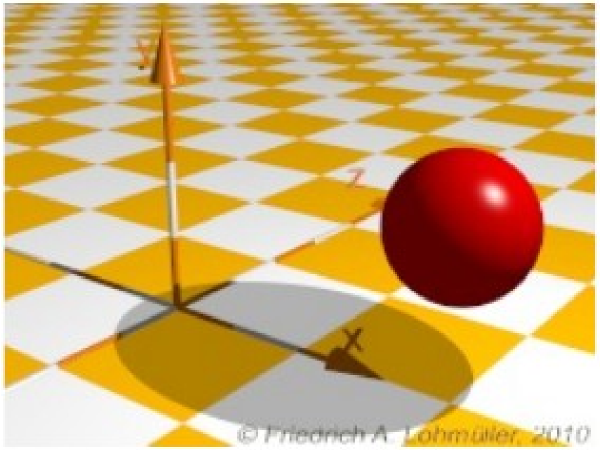
- Finally, we can color the pixel!



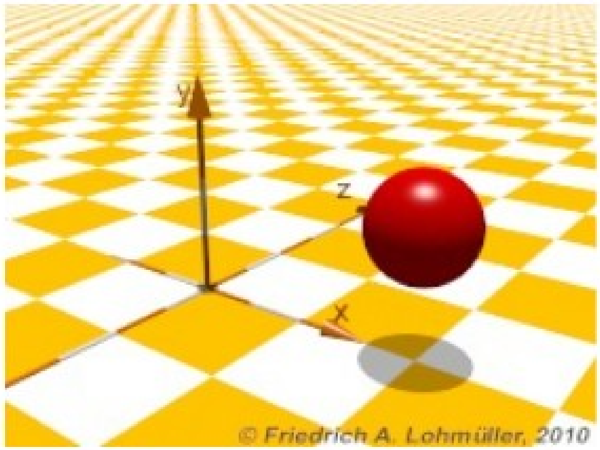
Here is how a ray tracing algorithm could look like:

```
Image Raytrace(Eye eye, Scene scene, int width,
int height) {
    Image image = new Image(width, height);
    for (int i = 0 ; i < height ; i++)
        for (int j = 0; j < width; j++) {
            Ray ray = RayThruPixel(eye, i, j);
            Intersection hit = Intersect (ray,
scene);
            image[i][j] = FindColor(hit);
        }
    return image;
}
```

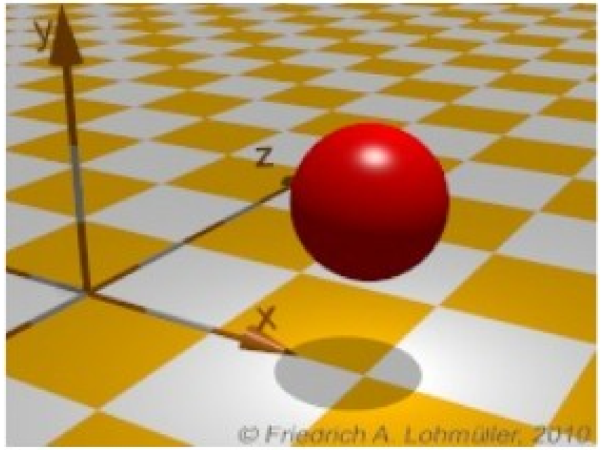
Light sources



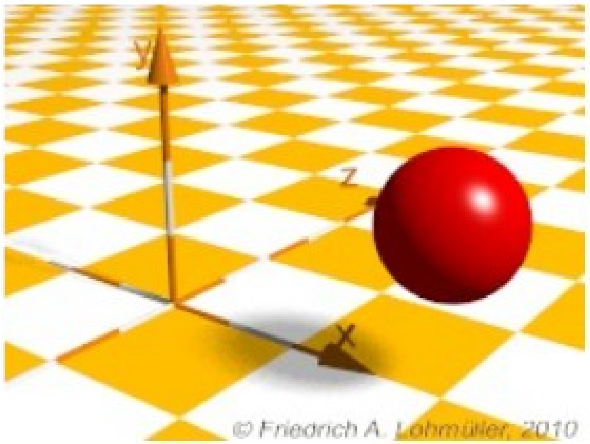
Point Light



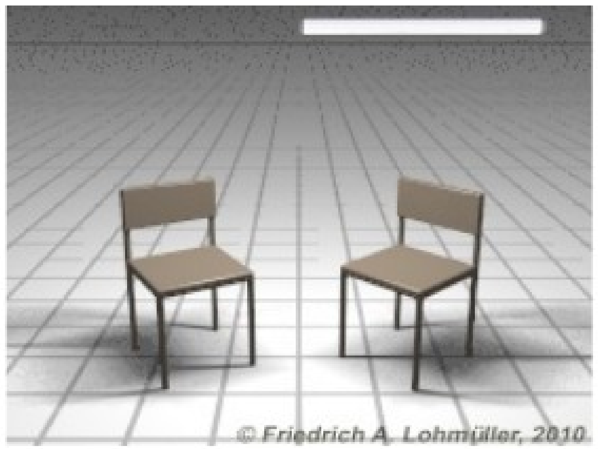
Directional Light



Spot Light



Area Light

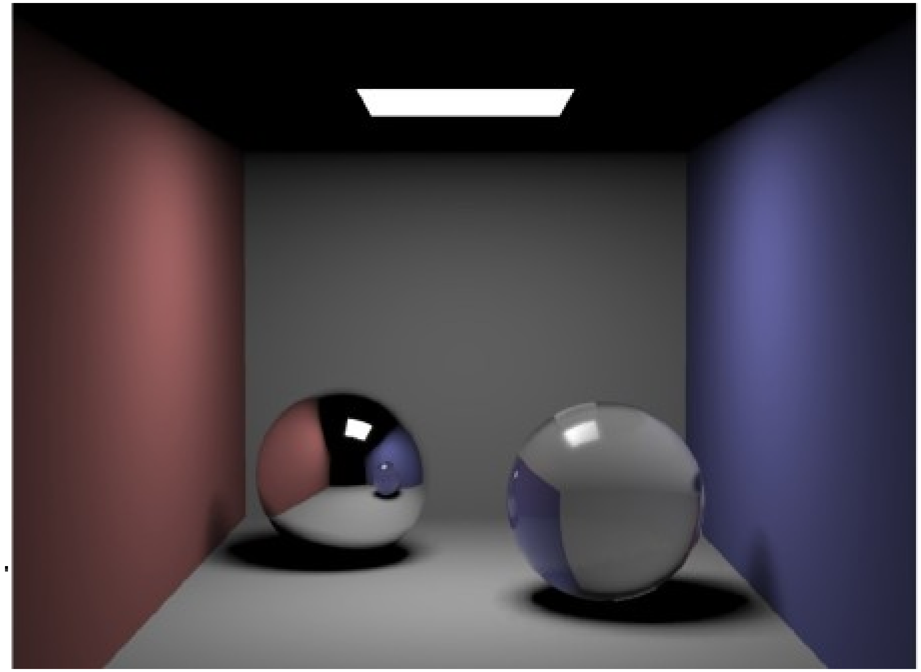


Area Light from a light tube

Volume light

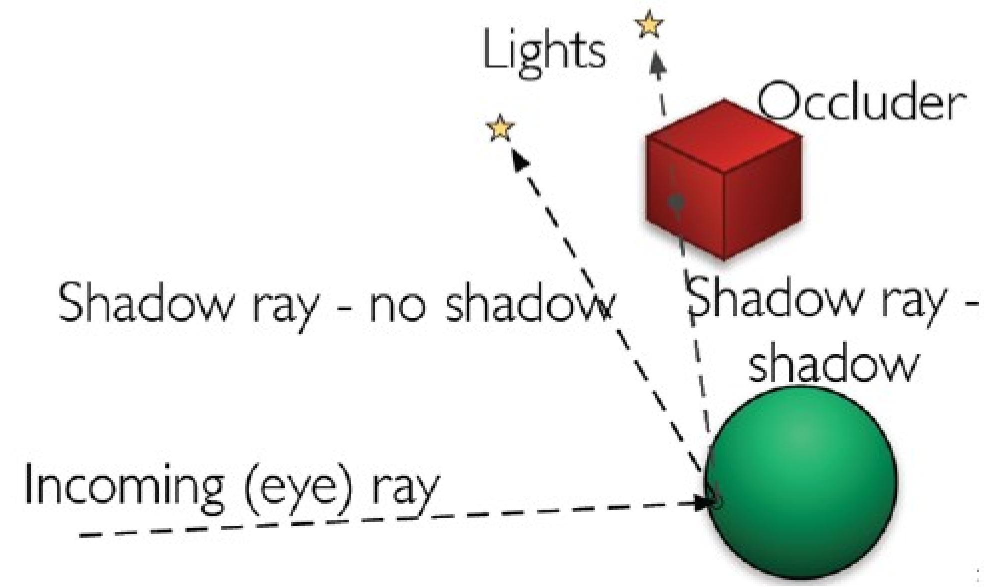
Area lights

- Treat the area light like a bunch of point lights
- Shoot a number of rays from the intersection point to different points on the area light
- Take the average of the results
- Creates soft shadows effect



Shadows

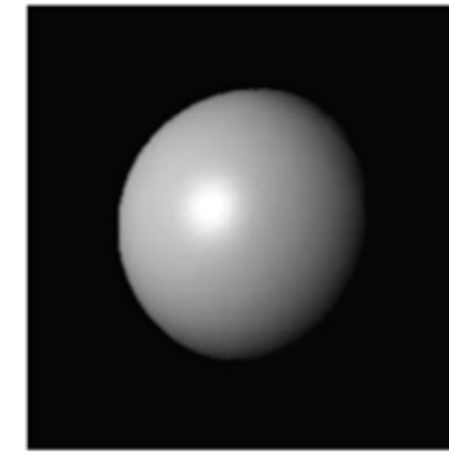
- Casting the shadow ray can produce two results:
  - Free path to the light source
  - Occluding object on the way
- We have to test for an occlusion between the source of light and the origin of the ray
  - If something is present, we could leave the pixel black = deep shadow
  - When did you see completely black shadow last time? → there is always ambient light…
  -

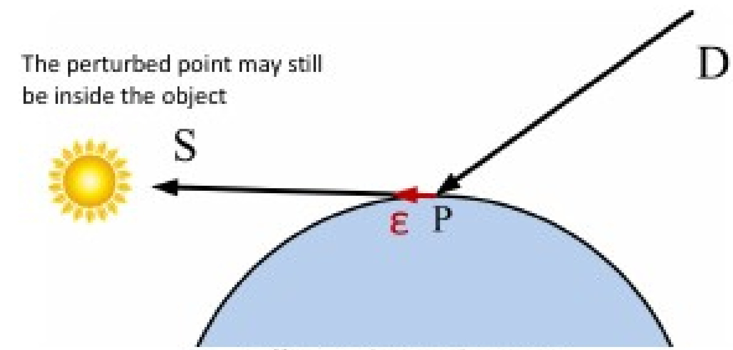


Shadows

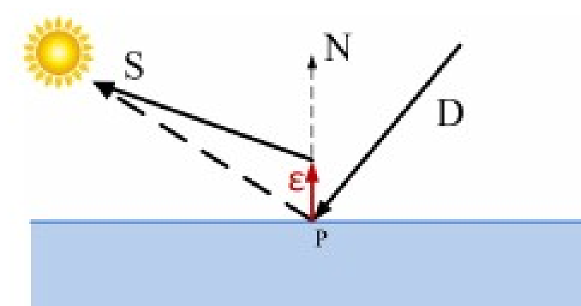
- Inaccuracies can cause trouble!
- Add ε to the starting point of shadow rays to avoid accidental re-intersection with the original surface: $t \in [\varepsilon, 1)$
  - This can often fail for grazing shadow rays near the objects silhouette
- Better to offset the intersection point in the normal direction from the surface
  - The direction of the shadow ray shot from the perturbed point to the light may be slightly different from the direction of the original shadow ray
- May also need to avoid placing the new starting point too close to or inside other nearby objects!


Incorrect self-shadowing


Correct



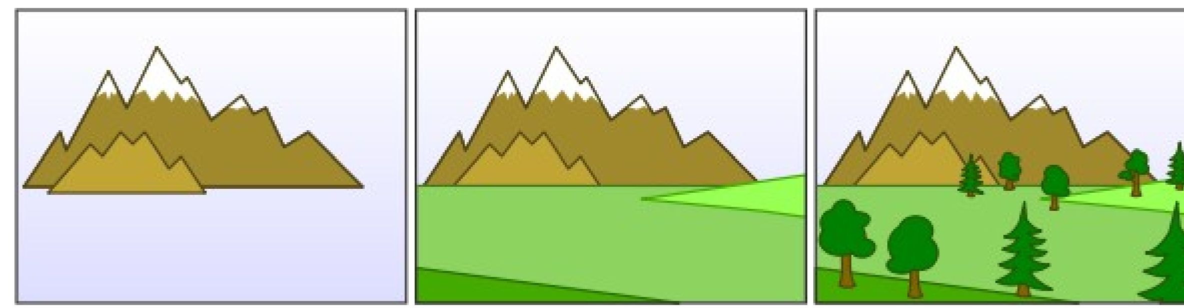
offset in the ray direction



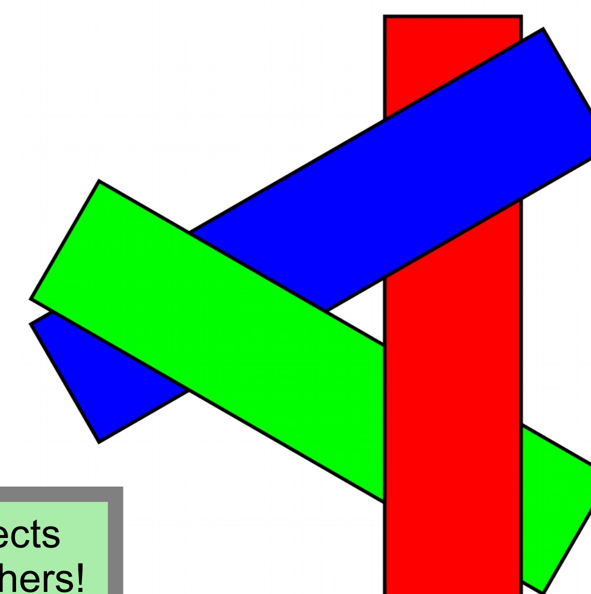
offset in the normal direction

Other approaches

- Painter's algorithm
  - Paint each object starting with the objects which have the highest depth



- Reverse Painter's algorithm
  - Paint the closest objects first but never touch the already painted parts (unless blending with transparency)
- Easy situation that can break both



Each of these objects is in front of the others!

Thank you!

- Questions?