Camera

Bo Zhu

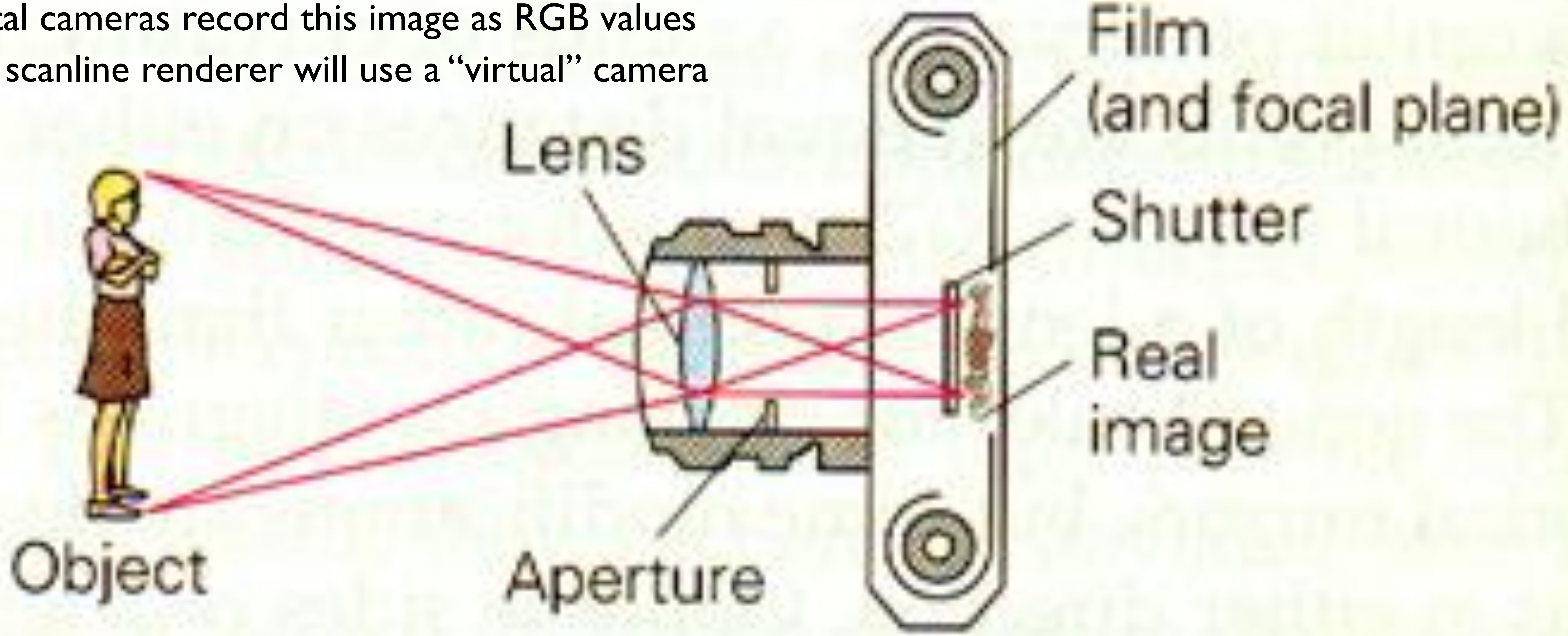
School of Interactive Computing

Georgia Institute of Technology

Motivational Video: Photography: The Rule of Thirds

Cameras

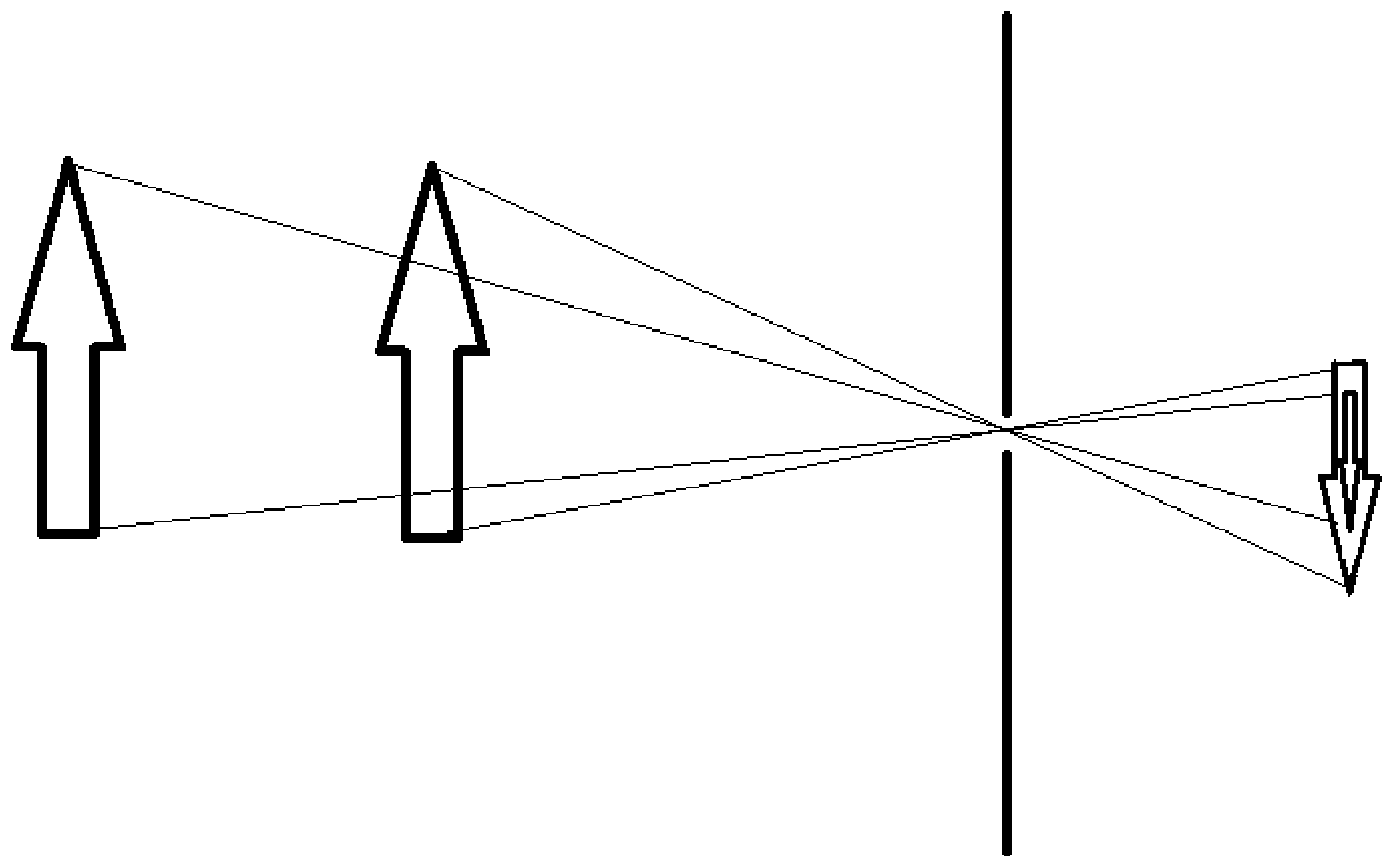
- Cameras work very much like the eye
  - Light from the environment is bent by the lens array to make an image on the film
  - Digital cameras record this image as RGB values
  - Our scanline renderer will use a "virtual" camera



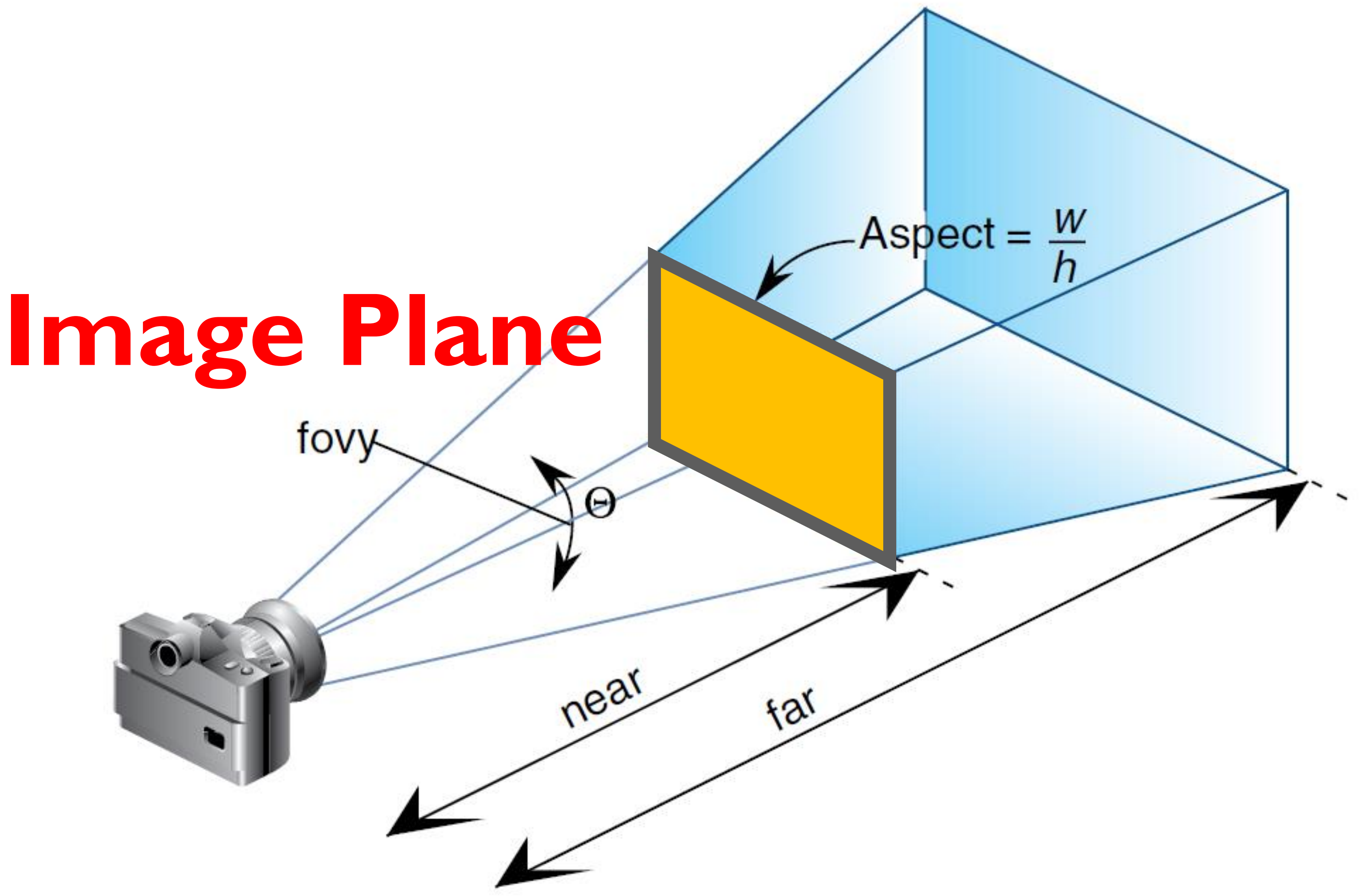
Pinhole Camera

- Simplified theoretical construct, but similar to an actual eye or camera
- Eyes and cameras can't have VERY small holes, because that limits the amount of entering light

Pinhole Camera



- Light leaving any point travels in straight lines
- We only care about the lines that hit the pinhole (a single point)
- Infinite depth of field – i.e., everything is in focus (no blur)
- An upside down image is formed by the intersection of these lines with an image plane
- More distant objects subtend smaller visual angles and appear smaller
- Objects occlude the objects behind them



Camera in Rendering Pipeline

- The modern scanline renderer uses a pinhole camera.
- However, the image plane (i.e. the film) is placed in front of the pinhole (rather than behind the pinhole in the physical world), so that the image is not upside down

In order to describe this camera model, you need to specify two things:

(1) Where the camera is, and where is it pointing to;

(2) Where the image plane is in front of the camera, and what's its depth.

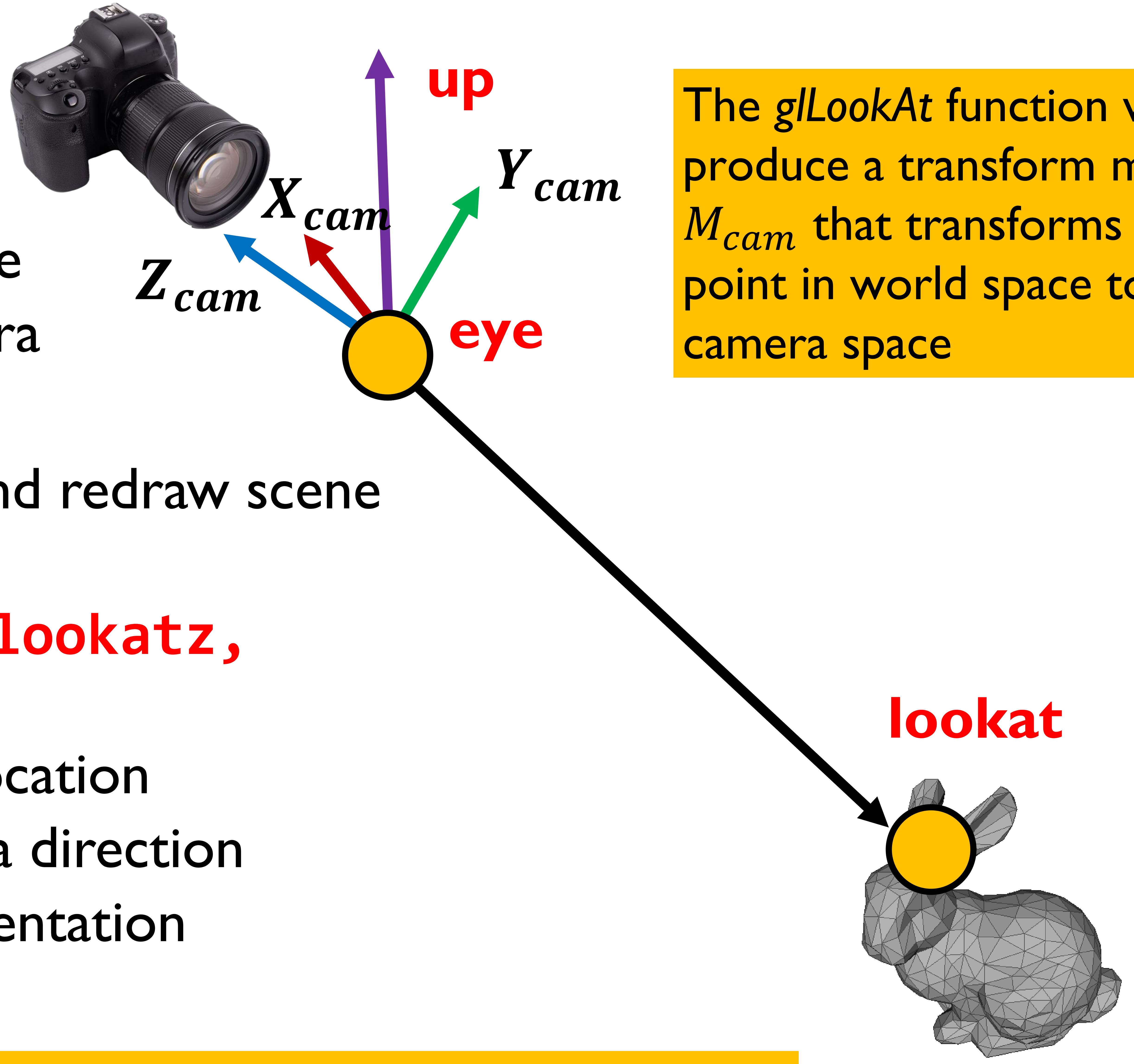
This is like how you take a photo in the real world

(1) **Move around** to center the target object within the camera

(2) **Adjust lens** of the camera to best fit the scene with the scope

Both steps are implemented with matrix multiplications

Move Around: ***glLookAt***



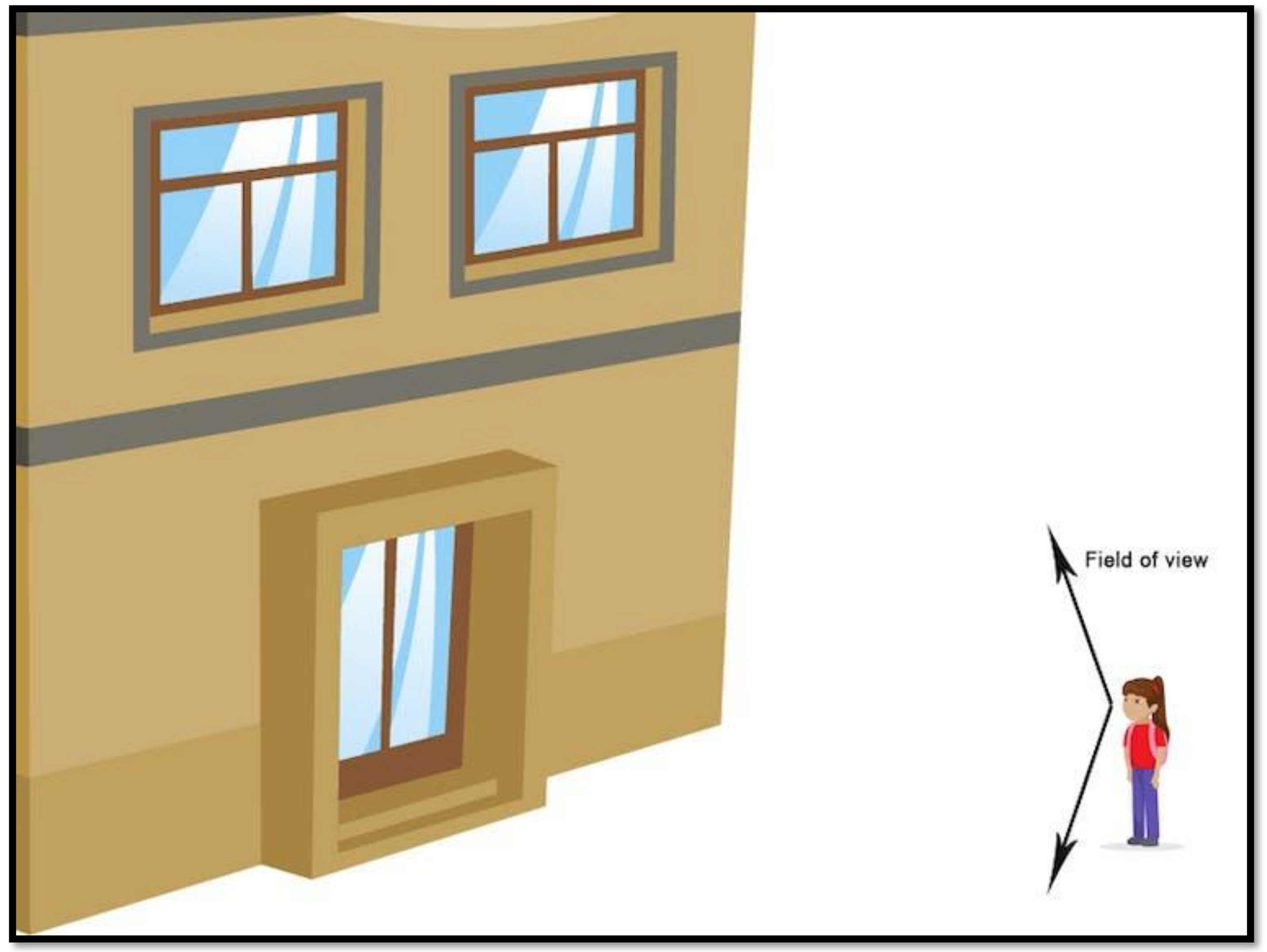
The *glLookAt* function will produce a transform matrix M_{cam} that transforms a point in world space to camera space

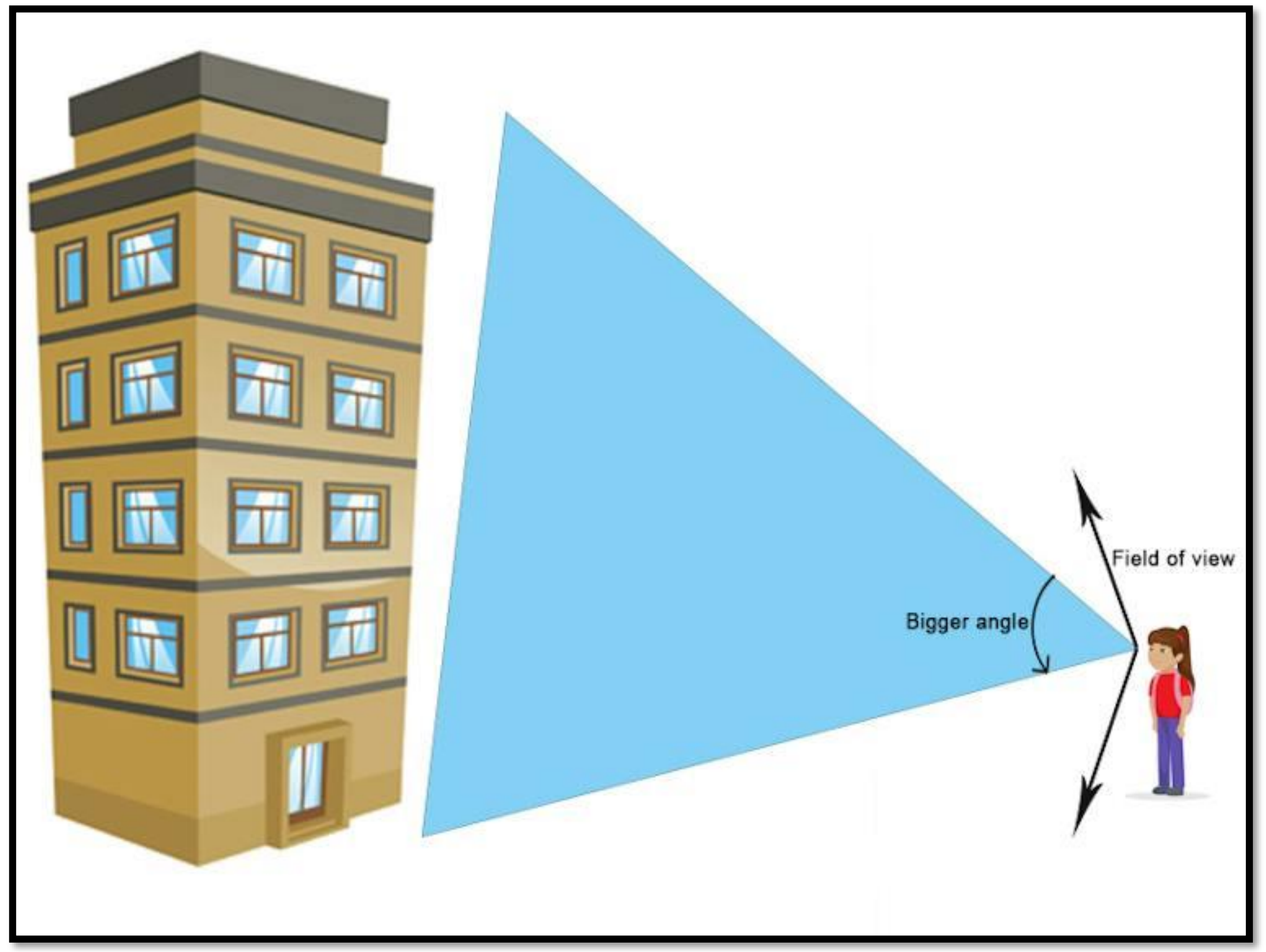
- Position the camera/eye in the scene
  - place the tripod down; aim camera
- To "fly through" a scene
  - change viewing transformation and redraw scene
- `LookAt( eyex, eyey, eyez, lookatx, lookaty, lookatz, upx, upy, upz )`
  - eye vector decides the camera location
  - lookat vector decides the camera direction
  - up vector determines unique orientation

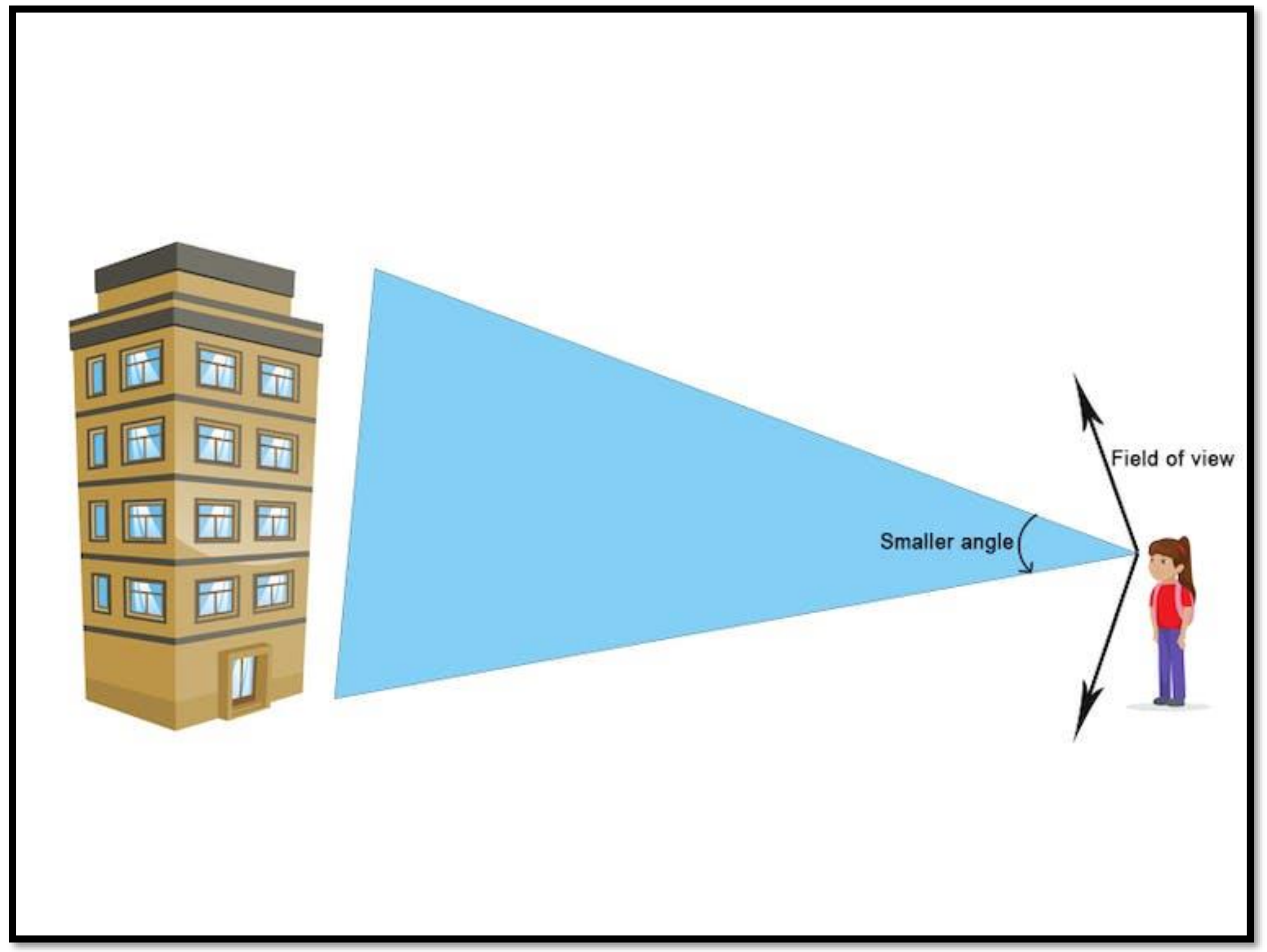
How do we understand this transformation?


be Stock | #255664678

Things appear smaller the further you are from them. Why?







Field of View

- This is how much you can see, without turning your head.
- When things are closer to you, they take up more of your field of view, so they seem bigger.
- When they're further away, they take up less of your field of view, and so seem smaller.

How do we reproduce the effect of FoV in our camera model?

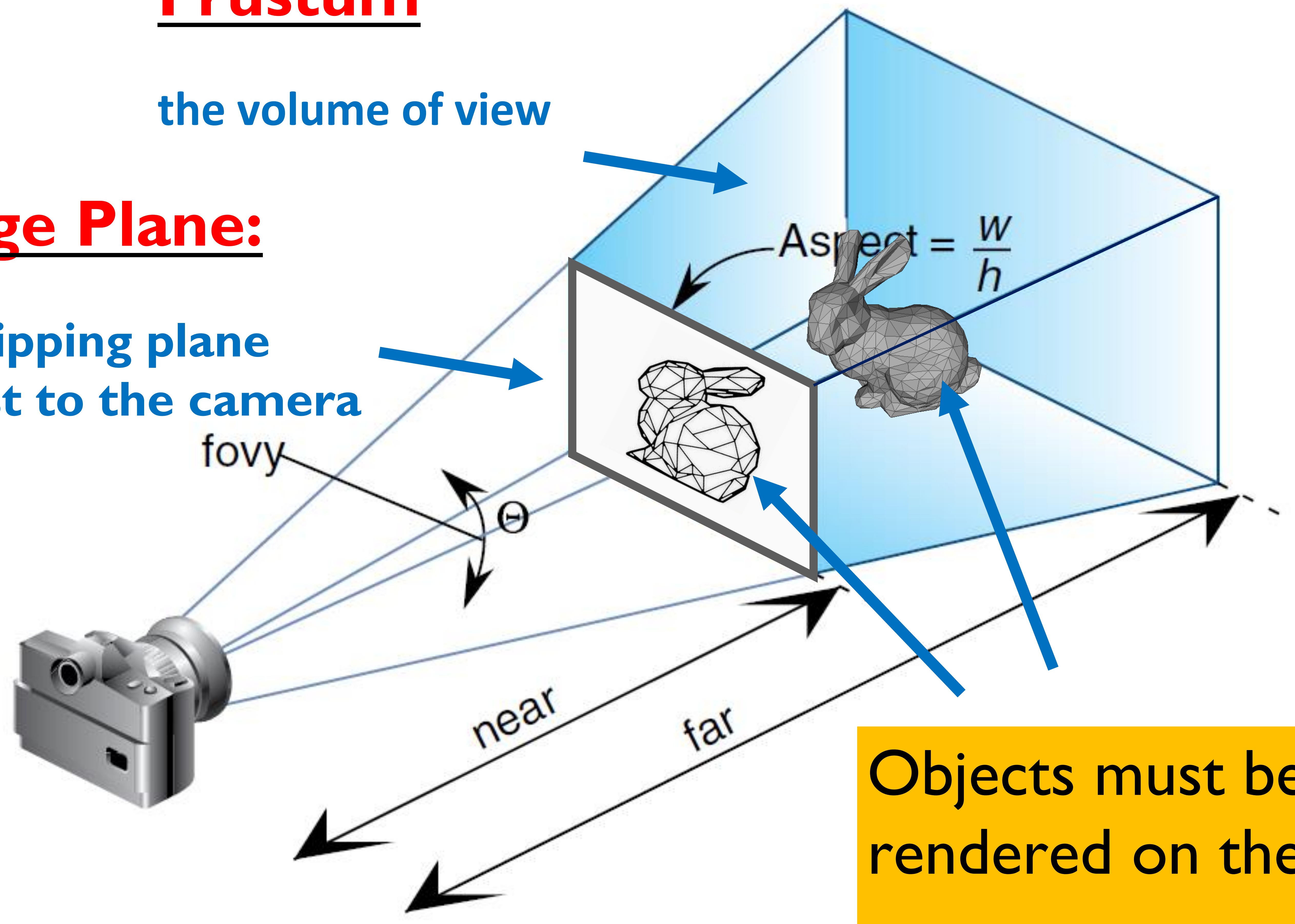
Adjusting Lens:
Viewing Volume (Frustum) and Image Plane

Frustum

the volume of view

Image Plane:

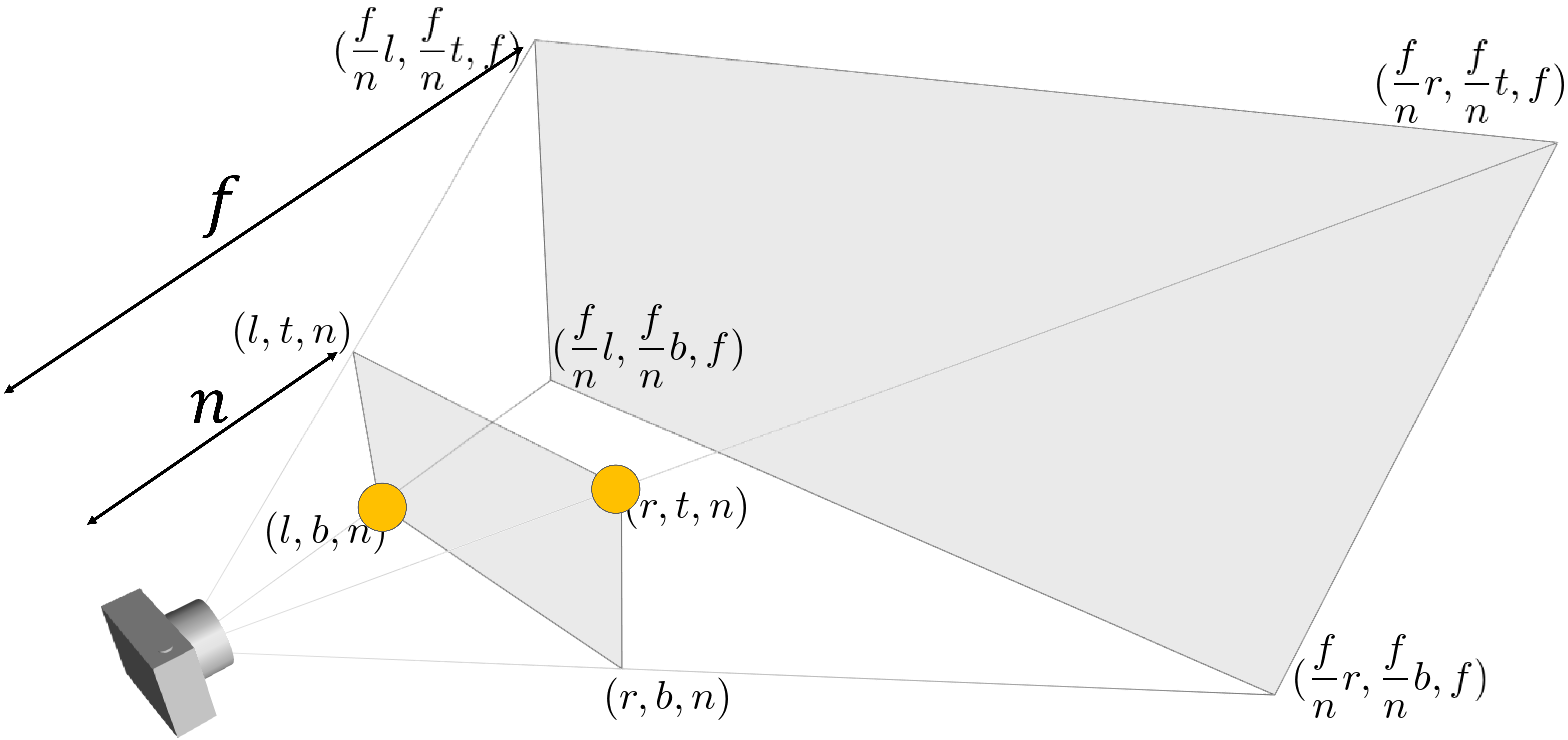
the clipping plane closest to the camera



Objects must be placed within the frustum to be rendered on the image plane;

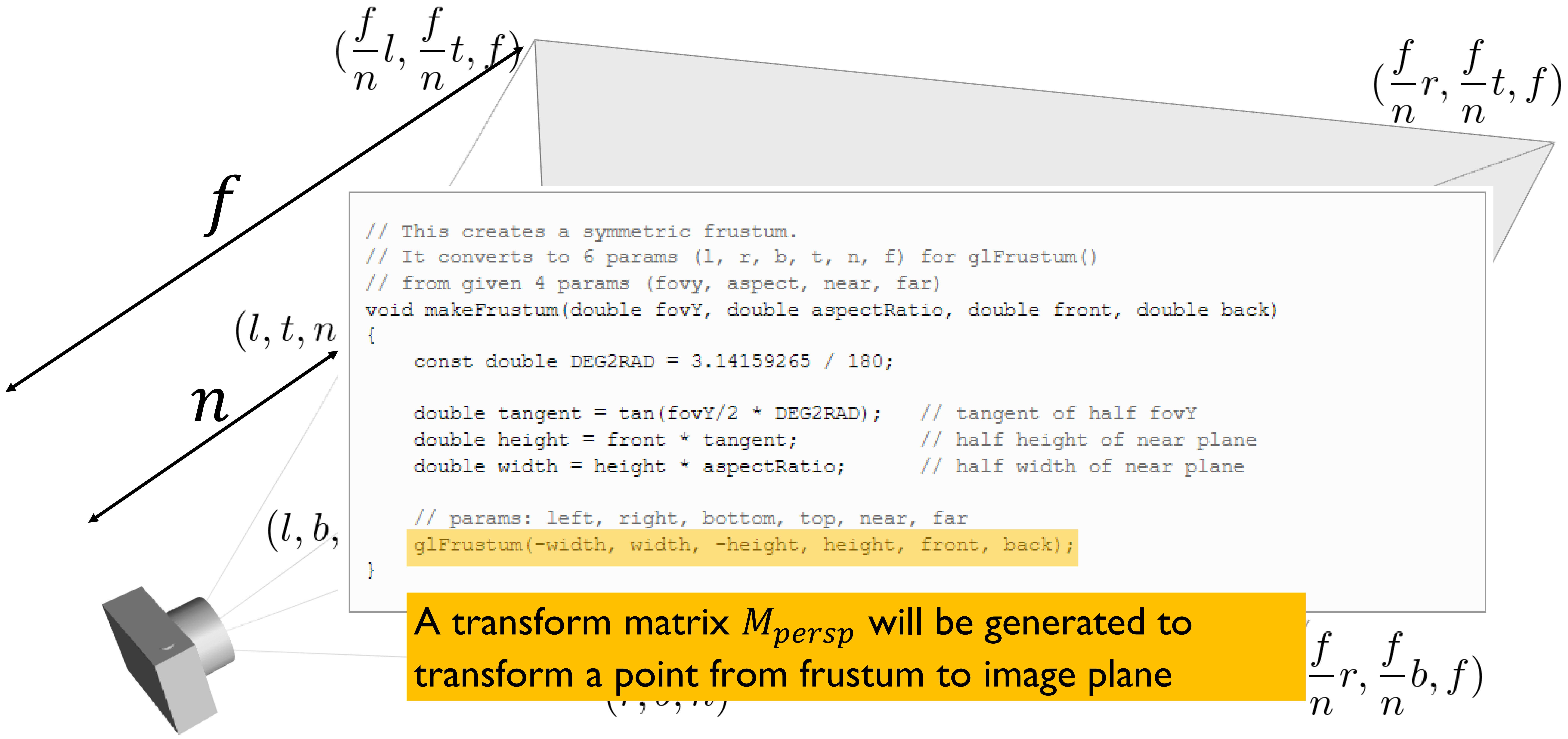
Objects outside the frustum won't be rendered

How do we specify a frustum mathematically?



We only need to specify the corners of the image plane, and the z depths of near and far clipping planes

OpenGL Implementation: ***glFrustum***

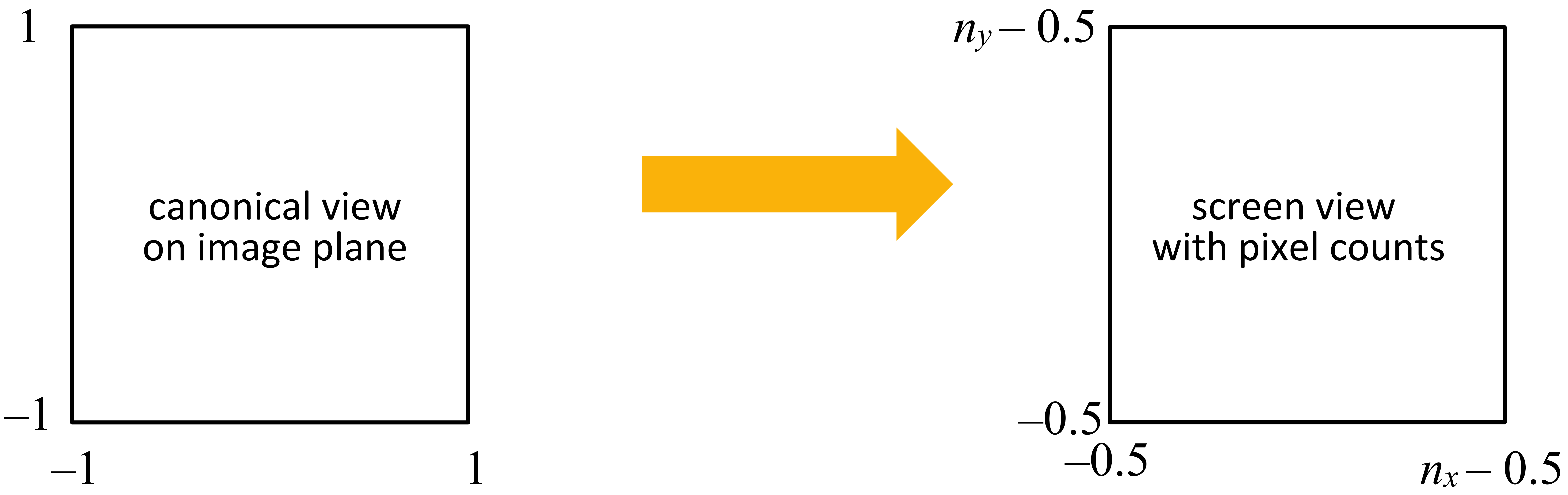



```
// This creates a symmetric frustum.
// It converts to 6 params (l, r, b, t, n, f) for glFrustum()
// from given 4 params (fovy, aspect, near, far)
void makeFrustum(double fovY, double aspectRatio, double front, double back)
{
    const double DEG2RAD = 3.14159265 / 180;

    double tangent = tan(fovY/2 * DEG2RAD);    // tangent of half fovY
    double height = front * tangent;           // half height of near plane
    double width = height * aspectRatio;       // half width of near plane

    // params: left, right, bottom, top, near, far
    glFrustum(-width, width, -height, height, front, back);
}
```

A transform matrix M_{persp} will be generated to transform a point from frustum to image plane

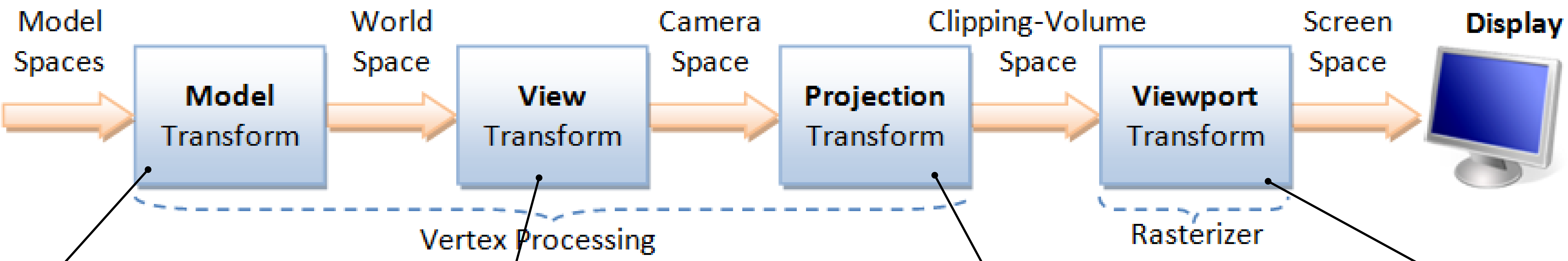
We only need to specify the corners of the image plane, and the z depths of near and far clipping planes

Viewport Transform:
Last, we use another matrix to transform everything onto screen pixels



The *glViewport* function will produce a transform matrix $M_{viewport}$ that transforms a point in image space to screen space

Put everything together:
Full Pipeline of Model and Camera Transform



1. **Model**

map local object coords to world coords

(Done in the previous lecture)

2. **Viewing**

map world coords to camera coords

3. **Projection**

map camera coords to canonical view volume

4. **Viewport**

map canonical view volume to screen space

Camera Analogy - Four Stages

- **Model transform**
  - moving the model
- **Viewing transform**
  - tripod–define position and orientation of the viewing volume in the world
- **Projection transform**
  - adjust the lens of the camera
- **Viewport transform**
  - enlarge or reduce the physical photograph

Mathematically, this pipeline can be represented as **a chain of matrix multiplications**

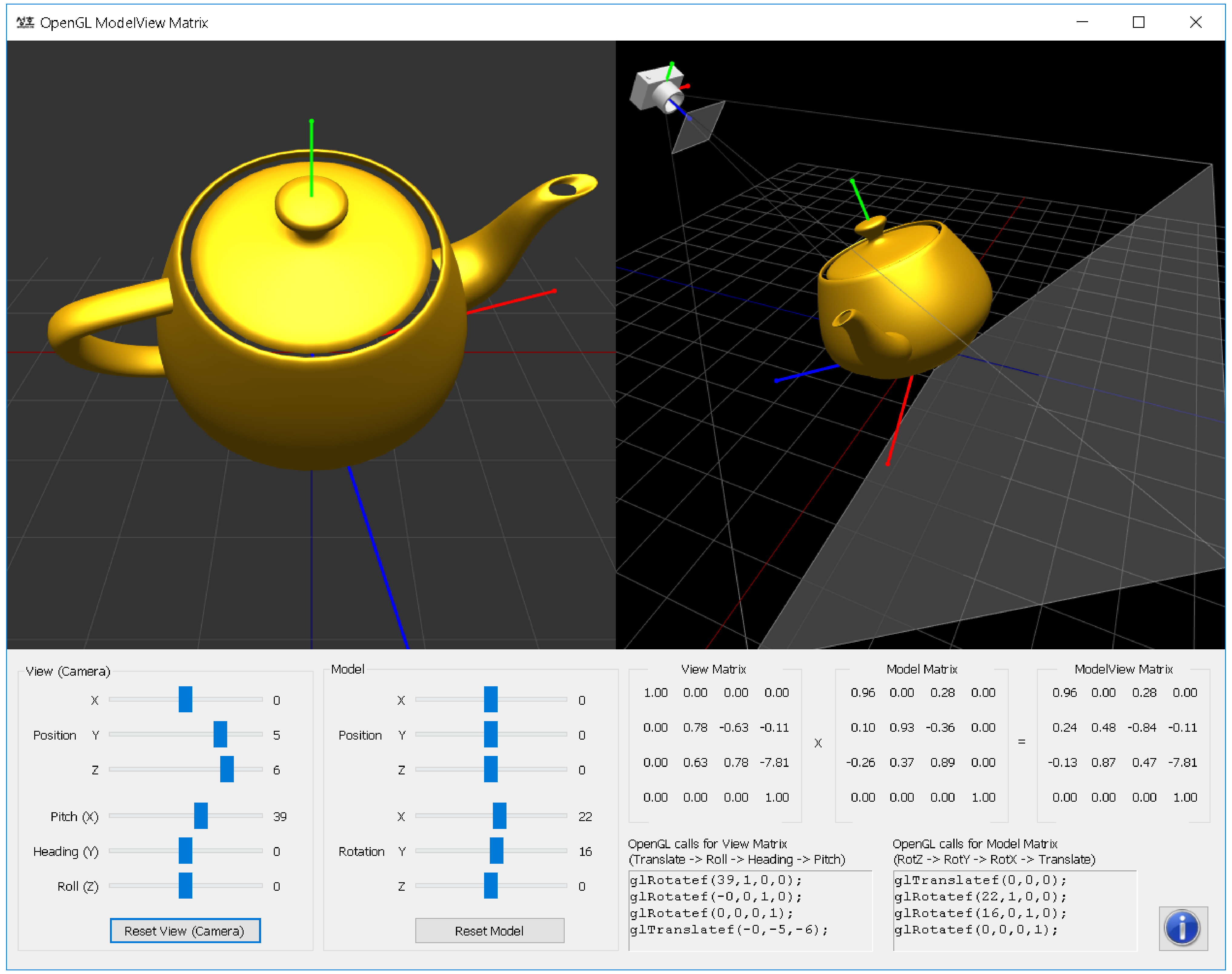
- Start with coordinates in object's local coordinates
- Transform into world coords (model transform, $\mathbf{M}_{\mathrm{m}}$)
- Transform into eye coords (view transform, $\mathbf{M}_{\mathrm{cam}}$)
- Perspective projection, $\mathbf{M}_{\mathrm{persp}}$
- Viewport transform, $\mathbf{M}_{\mathrm{vp}}$

$$\mathbf{p}_s = \mathbf{M}_{\mathrm{vp}}\mathbf{M}_{\mathrm{persp}}\mathbf{M}_{\mathrm{cam}}\mathbf{M}_{\mathrm{m}}\mathbf{p}_o$$

The model matrix is the only one you need to specify in your shader programs

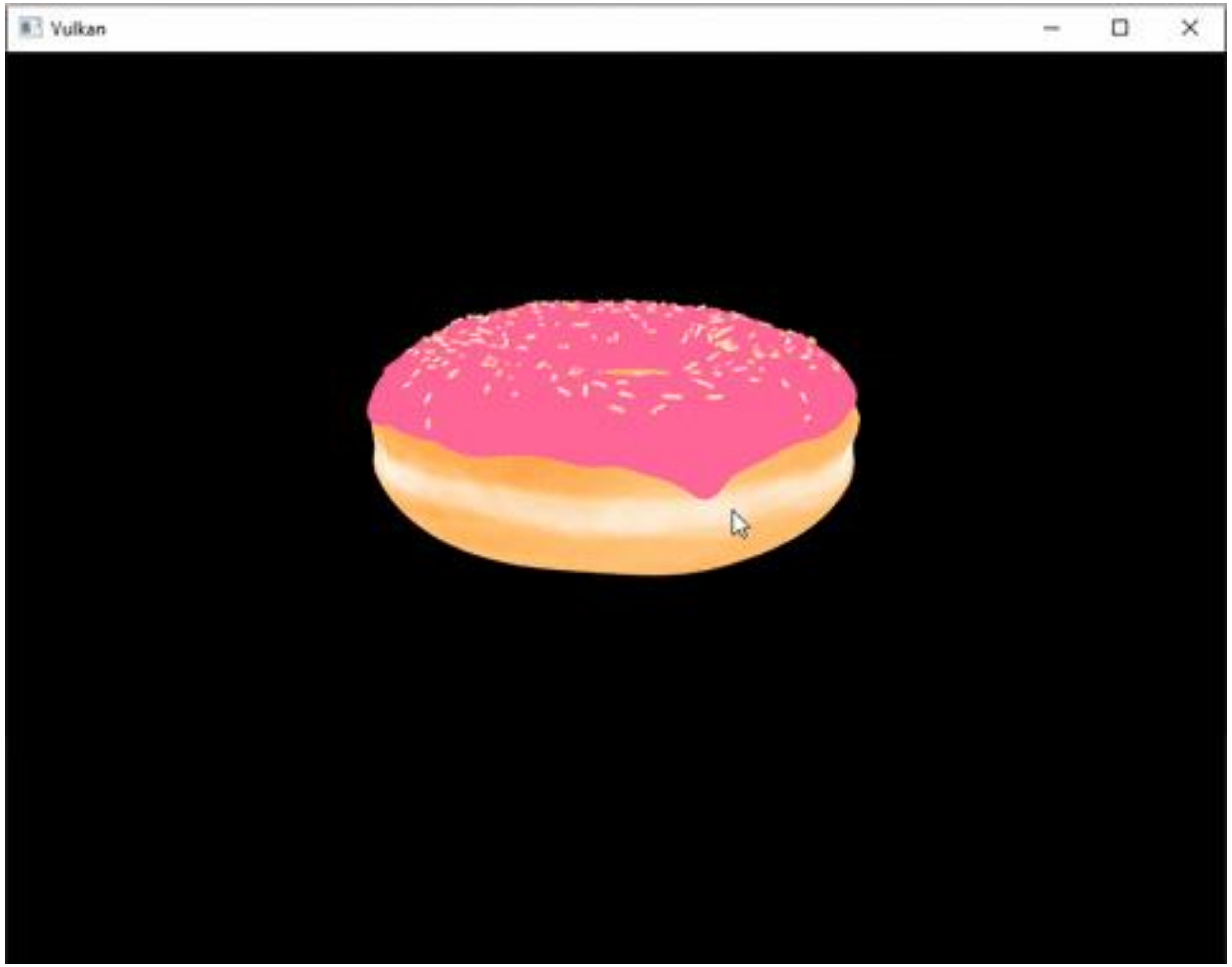
Live Demo for Model-View Transformation

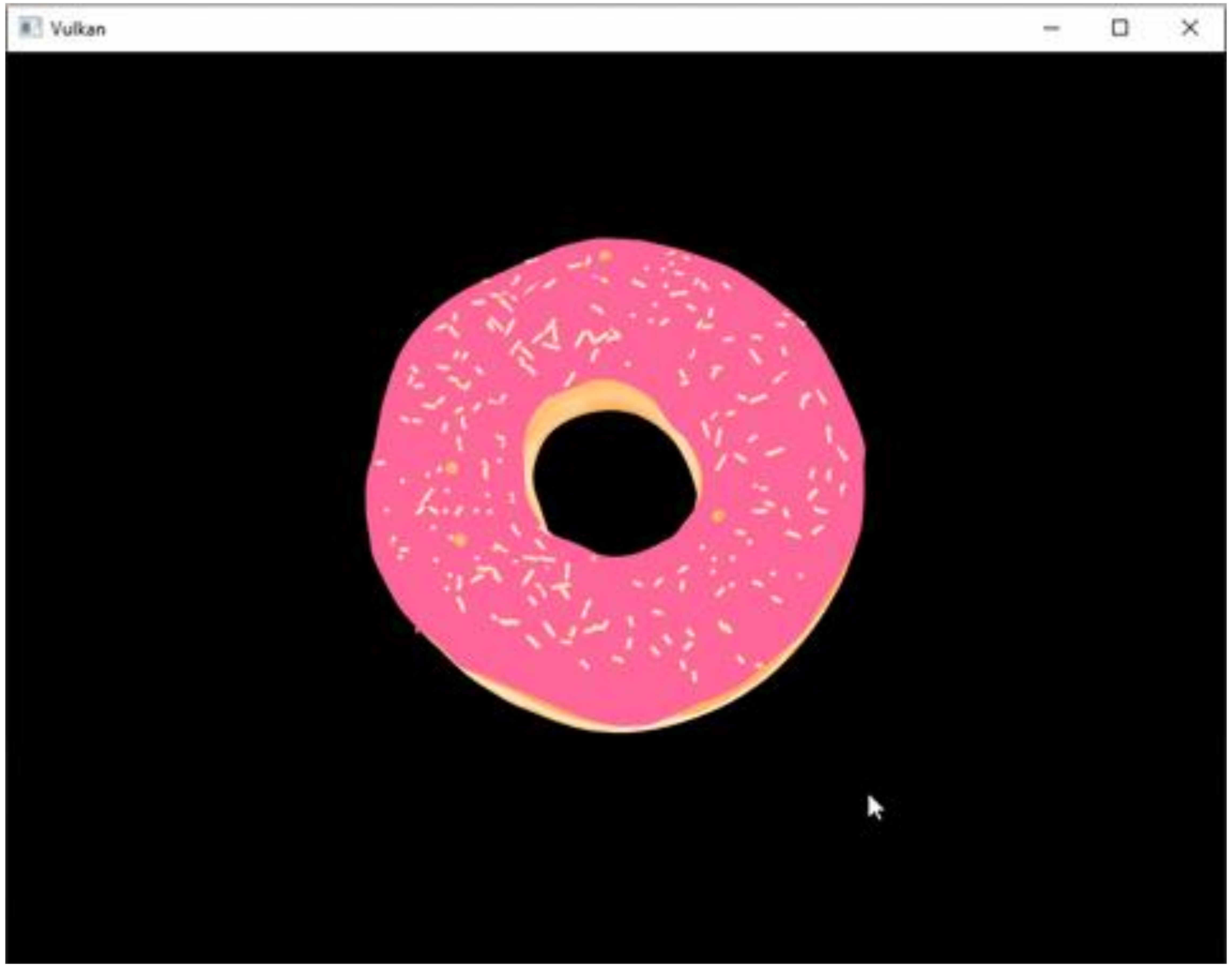
- http://www.songho.ca/opengl/gl_transform.html



How do we manipulate camera in a virtual 3D world?

- Typically, we directly manipulate the glLookAt() function by updating the eye, lookat, and up vectors.
- These vectors are usually tracked with interpolated spline functions to ensure smooth motions.

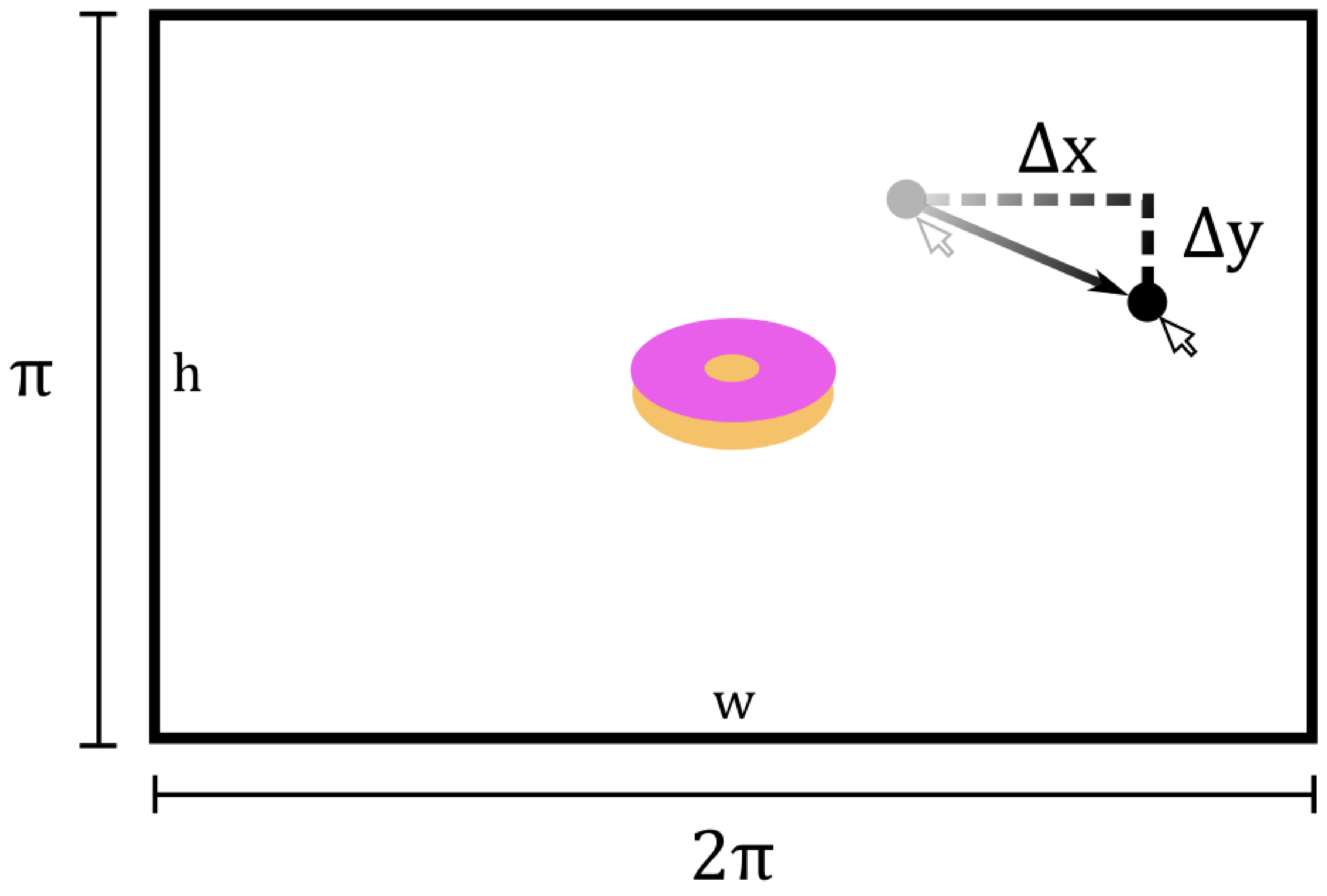


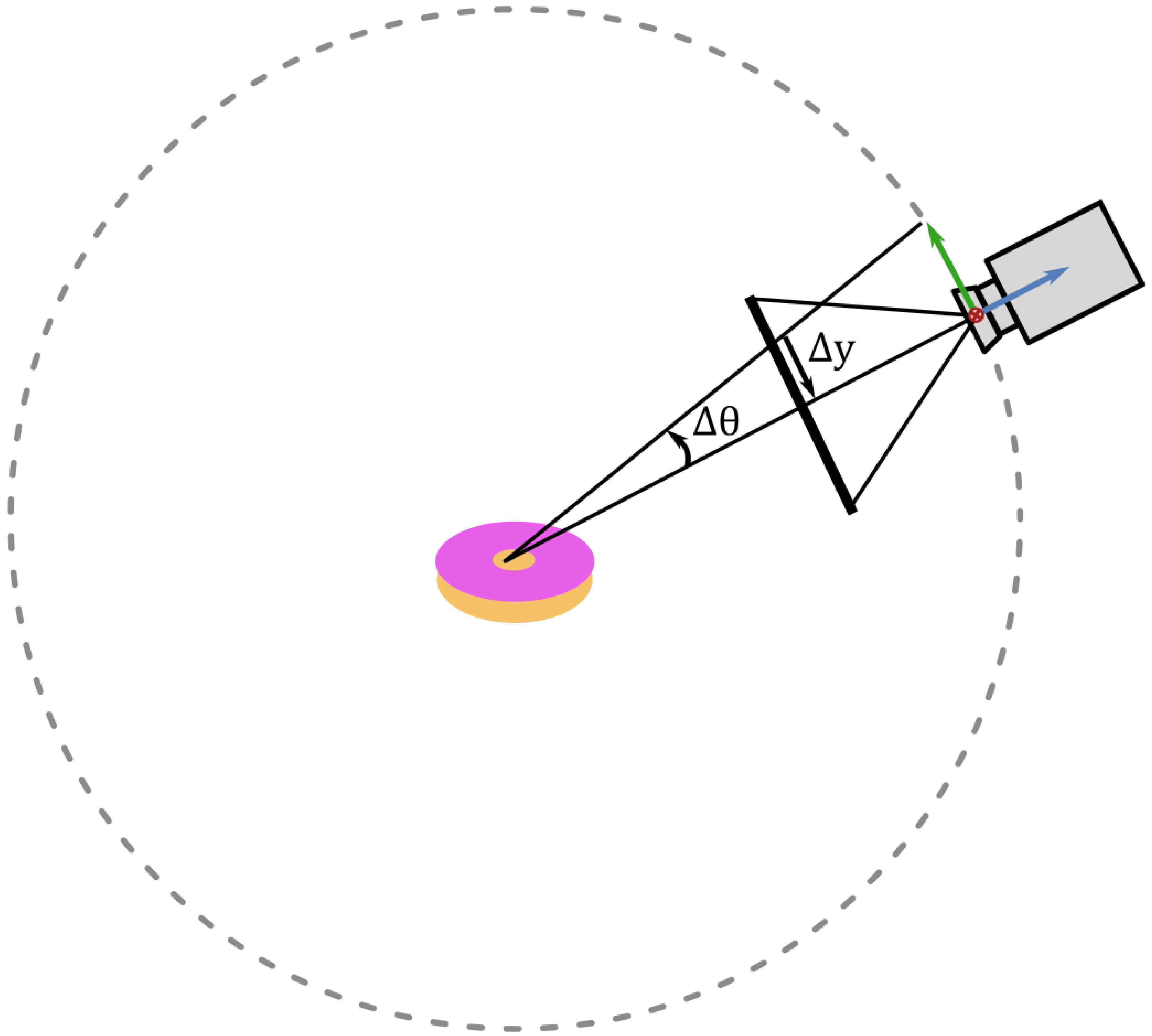


How do we manipulate camera on screen?

https://asliceofrendering.com/camera/2019/11/30/ArcballCamera

Arcball Algorithm





- Calculate the amount of rotation in x and y given the mouse movement.
- Rotate the camera of Θ_x radians around the up axis.
- Rotate the camera of Θ_y radians around the right axis.