

(Correction to glOrtho discussion below.)

Current Assignments:

Robot arm, fingers on planet project 3.

Homework 4: Active study of building an axis rotation matrix (Due Fri.)

Homework 5: Active study of object versus coordinate transformations.

Midterm 2: Monday: Cover up to basic parallel and perspective projection (projects, homeworks, today's lecture.) Open book+one sheet of notes as before.

3d-Viewing: Chapter 7.

“Point the Camera” means transform world coordinates into **viewing coordinates**

“Choose lens, film angle, film size, etc.” means transform viewing coordinate by a **projection** onto a film plane.

What is a view camera? Do a Google search for “Calumet view camera sales” and see!

Perspective projections (and **parallel projections**) the first technique for making objects appear to be 3-dimensional.

Second technique: **Identifying Visible Lines and Surfaces:** and only rasterize them.

Modern systems use **projection** to transform the **view volume** of world-coordinate space into a *3-DIM* **normalized view volume cube**.

Parallel to z-axis viewing of the view volume contents yields a perspective (or other) image.

This viewing is simply “ignore the z coordinate” of the transformation of an object point into normalized view coordinates (x, y, z) .

When the viewing direction is FROM the positive z-axis towards the negative z-axis as it is in OpenGL, the value of z varies INVERSELY with the DISTANCE of the object point to the viewer.

So, the z-values calculated by projection are used within the **depth buffer** to implement **hidden surface removal**.

Idea: for each pixel position, render only the color from the “fragment” for which the z coordinate is minimum? maximum? (You figure out which!)

The **depth buffer** is today built into the graphics card, alongside the **color or frame buffer**.

Typically, the normalized view volume implements **clipping**.

Other techniques:

Depth Queuing: more distant objects have less intense color.

Surface Rendering: use computational models for lighting conditions and for how surfaces reflect or scatter light toward the eye.

Exploded or Cutaway Views: Typical of engineering, scientific, medical, etc. illustrations.

Display of hidden lines or surfaces behind the visible surfaces. Blender lets you choose between wire frame and surface rendered viewing when you build a scene.

User controlled or animated rotation. You will use that in Blender.

Stereoscopic display devices: Just calculate two images, one for each eye, each the same way as for one eye.

The viewing transformation is typically determined by

1. The **view point** $\mathbf{P}_0 = (x_0, y_0, z_0)$ aka **viewing position** aka **eye position** aka **camera position**.
2. The **view plane normal**: direction from which the eye is going to look.

It is the vector $\mathbf{P}_0 - \mathbf{P}_{\text{ref}}$ FROM
the **view reference** (“look at”) point \mathbf{P}_{ref} TO \mathbf{P}_0 .

3. The **up vector**: Some vector that will be transformed to into the y viewing direction.

(The projection will then transform the y direction into the UP direction on the display.)

Calculation:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and } \mathbf{R} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

T translates the view point to the origin. **IN OTHER WORDS**, transforms coordinate systems by translation so the new origin is the view point.

R rotates space so the view plane normal is aligned along the z axis, and the up vector is in the yz plane. **IN OTHER WORDS**, transform coordinate systems so the new z axis is along the view plane normal, etc.

How to get **R**? **GIVENS**:

- **N**: View plane normal calculated by $(x_0 - x_{\text{ref}}, y_0 - y_{\text{ref}}, z_0 - z_{\text{ref}})$.
- **V**: Up vector.

Calculate:

$$\mathbf{N} = \frac{\mathbf{N}}{|\mathbf{N}|} = (n_x, n_y, n_z)^t$$

$$\mathbf{u} = \frac{\mathbf{V} \times \mathbf{n}}{|\mathbf{V}|} = (u_x, u_y, u_z)^t$$

- (1) only division by $|\mathbf{V}|$ is necessary, no need to calculate length of $\mathbf{V} \times \mathbf{n}$.
- (2) **FAILURE IF** user gives parallel **N** and **V**!

$$\mathbf{n} = \frac{\mathbf{N}}{|\mathbf{N}|} = (n_x, n_y, n_z)^t ; \mathbf{u} = \frac{\mathbf{V} \times \mathbf{n}}{|\mathbf{V}|} = (u_x, u_y, u_z)^t ; \text{ and } \mathbf{v} = \mathbf{n} \times \mathbf{u}$$

$$\mathbf{R}^t = \begin{bmatrix} & & \\ \mathbf{u} & \mathbf{v} & \mathbf{n} \\ 0 & 0 & 1 \end{bmatrix}$$

1. rotates x -axis into \mathbf{u}
2. rotates y -axis into \mathbf{v}
3. rotates z -axis into \mathbf{n} .

\mathbf{SO} , $\mathbf{R} = (\mathbf{R}^t)^{-1} = (\mathbf{R}^{-1})^{-1}$ does the job. See it written out in the textbook.

We must carefully check that the right-hand-rule definition of the two cross products really makes \mathbf{v} point in the right direction. That is, make $\mathbf{v} \cdot \mathbf{V} > 0$

```
OpenGL: glMatrixMode(GL_MODELVIEW);  
glLoadIdentity( );  
gluLookAt(x0, y0, z0, xref, yref, zref, Vx, Vy, Vz);
```

sets viewing origin (“look-from point”) to (x_0, y_0, z_0)

sets reference position (“look-at point”) to $(x_{ref}, y_{ref}, z_{ref})$

sets up vector to (V_x, V_y, V_z)

and multiplies the viewing transformation matrix so defined into the current `MODELVIEW` matrix (which was `I`).

Exercise: What is the correct modelview matrix when for the default

OpenGL viewing parameters $\mathbf{P}_0 = (0, 0, 0)$, $\mathbf{P}_{ref} = (0, 0, -1)$ and $\mathbf{V} = (0, 1, 0)$?

Viewing Transformation: Transforms world coordinates to **viewing coordinates**.

Determines from which direction we want to look, and which way is up.

The user-specified **view plane normal** is transformed into the positive z-direction of viewing coordinates.

`gluLookAt (. . .);` is handy!

Projection Transformation: Transforms viewing coordinates to **normalized viewing coordinates**.

(Actually two stages: (1) Linear transformation of 4-component homogeneous coordinates (x, y, z, h) , then (2) **perspective division** that transforms (x, y, z, h) INTO $(\frac{x}{h}, \frac{y}{h}, \frac{z}{h}, .)$)

The normalized viewing coordinates should be designed so graphics outside the **normalized view volume** ($2 \times 2 \times 2$ cube centered at $(0, 0, 0)$ for OpenGL) are **clipped** away.

The 2-d projection is then computed by *throwing away* the z-coordinate.

But before that, the z-coordinate is used for hidden line/surface removal.

In your homework, the perspective division was the result of a projection:

$$\begin{bmatrix} x' \\ y' \\ z' \\ h' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ h(=1) \end{bmatrix}$$

NOTE THE 4-th ROW! In plain algebra,

$$x' = x; \quad y' = y; \quad z' = z$$

and

$$h' = z \text{ NOT the original } h(=1)$$

$$\text{so } x'/h' = x/z \text{ and so } y'/h' = y/z$$

Now let's do two simple projections:

Orthogonal projection onto the xy plane:

$$x' = x \text{ and } y' = y \text{ and } z' = 0$$

really simple, eh?

The OpenGL default is equivalent to:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity( );
glOrtho( xmin, xmax, ymin, ymax, dnear, dfar );
```

with

```
xmin = -1.0;    xmax = 1.0; //x-coords of clipping window
ymin = -1.0;    ymax = 1.0; //y-coords of clipping window
dnear = -1.0;   dfar  = 1.0; //NEAR and FAR DISTANCES..
//NEAR clipping plane is z=(-dnear)= 1.0
//FAR  clipping plane is z=(-dfar)= -1.0
gluOrtho2D(xmin, xmax, ymin, ymax);
//is equivalent to:
glOrtho( xmin, xmax, ymin, ymax, -1.0, 1.0 );
```

Let's implement `g1Ortho` — Use the math of your HW3 programming exercise:

Given (x, y, z) (viewing coordinates), how should x be transformed?

$$x \rightarrow \left(x - \frac{xwmin + xwmax}{2} \right)$$

is the signed distance of x from middle of the desired view volume.

$$\left(x - \frac{xwmin + xwmax}{2} \right) \rightarrow 2 \cdot \frac{\left(x - \frac{xwmin + xwmax}{2} \right)}{xwmax - xwmin}$$

normalizes it to the $[-1, +1]$ range (the length of this range is 2.0.)

Ditto for Y . BUT for Z , use `-dnear` and `-dfar`

So, larger z -buffer values correspond to larger distances from the viewer.

From the RedBook, `glOrtho(l, r, t, n, f)` generates

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

when $n = -1$ and $f = +1$, the lower-right-hand 2×2 matrix

$$\begin{bmatrix} \frac{-2}{f-n} & \frac{f+n}{f-n} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{-2}{1-(-1)} & \frac{0}{1-(-1)} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} z \\ 1 \end{bmatrix} = \begin{bmatrix} -z \\ 1 \end{bmatrix}$$

next, when the (x', y', z', h') from the above used to transform $(x, y, z, 1)$ is perspectivevly divided at last, the result is

$$\frac{-z}{1} = -z$$

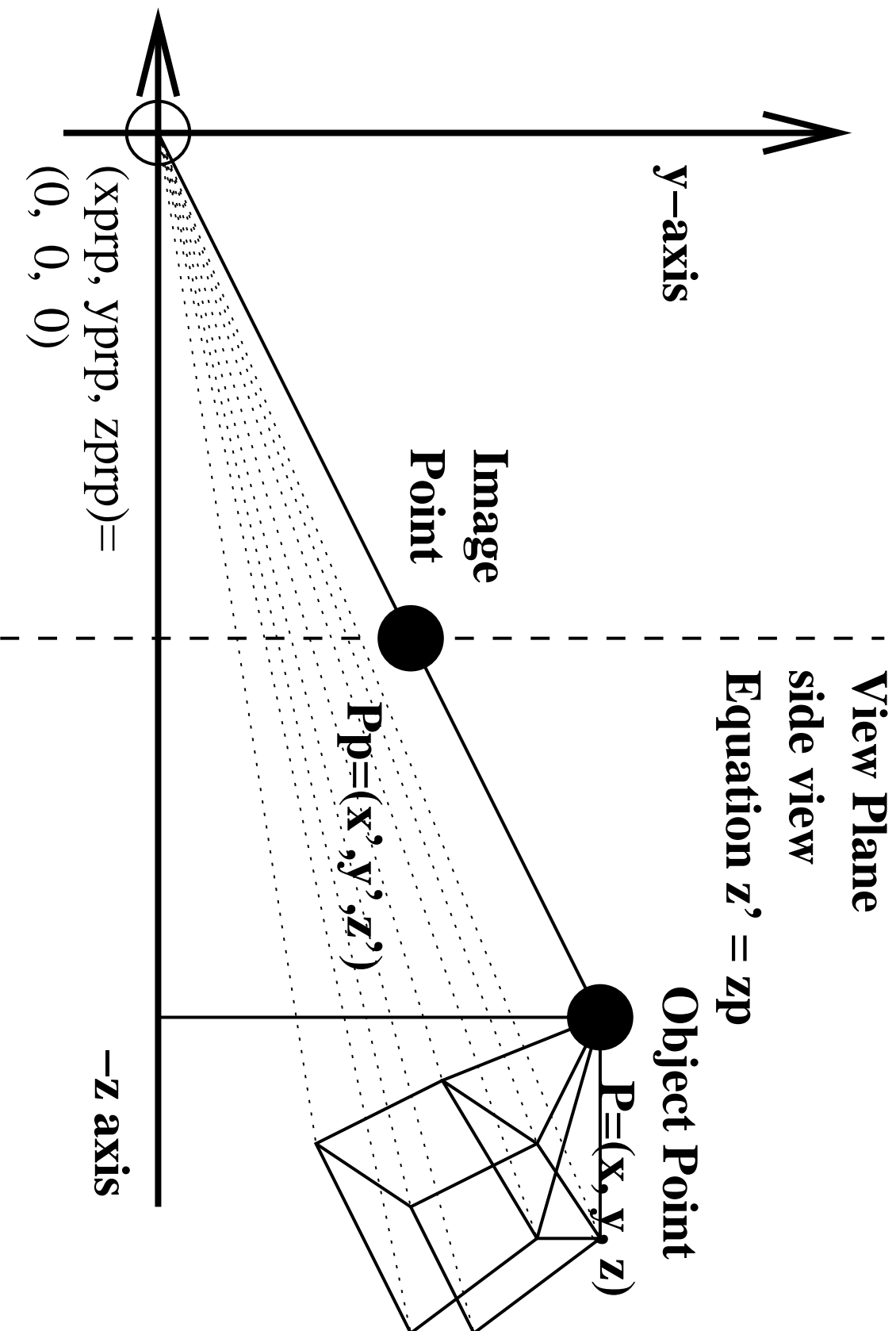
So, points in the near clipping plane $z = +1$ get transformed to normalized device coordinates with $z_{\text{ndc}} = -1$ (smallest Z value.)

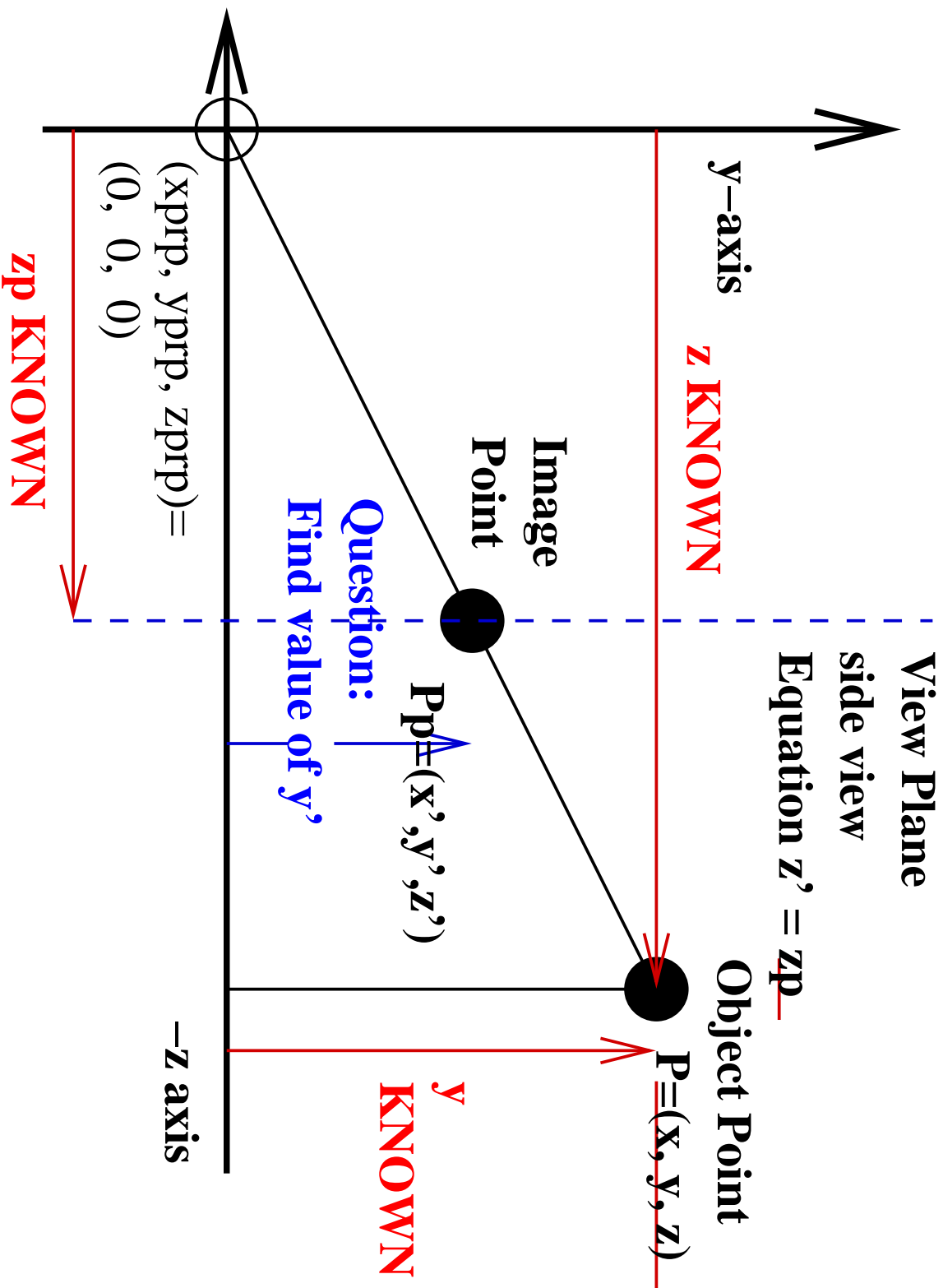
Perspective projection: Suppose the eye is at $(0, 0, 0)$ and the projection plane is given by the equation $z = z_{\text{proj}}$.

Where in the $z = z_{\text{proj}}$ plane do we draw the intersection of the **RAY** from the eye $(0, 0, 0)$ to the point (x, y, z) ? What are the x and y coordinates of the intersection?

It is easy to figure it out with similar triangles!

Do it twice; once for x and once for y .





Use the two SIMILAR (same shape, different size) triangles with common vertex $(0,0,0)$.

SIMILAR FIGURES: corresponding distance ratios are equal.

Get equation:

$$\frac{|\text{side opposite base (y-side)}|}{|\text{triangle base (z-side)}|} = \frac{y'(\text{UNknown})}{z_p(\text{known})} = \frac{y(\text{known})}{z(\text{known})}$$

Solve it for y' :

$$y'(\text{UNknown}) = \frac{y(\text{known})}{z(\text{known})} z_p(\text{known})$$

and do the analogous analysis and equation for x_p , x and x' .

Facts about perspective projection:

1. Straight lines are drawn as straight lines.
2. Parallel (straight) lines parallel to the **VIEWPLANE** are drawn as parallel lines.
3. Parallel lines **NOT** parallel to the viewplane are drawn as lines that meet. Where the images of a bunch of mutually parallel lines meet is called a **vanishing point**.

The typical bunches of parallel lines in photos and (pre-modern but post-middle age) paintings of buildings are the lines going in the principal horizontal directions of the objects.

So “two point” perspective is the most common—the vertical lines are parallel to the viewplane so they are drawn parallel.

In the mathematics of projective geometry, parallel projection is (merely) a special case of perspective projection!

