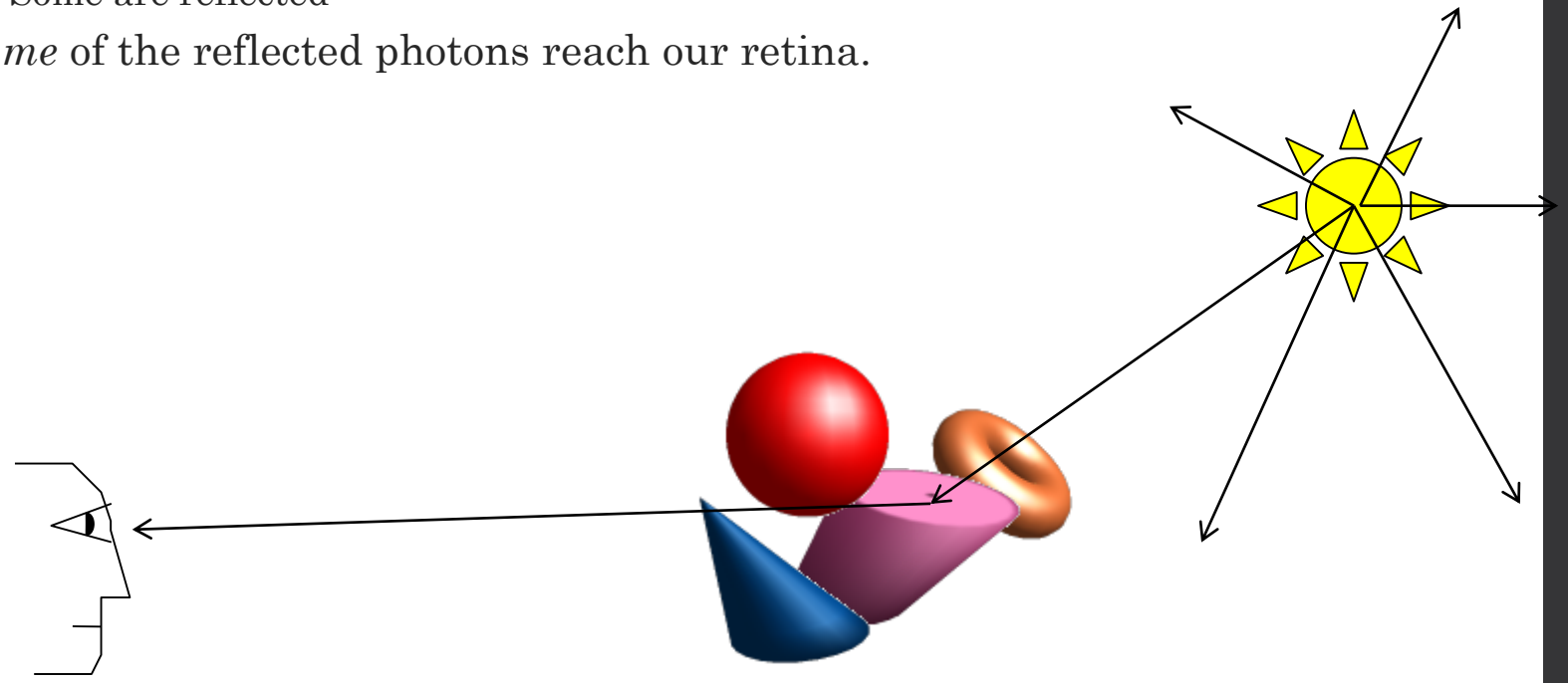
Raytracer intro

Lecture 6

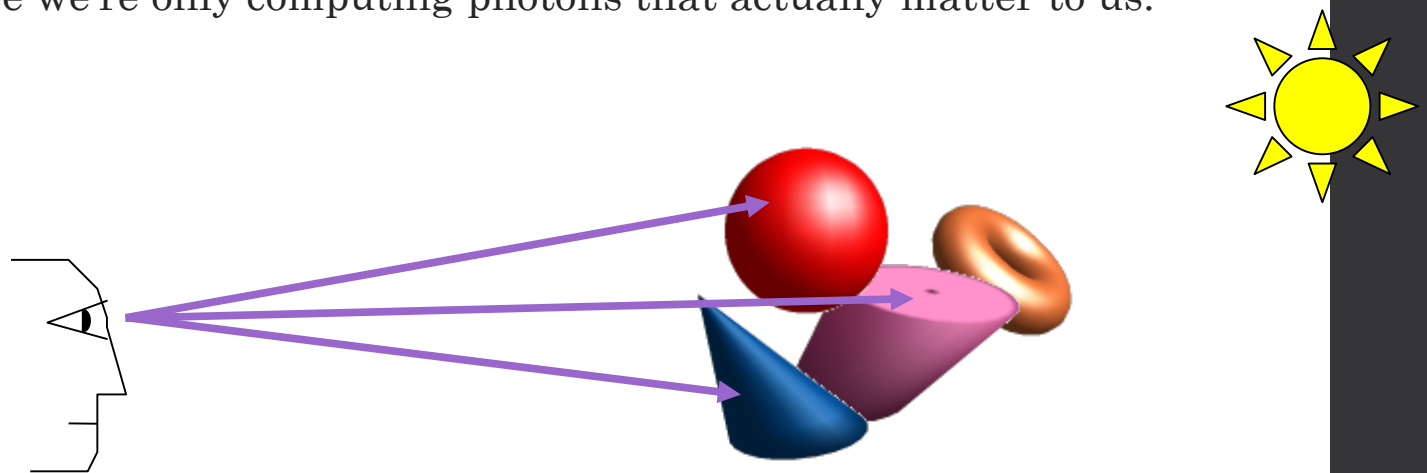
Raytracing Overview

- Loosely based on the way we perceive the world around us (visually)
- A (near) infinite number of photons are emitted by a **light source**.
  - *Some* bounce around our environment
    - Some are absorbed
    - Some are reflected
  - *Some* of the reflected photons reach our retina.



Overview, cont.

- Impractical to simulate!
  - Millions (Billions, Trillions) of "photons"
  - Most don't hit our eye.
- Observation:
  - But…if we trace photons *backwards* from the **eye** to the **light source** (by sending out a **ray**):
    - (At least) One ray per pixel
    - Definitely do-able on the computer.
  - If the ray hits something, use it to color the pixel.
  - We guarantee we're only computing photons that actually matter to us.



Overview, cont.

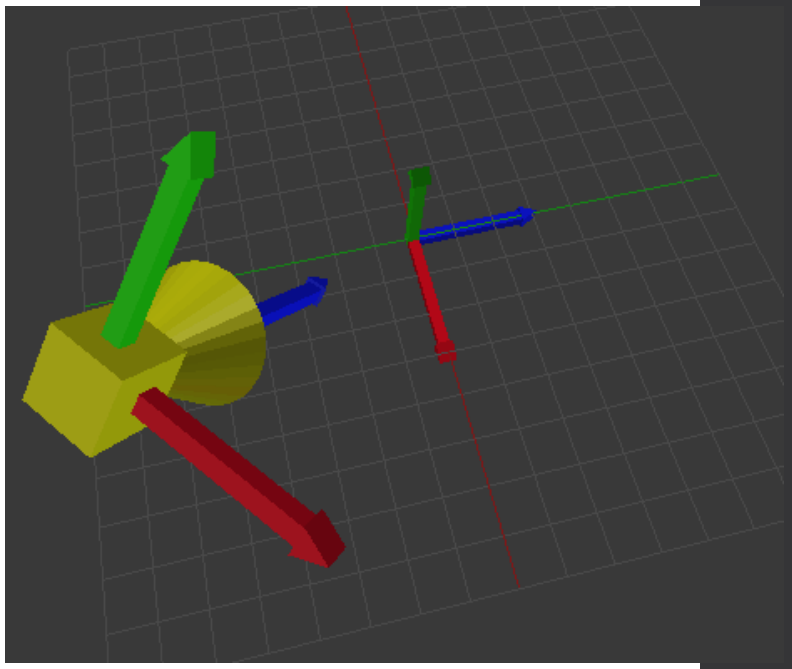
- This is the same technique used in early fps-games
- Technically, this is a **ray-casting**.

Overview, cont.

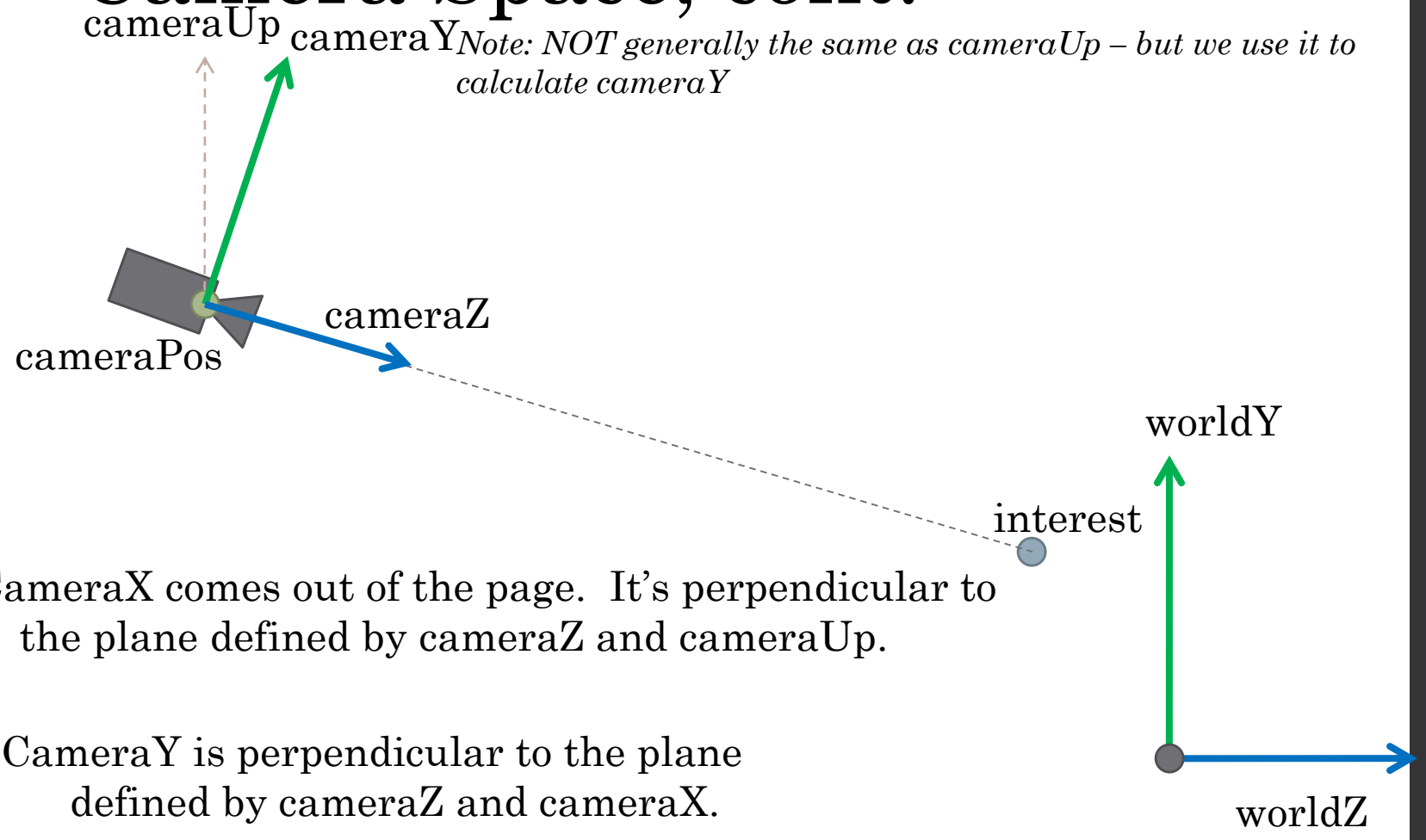
- More advanced renderings can be obtained by **recursively** bouncing rays off hit objects
  - Reflections
  - Refractions
  - Ambient Occlusion
  - Subsurface scatter
  - …

Phase 1: Define Camera Space

- We'll define camera space:
  - origin is (virtual) camera position.
  - axes are perpendicular and define a Left-handed coordinate system (since our world does)
  - Imagine yourself where the camera is (and oriented with the camera) – camera C.S. should look to you like world C.S.



Camera Space, cont.



CameraX comes out of the page. It's perpendicular to the plane defined by cameraZ and cameraUp.

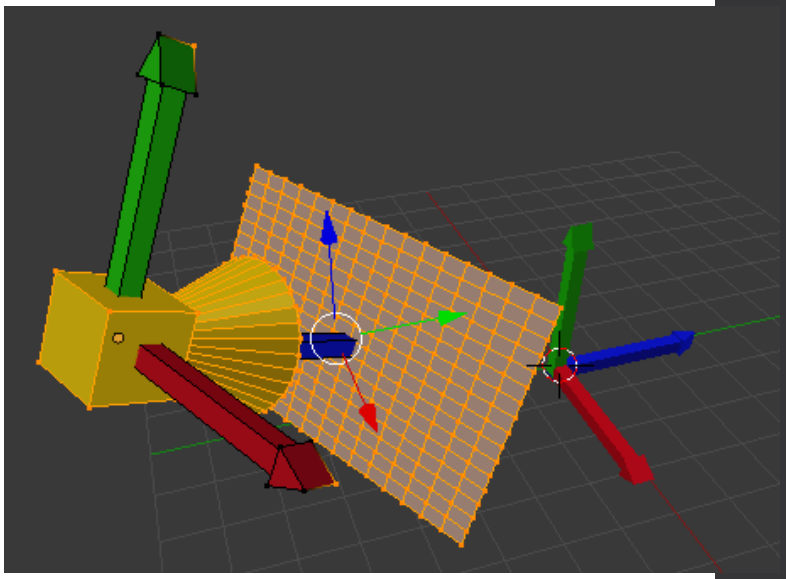
CameraY is perpendicular to the plane defined by cameraZ and cameraX.

Camera Space, cont.

- What you'll be given:
  - $\vec{\boldsymbol{C}}$: Camera position
  - $\overrightarrow{\boldsymbol{COI}}$: Position of the center of interest
  - $\overrightarrow{\boldsymbol{Cup}}$: The *general* upwards direction of the camera
- What you'll need to calculate:
  - $\overrightarrow{\boldsymbol{CamX}}$, $\overrightarrow{\boldsymbol{CamY}}$, and $\overrightarrow{\boldsymbol{CamZ}}$: the camera's local axes

Step2: Define Virtual view plane

- One key idea in R.T. is that of the **virtual view plane**.
- Imagine your pygame window (let's say 100 x 70 pixels) is sitting in front of the camera in our 3d world.
  - Centered about the camera's z axis
  - Tilted parallel to the camera x and y axes.
  - The same proportions as the pygame window

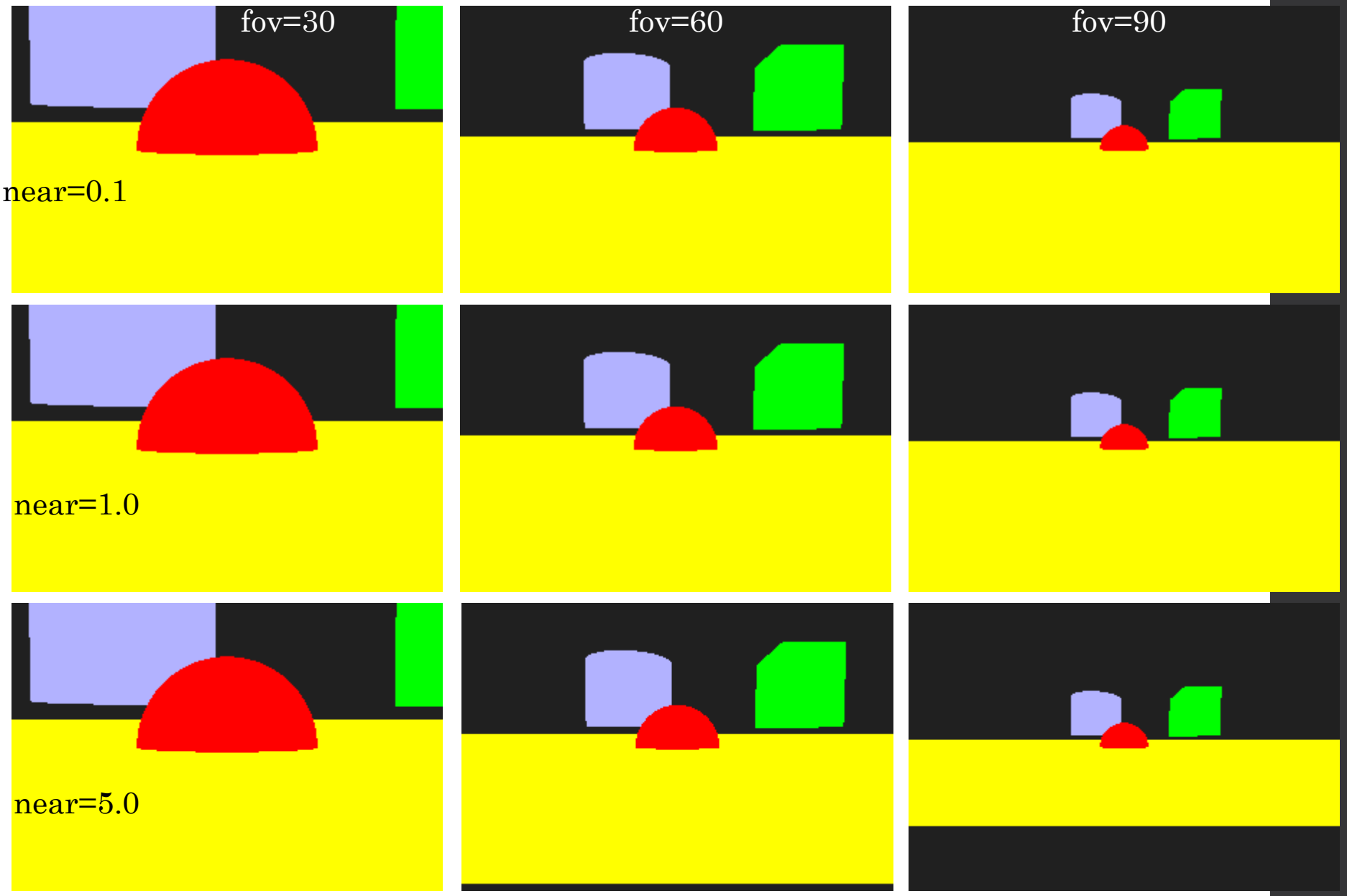


Note: in the drawing, this is a 17x17 pygame surface

Virtual view plane, cont.

- The details given to us:
  - The width, height of the pygame window
  - The **near** distance (how far in front of the camera is the plane, in virutal world-units [NOT pixels])
  - The (vertical) **field-of-view** (the angle made by the camera and the top-middle and bottom-middle points on the view plane)
  - [See next slide for an illustration of the role of both of these]
- We need to compute:
  - The width and height of the view plane (in virtual world units)
  - The position (in 3d) of the upper-left corner of the virtual view plane (that corresponds to the origin in the pygame window)
- [Do it on the board…]

FOV & Near's role (camera is the same in all cases)



Step3. Calculate the 3d position of an arbitrary pygame pixel

- You'll be given:
  - (ix, iy): integer positions on the pygame window.
  - view_width and view_plane_height and $\overrightarrow{view\_plane\_origin}$ (from previous calculations)
- Find the 3d position of that pixel's counterpart in 3d.
- [Do it on the board…]

Step 4: Tying it all together

- An outline of the RayTracer:
  - For each pixel (ix, iy)
    - Calculate the 3d counter-part to (ix, iy) [step3]
    - Create a Ray (origin = camera, direction = away from camera [for perspective effect])
    - [Talk briefly about Orthogonal projections]
    - See if that ray hits any objects in the scene:
      - If not, set (ix, iy) to a background color
      - If so, get the color of the closest hit point / object and set (ix, iy) to that color
- Some considerations:
  - Raytracing takes a long time – don't "freeze" the program.
    - Note our "one-line-at-a-time" approach.
  - We'll modify the last step late to include lighting / shading.

Just for fun

- Paul Heckbert: 1984 challenge
- Andrew Kensler's solution
  - http://fabiensanglard.net/rayTracing_back_of_business_card/