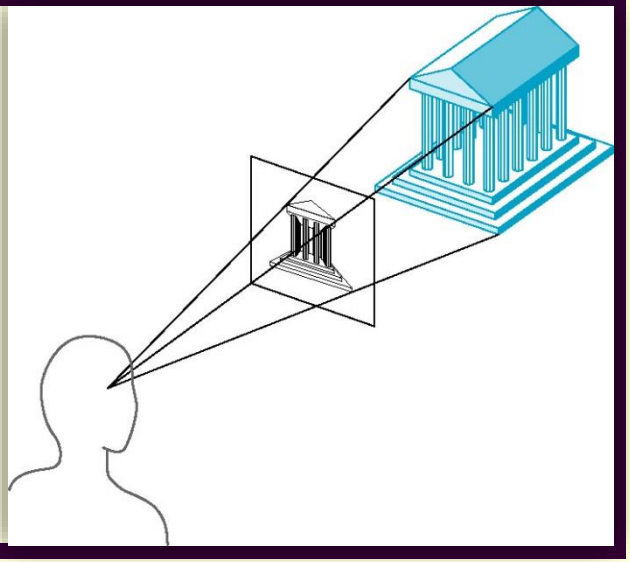
CS 413 Introduction to Ray and Vector Graphics
Chapter 8: Perspective Viewing

Instructor: **Joel Castellanos**

e-mail: joel@unm.edu

Web: http://cs.unm.edu/~joel/

Farris Engineering Center: 319



3/25/2019

Ray Tracer IV: Perspective Viewing

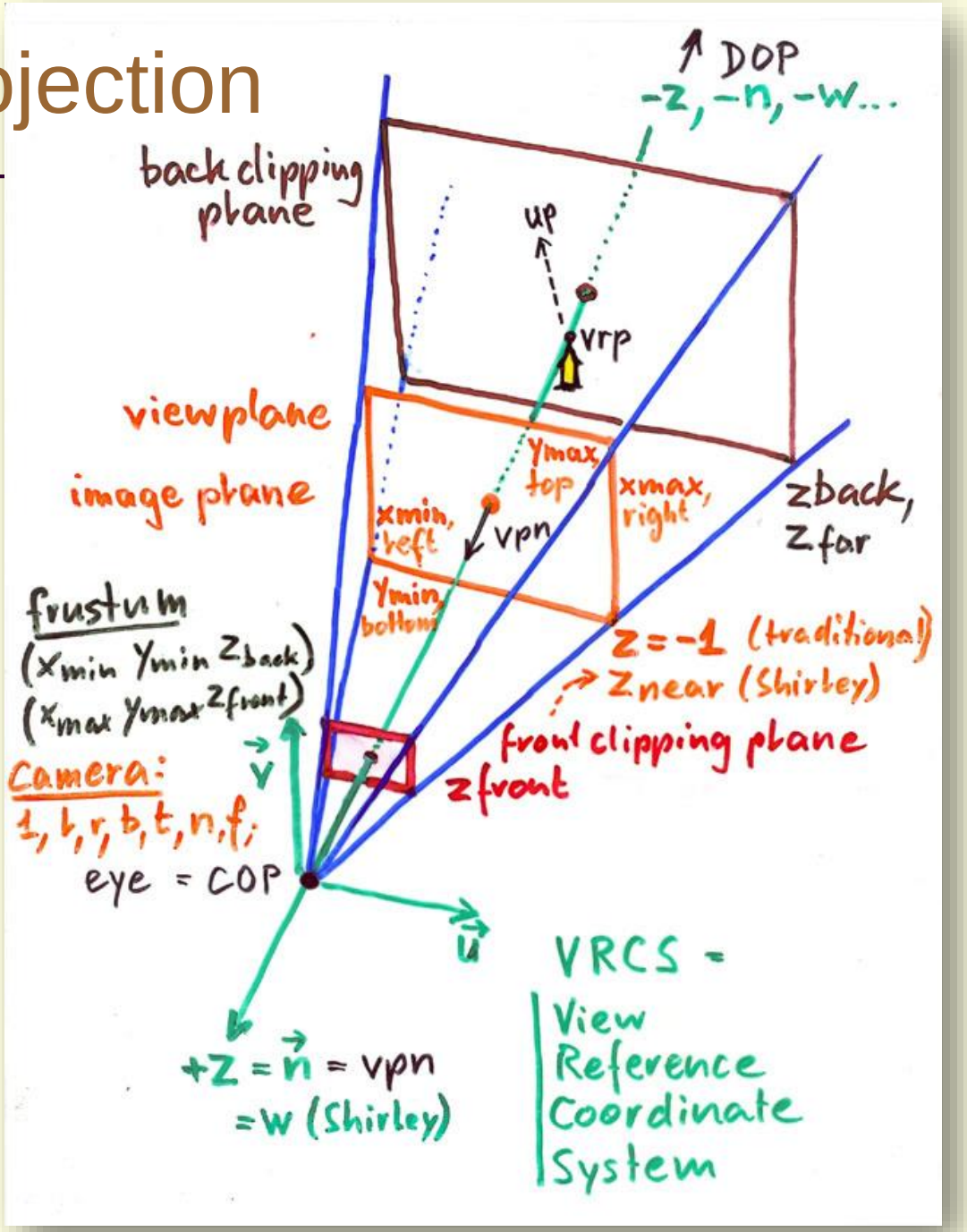
Implement a pinhole camera and use the objects in figure 9.9 of the book for testing: an olive sphere, a teal triangle and an orange rectangular solid on an infinite gray and white checkerboard floor in black world. Allow the user to:

a) Independently move the sphere's center, the triangle's normal vector and the triangle's centroid.
b) Specify an arbitrary camera location.
c) Specify an arbitrary view direction.
d) Specify an arbitrary up vector.
e) Specify a view plane distance.
f) Specify at least two different sampling techniques.
g) Specify total sample points.

Notes on Ray Tracer IV

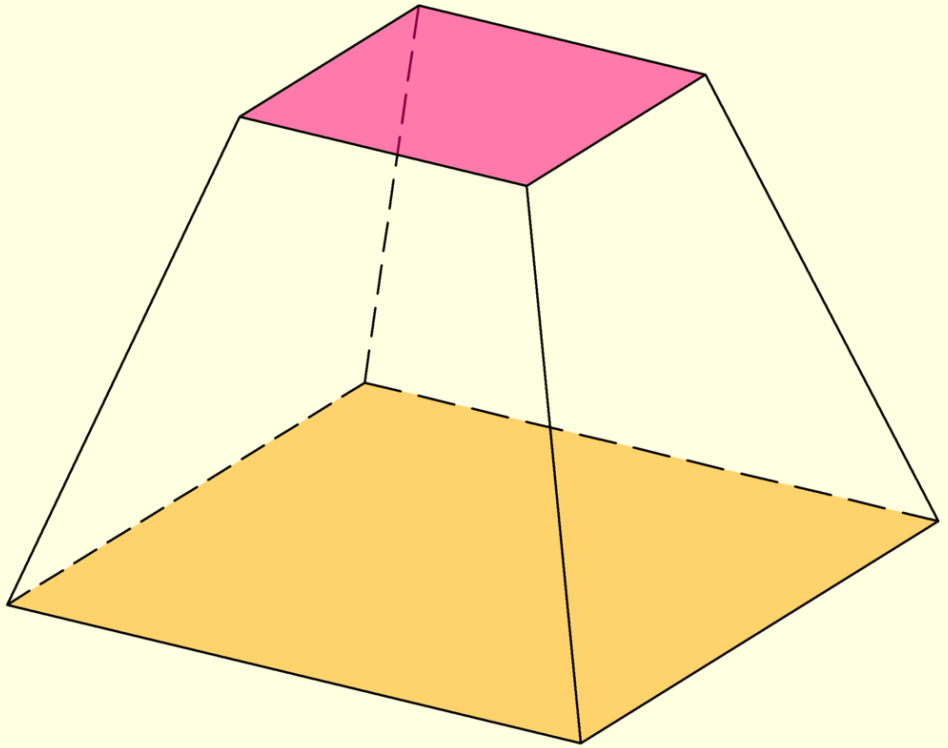
- Hit functions for axis-aligned boxes and triangles are in chapter 19.
- Your program must not crash on singularities, but may, for now, not correctly render.
- Implement the roll angle as described in section 9.8
- Your image is not yet expected to have shading or shadows.
- Object sizes:
  - Circle: radius=1 meter
  - Triangle: sides = 2, 2, and 1 meter.
  - Rectangular solid: 0.5 x 1.5 x 2 meters.

Perspective Projection



Frustum

In computer graphics, the viewing frustum is the three-dimensional region which is visible on the screen which is formed by a clipped pyramid.



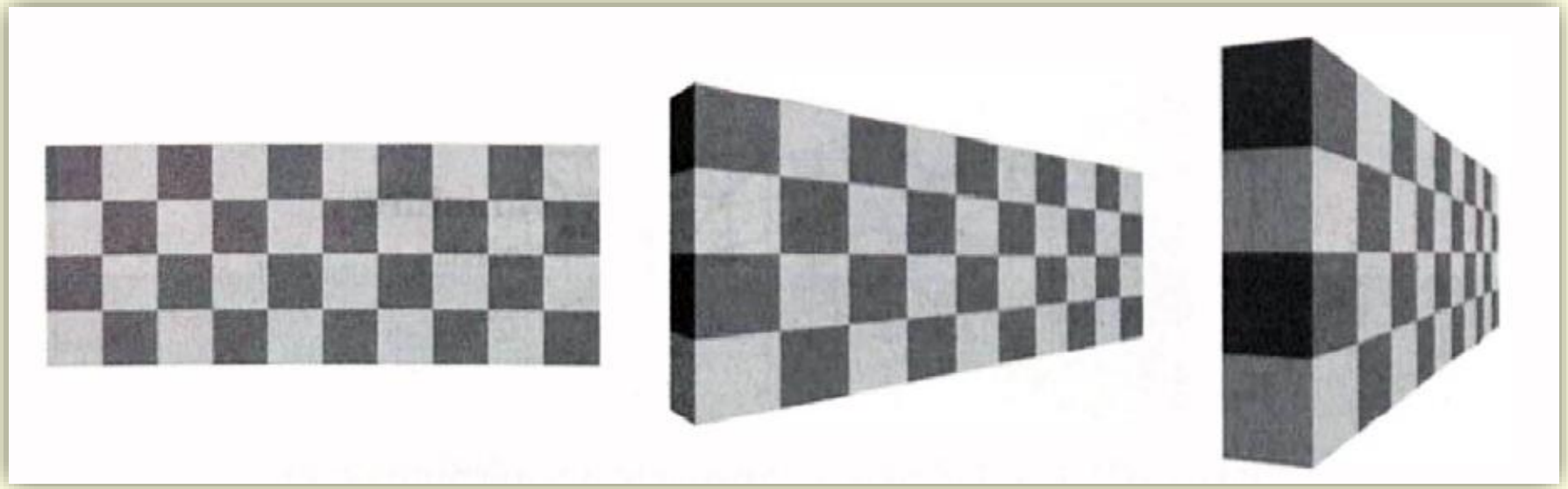
Properties of Perspective Projection

Property 1: The perspective projection of an object becomes smaller as the object gets farther away form the center of projection.

What are some non-obvious, and interesting effects of this?

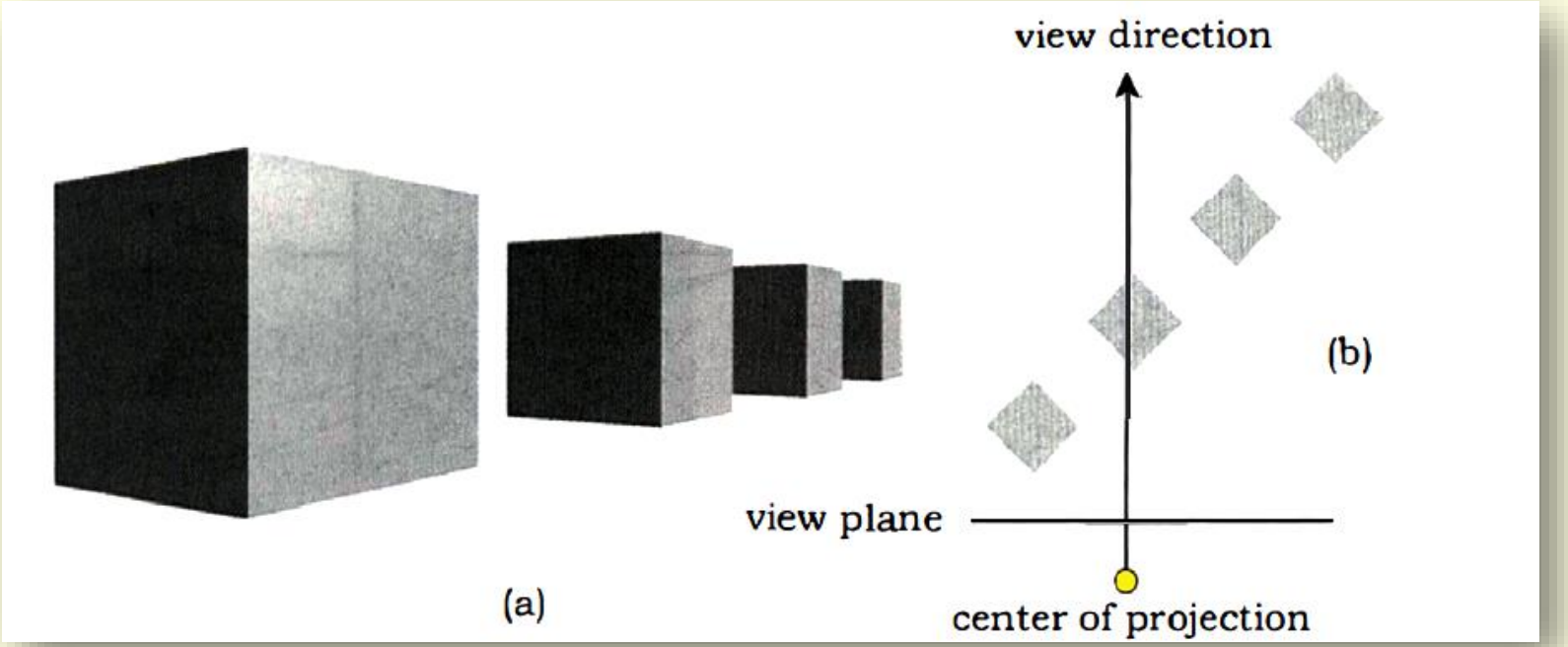
Properties of Perspective Projection

Property 2: As an object is rotated, its projected width becomes smaller. This is known as ***foreshortening***.



Properties of Perspective Projection

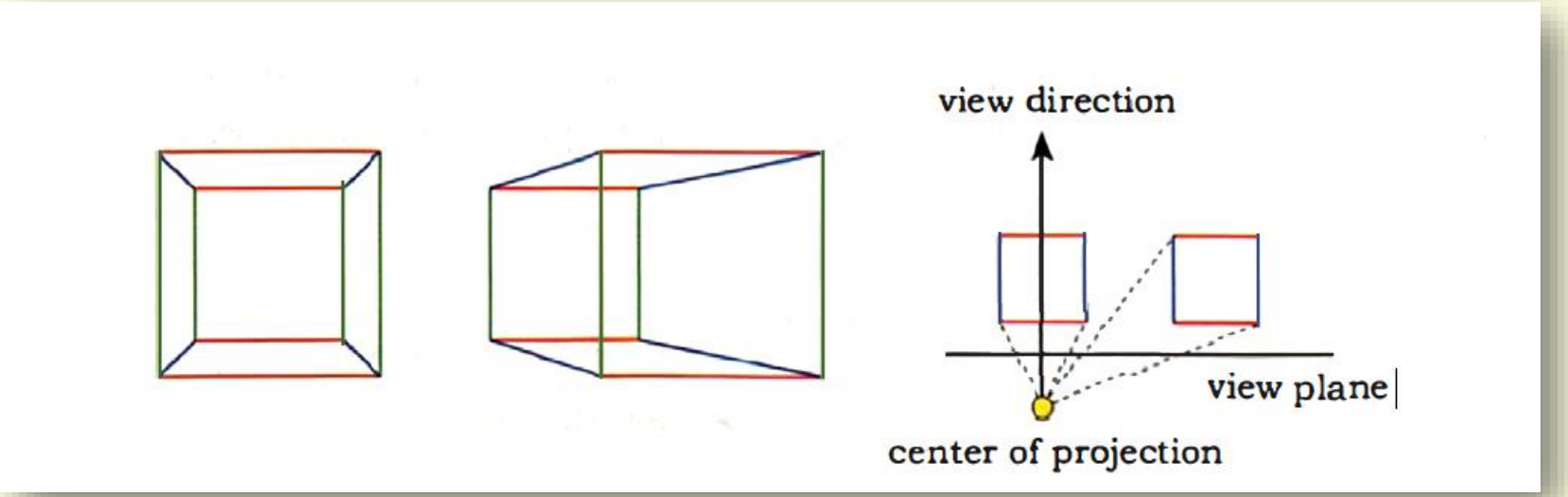
Property 3: Perspective projections preserve straight lines.



Properties of Perspective Projection

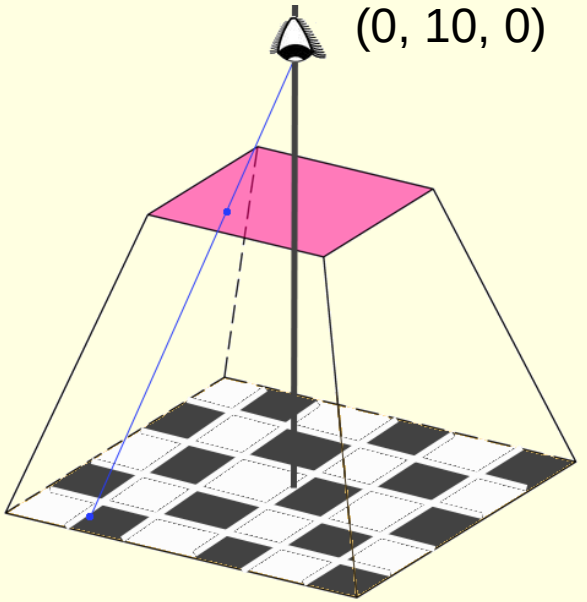
Property 4: Sets of parallel lines that are parallel to the view plane remain parallel when projected onto the view plane.

Property 5: Sets of parallel lines that are not parallel to the view plane converge to a vanishing point on the view plane.



Axis-Aligned Perspective Projection Straight down Y-Axis

- Eye: (0, 10, 0)
- One Point on View Plane: (x_{vp}, 5, z_{vp})



$$\overrightarrow{ray} = \overrightarrow{eye} + d(\overrightarrow{ViewPlanePt} - \overrightarrow{eye})$$

$$\begin{pmatrix} x_{hit} \\ 0 \\ z_{hit} \end{pmatrix} = \begin{pmatrix} 0 \\ 10 \\ 0 \end{pmatrix} + d \begin{pmatrix} x_{vp} - 0 \\ 5 - 10 \\ z_{vp} - 0 \end{pmatrix}$$

$$0 = 10 + d(5-10)$$

$$0 = y_{eye} + d(y_{vp} - y_{eye})$$

$$d = \frac{y_{eye}}{(y_{eye} - y_{vp})}$$

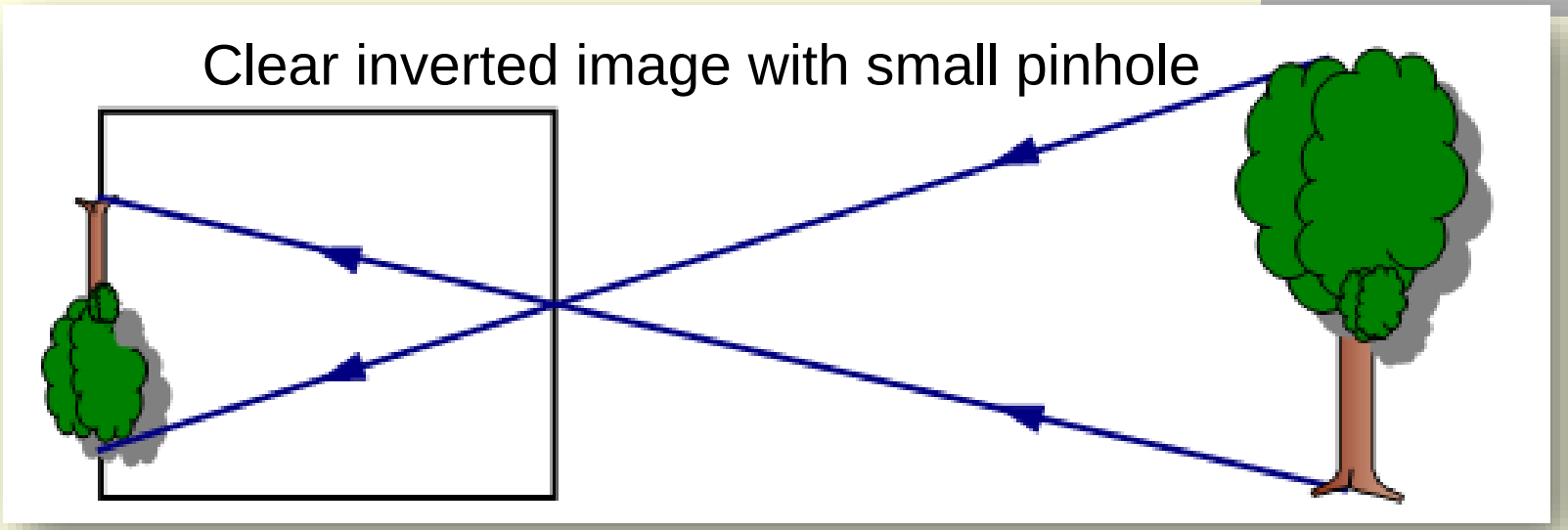
$$x_{hit} = x_{eye} + d(x_{vp} - x_{eye})$$

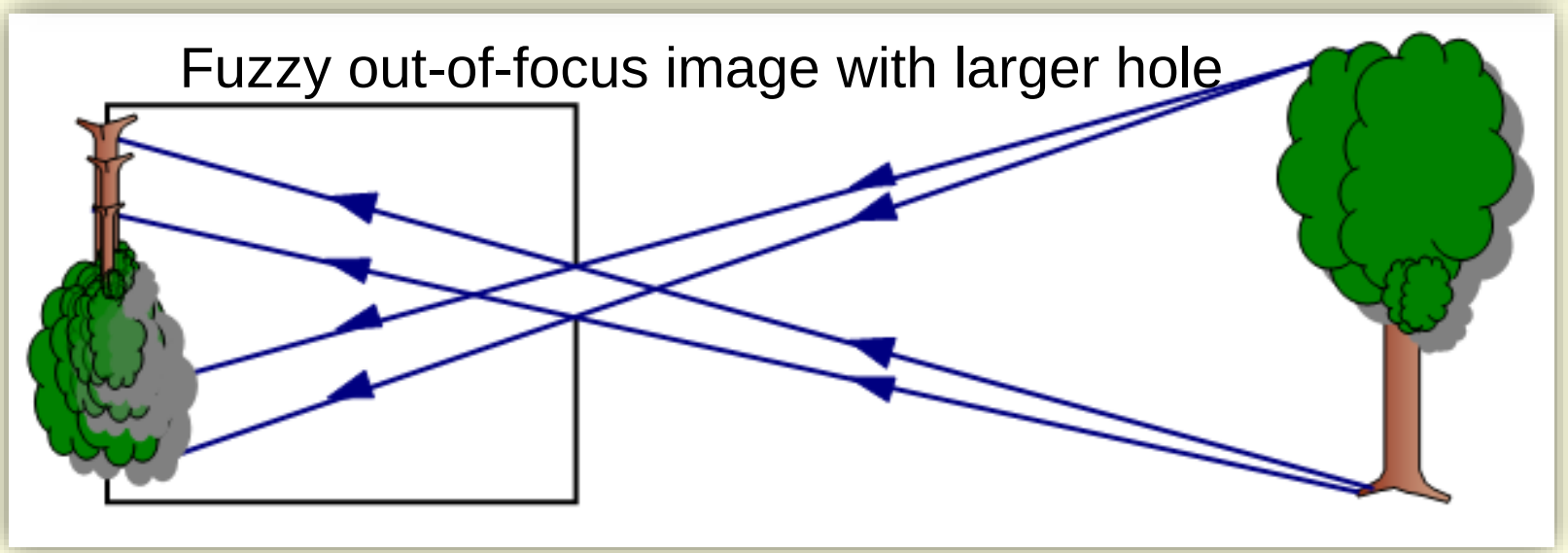
$$z_{hit} = z_{eye} + d(z_{vp} - z_{eye})$$

Chapter 9: A Practical Viewing System

- The ***virtual pinhole camera*** implements perspective viewing with the following features:
- An arbitrary eye point.
- An arbitrary view direction (The view plane is defined as being perpendicular to the view direction and centered on the ray from the eye point).
- An arbitrary orientation about the view direction.
- An arbitrary distance between the eye point and the view plane.

Physical Pinhole Camera





Lens Aperture

Aperture

Aperture ≠ Shutter

Shutter

Why color reflections?

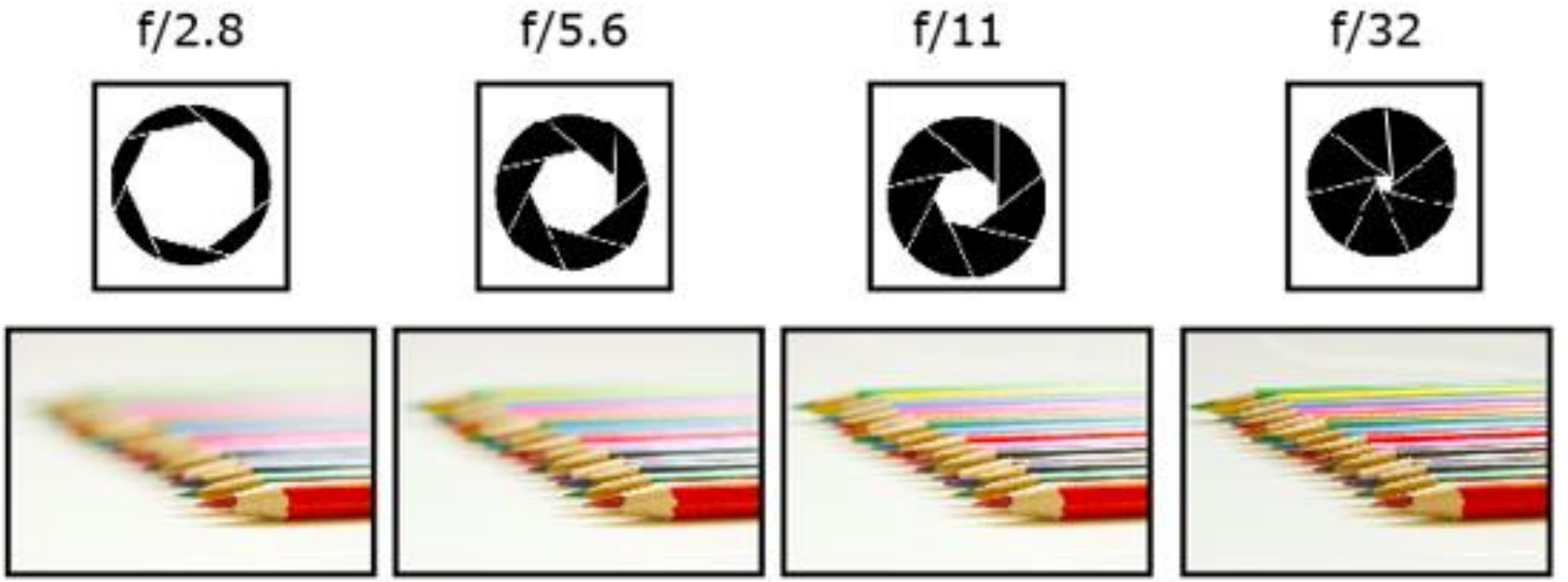
Large aperture lenses are expensive

Canon, Prime 50mm f/1.8 USM Lens: $125.00

Canon, Prime 50 mm f/1.4 USM Lens: $399.00

Canon, Prime 50 mm f/1.2 USM Lens: $1,549.00

Depth of Field

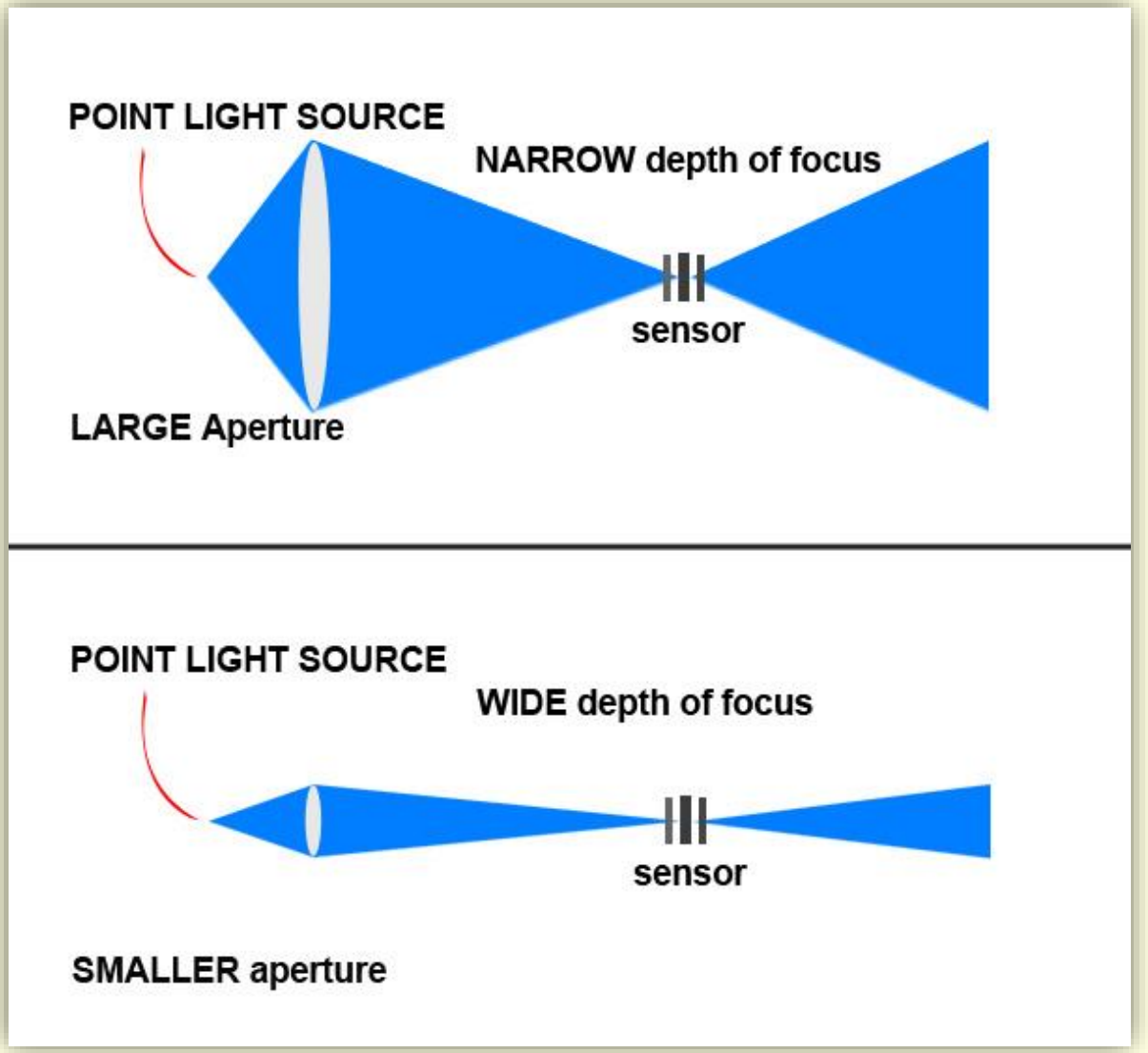


Circle of Confusion

Real lenses do not focus all rays perfectly.

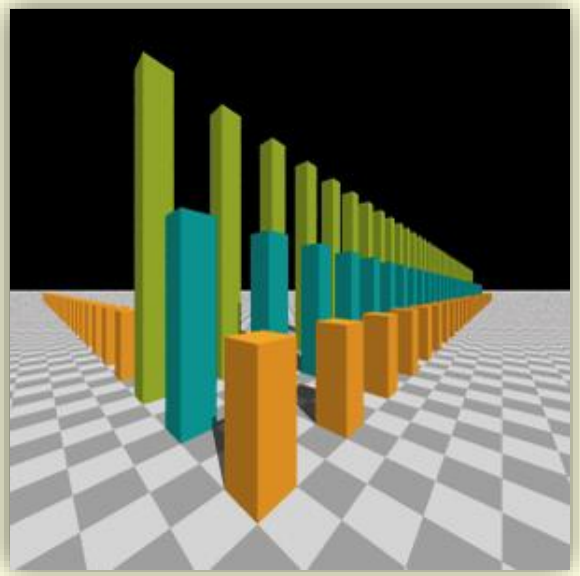
Thus, at best focus, a point is imaged as a spot rather than a point.

The smallest such spot that a lens can produce is often referred to as the *circle of least confusion*.

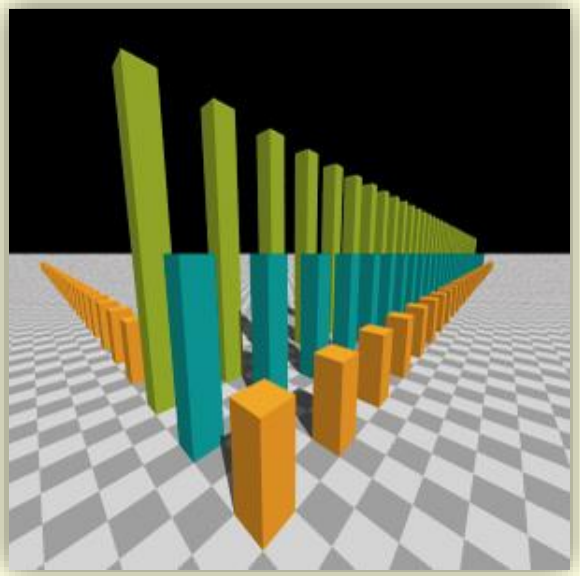


Perspective Views of Boxes (figure 9.10)

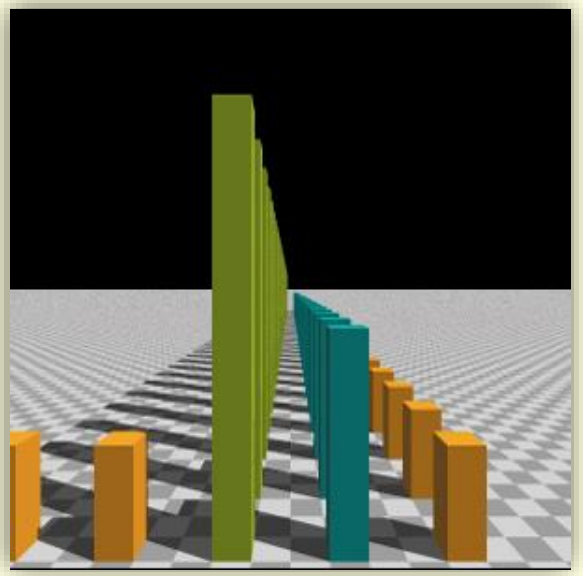
- How many vanishing points are there in each image?
- In each image, does the view direction point up or down or is it horizontal? How can you tell when it's horizontal?



a



b



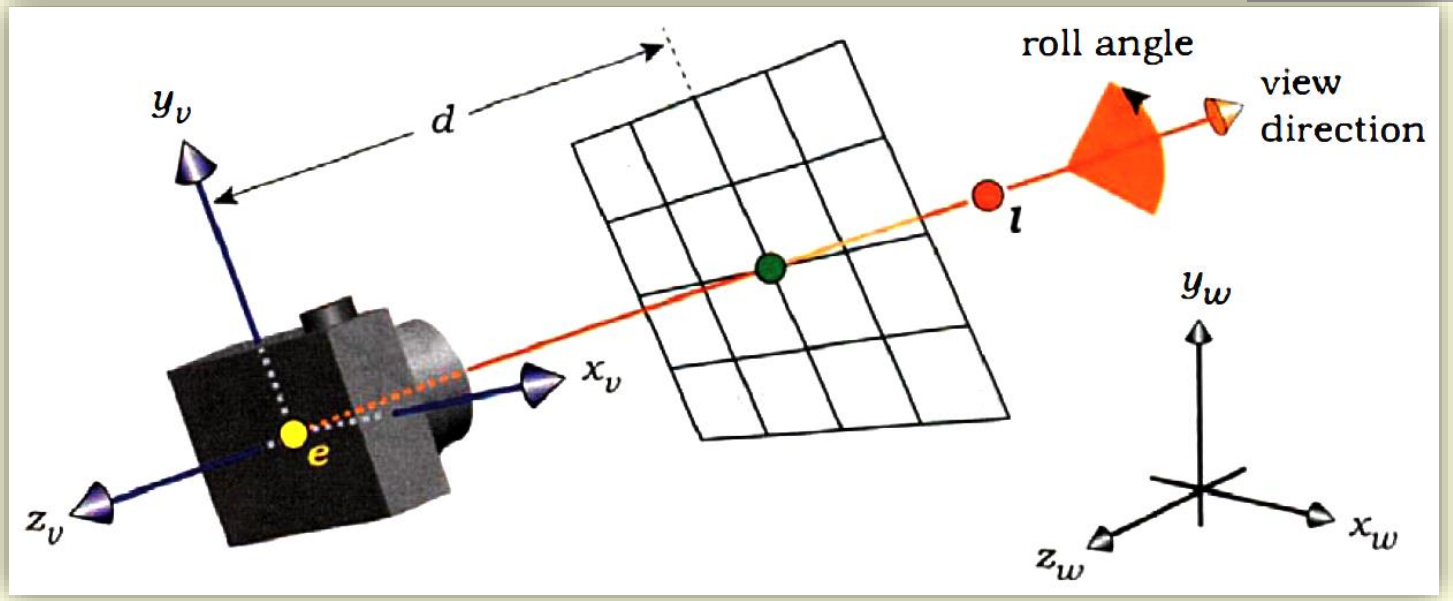
c

Quiz

1) What is an orthonormal basis (ONB)?
2) In the equation: $\vec{a} = \vec{b} \times \vec{c} \,/\left\| \vec{b} \times \vec{c} \right\|$

   $\vec{a}$, $\vec{b}$ and $\vec{c}$ are vectors. This means they have both magnitude and direction. What can be said about the magnitude and direction of $\vec{a}$?

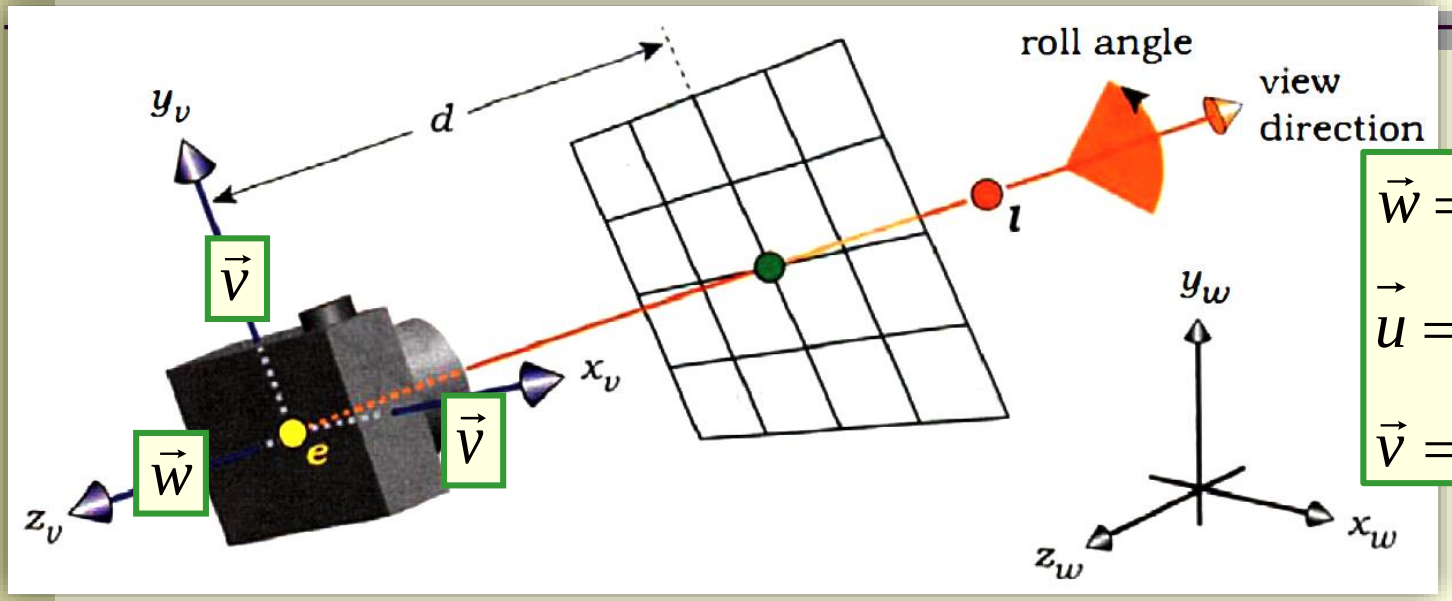
Virtual Pinhole-Camera Viewing System



User Input:

- The eye point, e.
- The look-at point, l.
- The up vector, up.
- The view-plane distance d.

Primary-Ray Calculation



$$\vec{w} = (e - l) / \|e - l\|$$

$$\vec{u} = \overrightarrow{up} \times \vec{w} / \left\| \overrightarrow{up} \times \vec{w} \right\|$$

$$\vec{v} = \vec{w} \times \vec{u}$$

The (x_v, y_v) coordinates of a sample point p on the pixel in row r and column c are:

$$x_v = s(c - h_{res}/2 + p_x)$$

$$y_v = s(r - v_{res}/2 + p_y)$$

The primary-ray direction d is: $\vec{d} = x_v\vec{u} + y_v\vec{v} - d\vec{w}$

Pinhole Camera: Fields

A pinhole camera is simple: all rays pass through a single point (unlike a lens where rays pass through the lens surface).

Camera Object:

- Class Fields:
  - Vector3 position
  - Vector3 viewDirection
  - Vector3 upVector (0,1,0)
  - double viewDistance

    or

    double angleOfView (In the vertical view plane.
  - double nearClipDistance
  - double farClipDistance

Pinhole Camera: Render Loop Algorithm

1) Set all pixels to the background color.
2) Loop for i=0 through numberOfRays.
3) Sample a row and column in screen pixel space,
4) Calculate a primary ray from the camera (eye) location through some location in the chosen pixel.
5) Calculate minimum and maximum time along that ray within the frustum.
6) Set the minimumHitTime to the max time of the frustum.
7) Loop through each object in the scene and call its hit function, updating minimumHitTime.
8) After looping through each object, if minimumHitTime is less than the max time of the frustum, render the pixel (if multiple rays are cast through the same pixel, merge before render).

Pinhole Camera Step 3: Coordinate System

In this project, we are working with 3 coordinate systems:

- Doubles (x, y, z): a point in the global world coordinate system.
- Integers (col, row): a point in screen pixels.
- Doubles $(viewX, viewY)$: a point in view plane coordinates.

To avoid distortion, choose view bounds with the same aspect ration as the pixel window. For example, if the pixel window is 1000x500 and the viewX range is 10 meters, then the viewY range should be 5 meters.

Pinhole Camera Step 3: View plane Points

Given pixel (col, row), the view plane coordinates ($viewX$, $viewY$) of a random point within a pixel are given by:

$$viewX = pixelSize * (col + r_1) + viewMinX$$
$$viewY = pixelSize * (totalRows - row - r_2) + viewMinY$$

Where

$$pixelSize = \frac{viewMaxX - viewMinX}{totalCol}$$

r_1 and r_2 are random doubles in range [0,1).

Note: to avoid distortion choose the view bounds so that,

$$\frac{viewMaxX - viewMinX}{totalCol} = \frac{viewMaxY - viewMinY}{totalRow}$$

Pinhole Camera Step 3: Ray Direction

Each primary ray starts at the eye.
Each primary ray's direction is:

$$\vec{\boldsymbol{d}} = viewX(\vec{\boldsymbol{u}}) + viewY(\vec{\boldsymbol{v}}) - viewPlaneDistance(\vec{\boldsymbol{w}})$$

Normalized: $\hat{\boldsymbol{d}} = \frac{\vec{\boldsymbol{d}}}{\|\vec{\boldsymbol{d}}\|}$

Where:

$$\vec{\boldsymbol{w}} = \left(\overrightarrow{\boldsymbol{eye}} - \overrightarrow{\boldsymbol{viewDirection}}\right) / \left\|\overrightarrow{\boldsymbol{eye}} - \overrightarrow{\boldsymbol{viewDirection}}\right\|$$

$$\vec{\boldsymbol{u}} = \overrightarrow{\boldsymbol{up}} \times \vec{\boldsymbol{w}} / \|\overrightarrow{\boldsymbol{up}} \times \vec{\boldsymbol{w}}\|$$

$$\vec{\boldsymbol{v}} = \vec{\boldsymbol{w}} \times \vec{\boldsymbol{u}}$$

$$\overrightarrow{\boldsymbol{up}} = (0,1,0)$$

Pinhole Camera Step 3: Efficiency Note

- Given an equation of a normalized vector such as:

$$\vec{\boldsymbol{u}} = \overrightarrow{\boldsymbol{up}} \times \vec{\boldsymbol{w}} / \| \overrightarrow{\boldsymbol{up}} \times \vec{\boldsymbol{w}} \|$$

- The same calculation appears in the numerator and the denominator. However, do not calculate the same thing twice. Rather first calculate the non-normalized vector. Then call the normalize function of Vector3D.

Pinhole Camera Step 6: Hit Function (1 of 2)

The hit function has the form:

- **`hit(Ray& ray, double& tmin, ShadeRec& sr)`**
- When an object's hit function returns true, if the returned **`tmin`** is less than the current ray's **`minimumHitTime`**, AND the **`tmin`** is greater than the frustum minimum time, then:
  1) Update **`minimumHitTime`** to equal the object's **`tmin`**.
  2) Save a reference to the object whose hit function returned true. Overwrite any reference saved with a larger **`tmin`**. Only the nearest object is important.
  3) Save the DATA in the returned **`ShadeRec`** (or save the pointer if each object has a unique **`ShadeRec`**). **`ShadeRec`** is used to color (shade) the point.

Pinhole Camera Step 6: Hit Function (2 of 2)

Note 1: In this project, the only field of **`ShadeRec`** we care about is **`hit_point`** and we only care about it for the checkerboard object. All other objects have only a single color and we are not using lighting or other shading effects).

Note 2: The value of **`t`** in **`ShadeRec`** is the same as the value returned in **`tmin`**.

Pinhole Camera Step 7: Render

- After looping through each object, if **`minimumHitTime`** is less than the max time of the frustum, convert the view plane point, p, to screen coordinates and render the point.
- To render a point, a color is needed.
- I suggest adding to Sphere.cpp, Plane.cpp, Rectangle.cpp and Checkerboard.cpp the function:

```
getColor(double x, double y, double z)
```

- For now, all our objects, except for the checkerboard have only one color, so for those objects getColor will always return the same color regardless of the values of x, y and z. If each Shpere.cpp object has an instance field, color, then each instance of Sphere.cpp can have a different color.

Equation of a Checkerboard pseudo code

```
//Infinite checkerboard of unit squares in x-z plane.
public Color getColor(double x, y, z)
{
  if (abs((floor(x)))%2 == abs((floor(z)))%2)
        return Color.BLACK;
  return Color.WHITE;
}
```

Note: This is NOT a hit function. It is called after a hit has been found and is given the values **x**, **y** and **z** of the hit from **ShadeRec.hit_point**

For the hit function, make Checkerboard extend the book's plane object and use the plane's hit function.