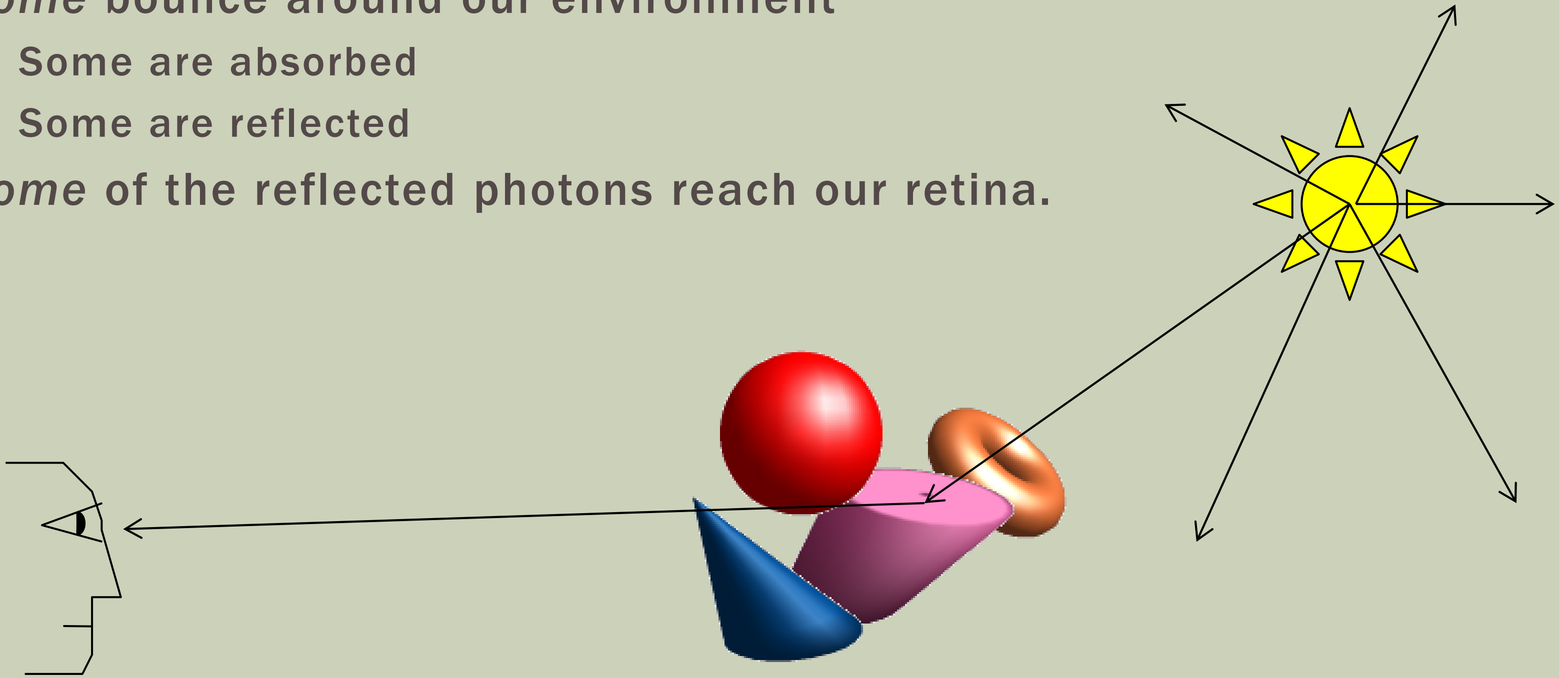
RAYTRACER PART I OF II

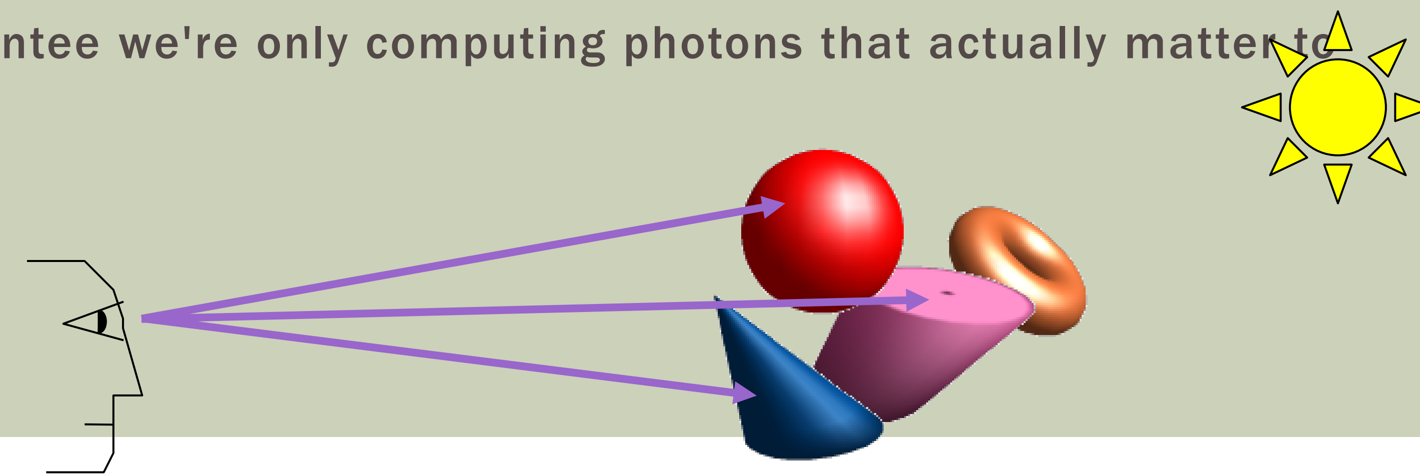
RAYTRACING OVERVIEW

- 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.

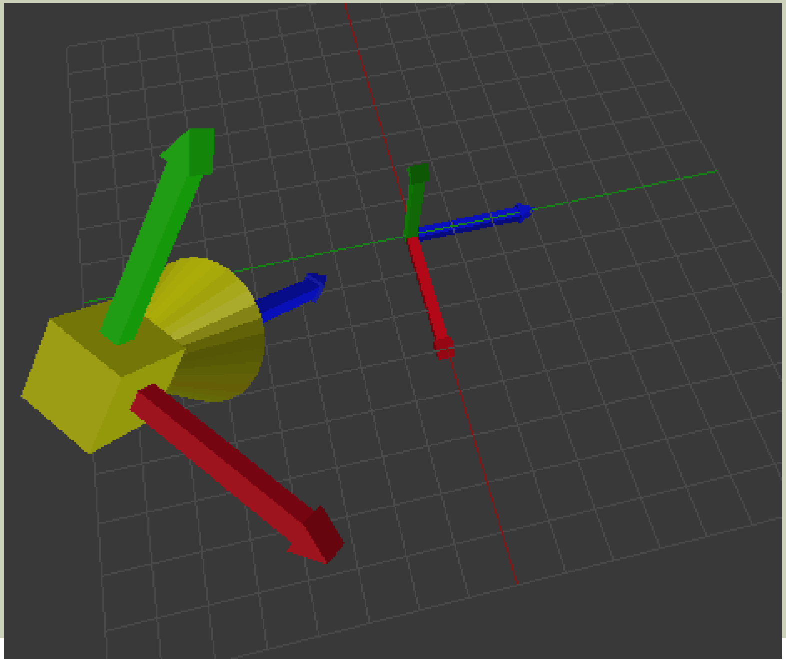
- This is the same technique used in early fps-games
- Technically, this is a **ray-casting.**

OVERVIEW, CONT.

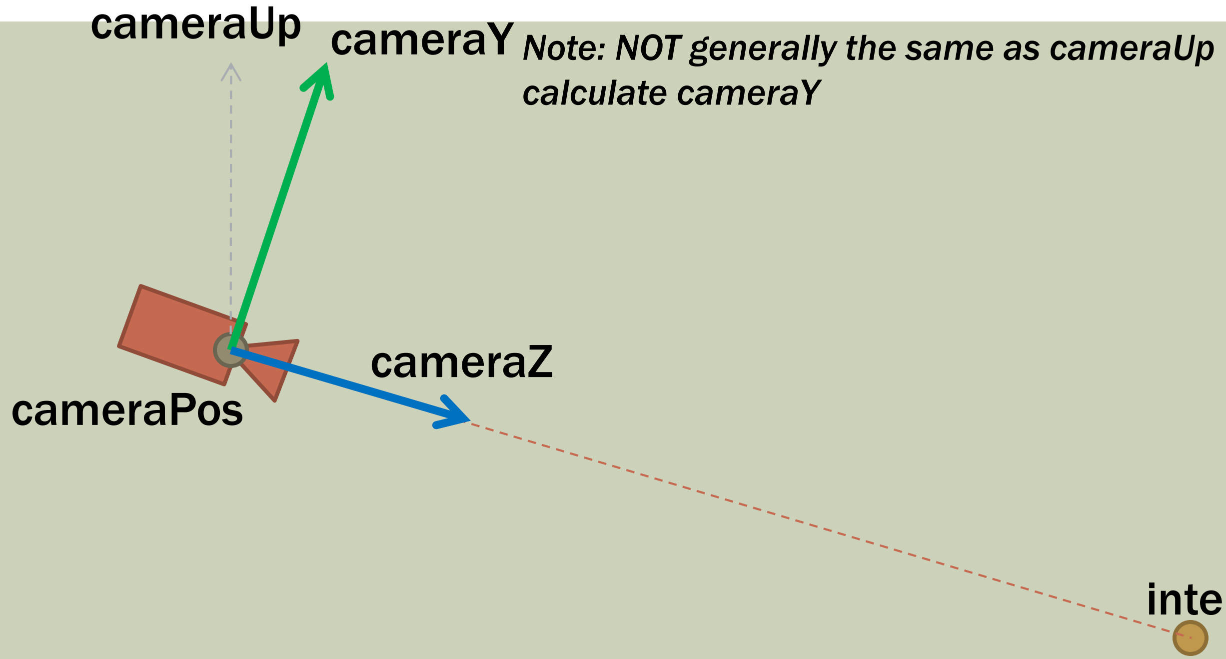
- More advanced renderings can be obtained by **recursively** bouncing rays off hit objects
  - Reflections
  - Refractions
  - Ambient Occlusion
  - Subsurface scatter
  - …

PHASE 1: DEFINE CAMERA SPACE

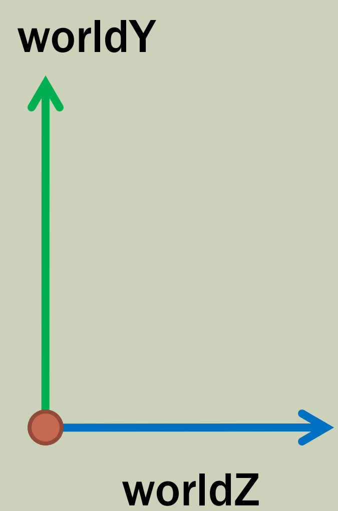
- We'll define camera space:
  - origin is (virtual) camera position.
  - axes are perpendicular and define a Left-handed coordinate system (since our world does)
  - Imagine yourself where the camera is (and oriented with the camera) – camera C.S. should look to you like world C.S.



CAMERA SPACE, CONT.



Note: NOT generally the same as cameraUp – but we use it to calculate cameraY



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

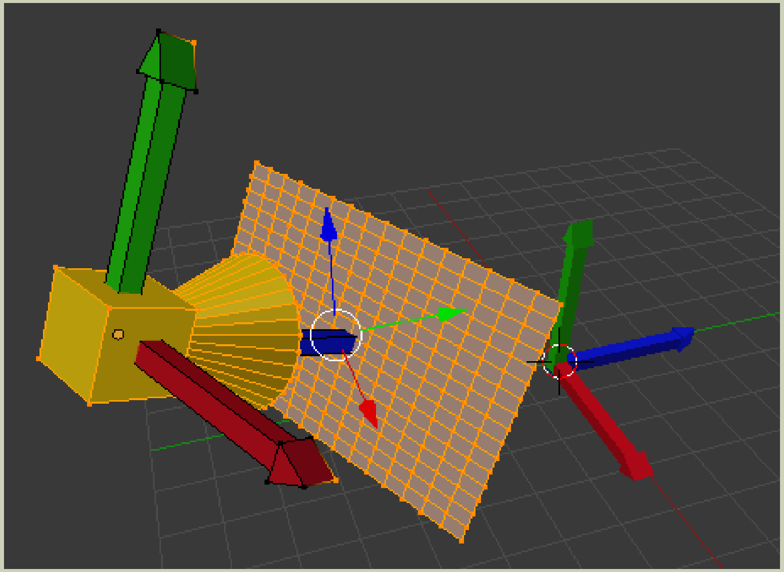
CameraY is perpendicular to the plane defined by cameraZ and cameraX.

CAMERA SPACE, CONT.

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

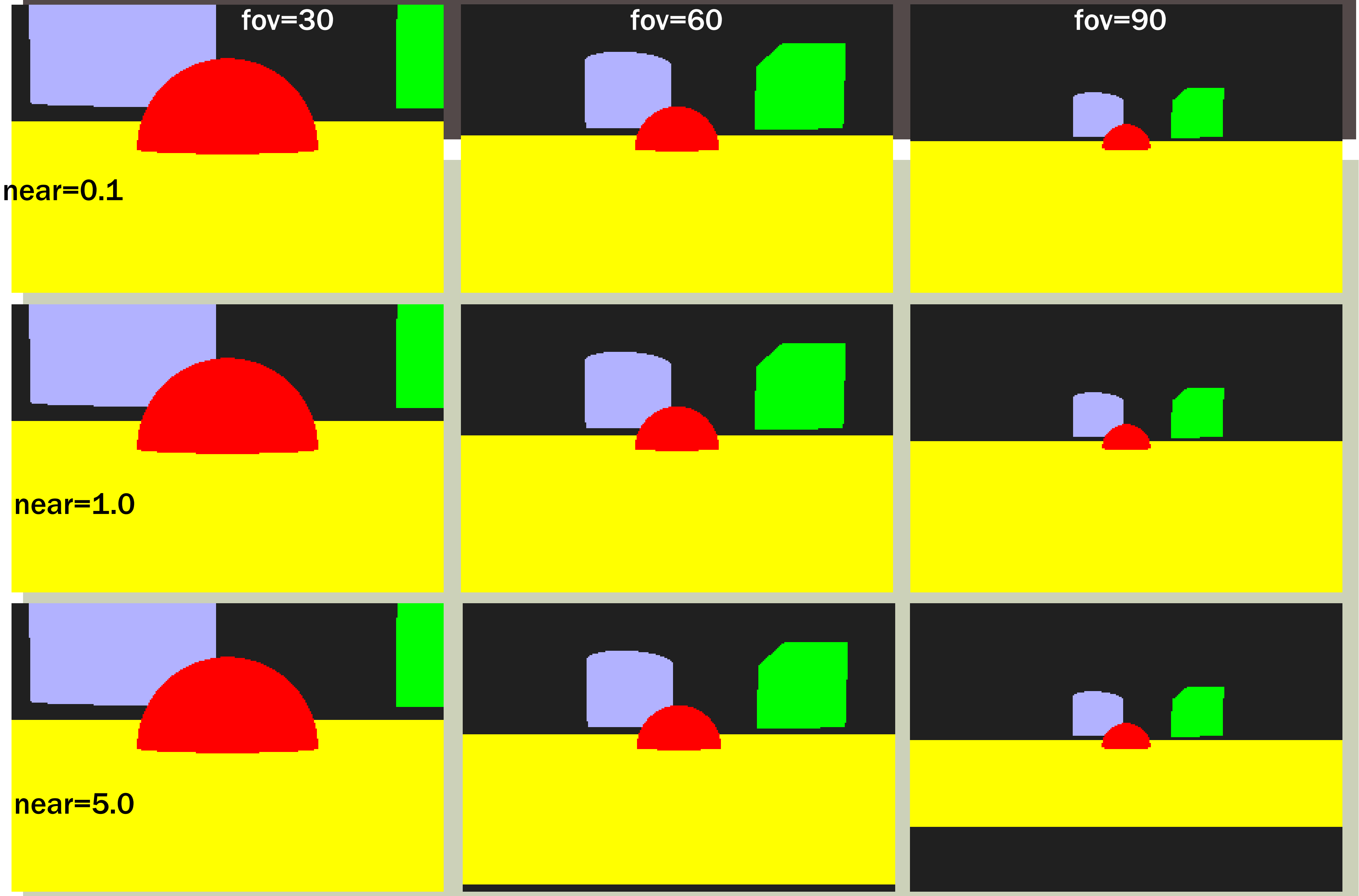


Note: in the drawing, this is a 17x17 pygame surface

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)



PHASE 3: GEOMETRIC PRIMITIVES

- Now we need something to render!
- In rasterizers (later), everything is polygon-based.
- In raytracers, it *can* be polygon-based (see the bonus section)
  - More often, though, it is a symbolic formula.
- 3 common ways to define a primitive symbolically:
  - **Implicit**: A definition of all points on the surface (a test)
    - Example: points on a (3d) sphere:
      - $x^2 + y^2 + z^2 = r^2$
  - **Parametric**: A way to generate all points on the surface
    - Example: t in the range 0…1, points on a (2D) unit circle centered at origin.
      - $\overrightarrow{P(t)} = [\cos(2\pi t) \quad \sin(2\pi t)]$
  - **"Straightforward"**: The usual way we explain the surface
    - Example: Sphere
      - Specify center (vector3) and radius (scalar)
    - The way we'll implement it in python

(INFINITE) PLANES (9.5)

- Implicit form: $\vec{p} \bullet \hat{n} = d$

  Or as scalars (in 3D): $ax + by + cz = d$

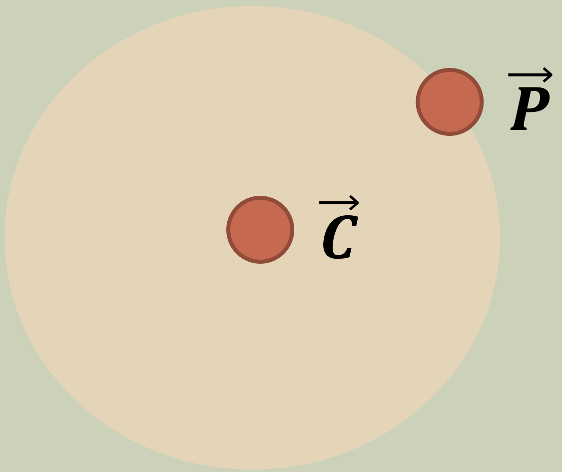
    Where [a, b, c] is $\hat{n}$ and [x, y, z] is $\vec{p}$
- Graphical Interpretation of $\hat{n}$ and d:
- [On board]
  - Above / on / below "test"
  - Finding closest point on a plane to another point (possibly not on the plane)
  - Drawing in pygame
  - Ray intersection

SPHEROIDS (9.3)

- Implicit form

$$\left\|\vec{P}-\vec{C}\right\| = r \qquad \text{or} \qquad \left(\vec{P}-\vec{C}\right)\bullet\left(\vec{P}-\vec{C}\right) = r^2$$

  - Where
    - P is any point on the spheroid
    - C is the center of the spheroid
    - r is the radius of the spheroid
- [On board]
  - in / out / on test
  - Ray intersection test



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…]

PHASE 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 <u>closest</u> 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.

PHASE 5!: FOR THE BRAVE…

- I won’t go through these in class (at least until we’ve gone through the normal material)
- These are two additional primitives you can implement to earn some bonus points.

AABB (9.4.1)

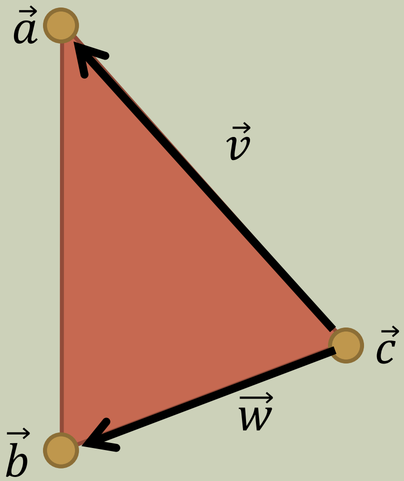
- Straightforward:
  - Define exactly 2 of these (VectorN's)
    - pmin: the minimum x/y/z value of the box
    - pmax: the maximum x/y/z value of the box
    - center: the middle position of the box
    - pextents: a vector which is long enough to connect pmin & pmax
      - Or said another way, the elements are width / height / depth.
  - Note: given any two, you can derive the other two.
- A good "rough test" for complex primitives

AABB

- The easy way to do a Ray-AABB hit-test is to:
  - define a set of 6 planes (in 3d) (probably in __init__)
  - When given a ray, test it against all 6 planes. But…
  - …hitting the plane isn't good enough:
    - If hitting either the left or right plane: the hit point must satisfy:
      - pmin.y <= hitPt.y <= pmax.y
      - pmin.z <= hitPt.z <= pmax.z
    - If hitting the top / bottom plane, the hit point must satisfy:
      - pmin.x <= hitPt.x <= pmax.x
      - pmin.z <= hitPt.z <= pmax.z
    - If hitting the front / back plane (normals of plane in z direction), this hit point must satisfy:
      - pmin.x <= hitPt.x <= pmax.x
      - pmin.y <= hitPt.y <= pmax.y
  - If the hit point satisfies these, count it as a hit with the box.

POLYGON

- A polygon is a planar collection of points
  - Note: most modellers (blender / maya) will allow non-planar poly's.
  - This math won't work without them.
  - To be sure, you can triangulate your mesh (triangles are *always* planar)
- Recall (from Lecture4) that the *area* of a triangle made up of 3 points is calculated as:



$$\vec{v} = \vec{a} - \vec{c}$$

$$\vec{w} = \vec{b} - \vec{c}$$

$$area(\Delta abc) = \frac{\|\vec{v}\ x\ \vec{w}\|}{2}$$

- To determine if a point is within a triangle, calculate the **barycentric coordinates** for each point in triangle:
  - **bary**($\vec{a}$) = $area(\Delta pcb)/area(\Delta abc)$
  - **bary**($\vec{b}$) = $area(\Delta pad)/area(\Delta abc)$
  - **bary**($\vec{c}$) = $area(\Delta pab)/area(\Delta abc)$
- **Note:**
  - $\vec{p} = bary(\vec{a}) * \vec{a} + bary(\vec{b}) * \vec{b} + bary(\vec{c}) * \vec{c}$
- **The point p can only be in the triangle if:**
  - $1 - \varepsilon \leq bary(\vec{a}) + bary(\vec{b}) + bary(\vec{c}) \leq 1 + \varepsilon$
  - ε is a small number (0.0001) needed for float errors
- So the Ray-triangle test is:
  - See if Ray hits plane.
  - If it does, do the barycentric test above.