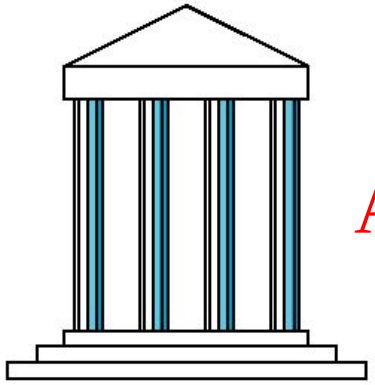
CSCI-396
Jeff Bush

Projection

Planar Geometric Projections

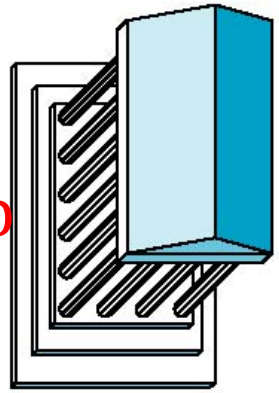
- Standard projections project onto a plane
- The projections all meet at the ***center of projection (COP)***
  - The COP can be at infinity causing all projections to be parallel and then there is a direction of projection (DOP)
  - Difference between ***perspective*** and ***parallel*** views
- Both types preserve lines but not necessarily angles
- Non-planar projections are needed for special applications

Classical Projections

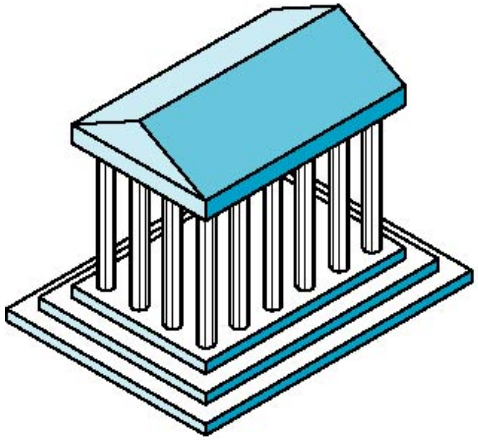


Front elevation

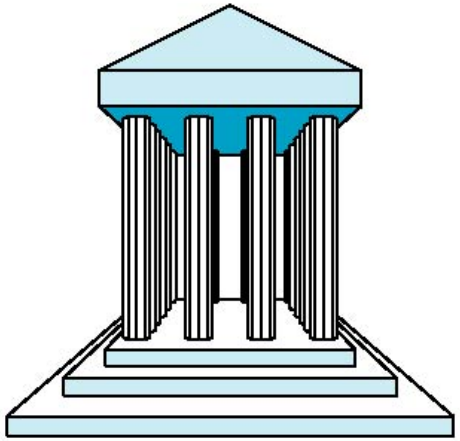
Elevation oblique



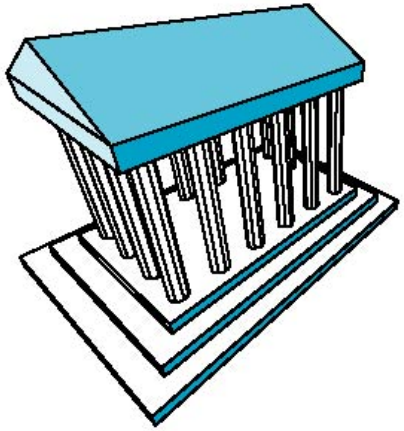
Plan oblique



Isometric



One-point perspective

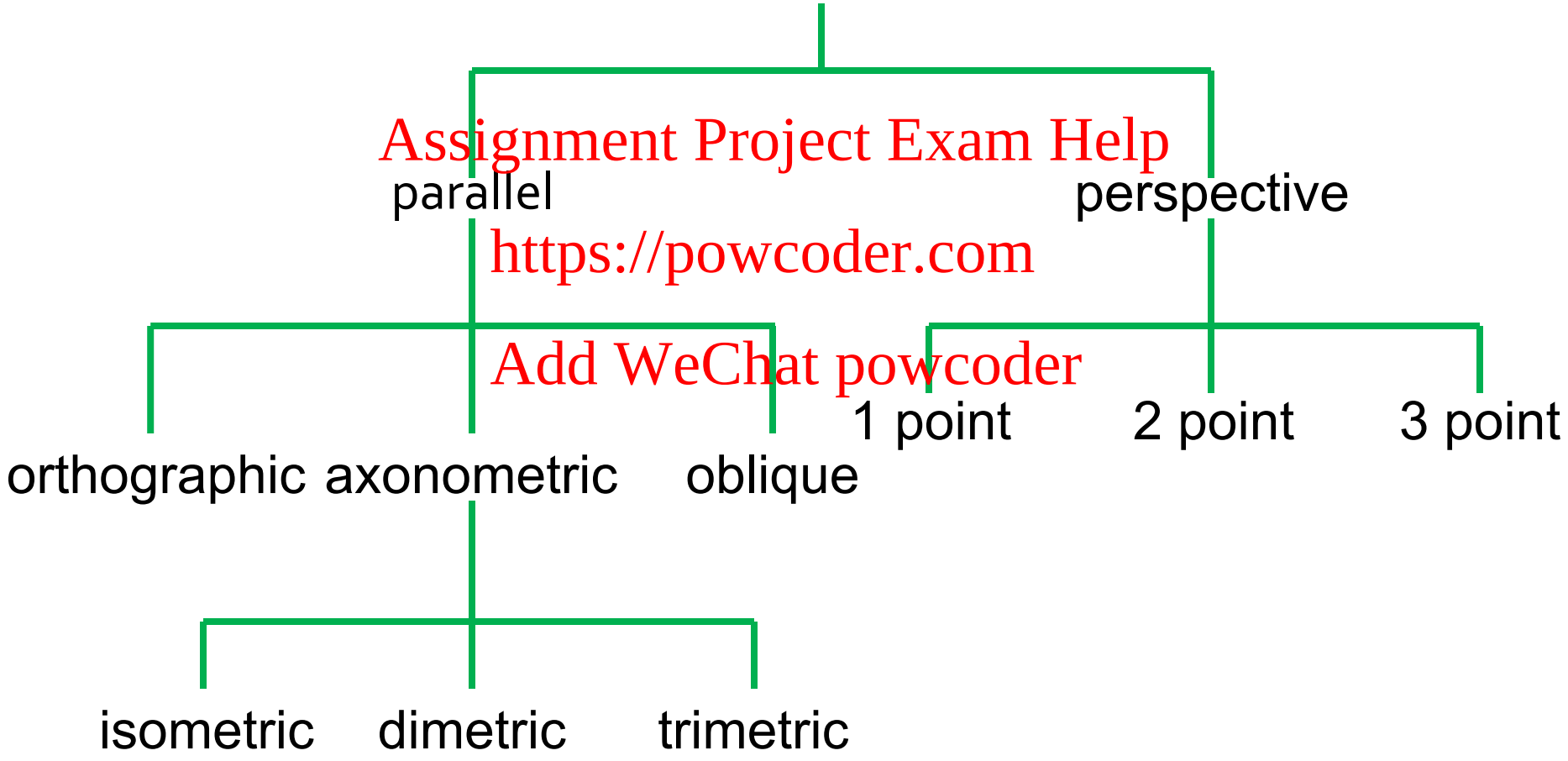


Three-point perspective

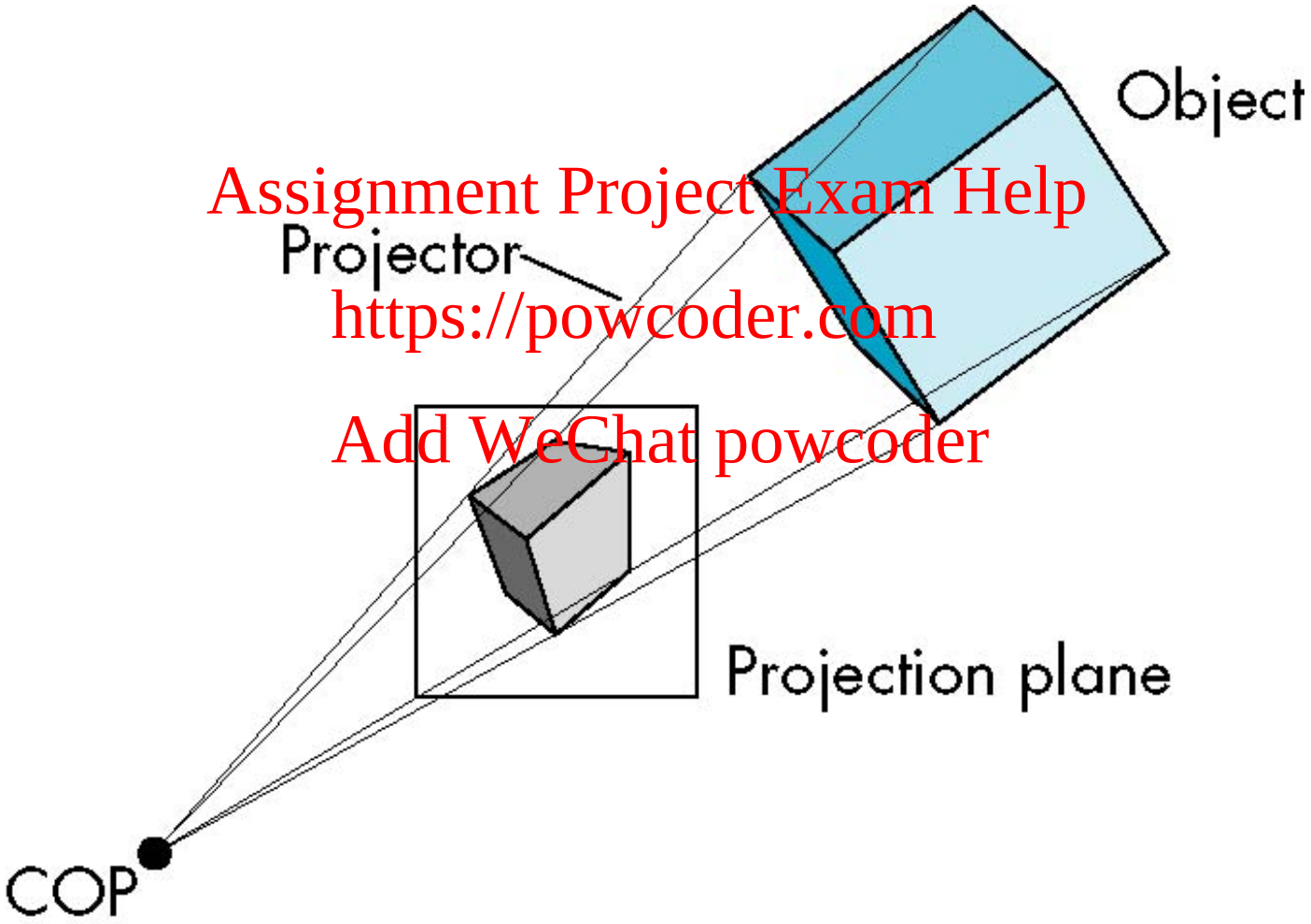
Perspective vs Parallel

- GPU treats all projections the same and implements them with a single pipeline
- Classical viewing developed different techniques for drawing each type of project
- Even "fundamental" distinction between parallel and perspective is mathematically the same
  - Just with a COP at infinity

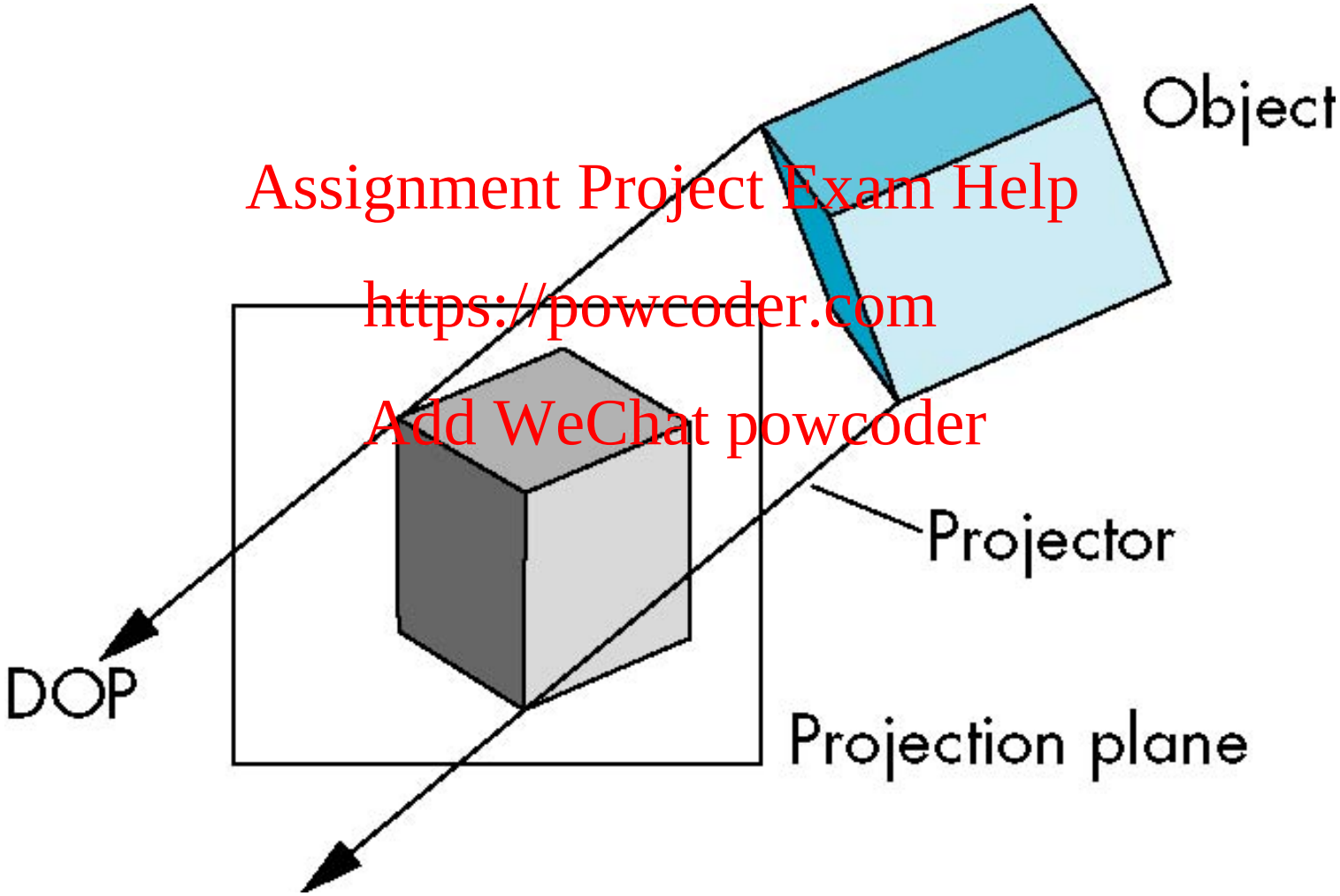
Planar Geometric Projections



Perspective Projection

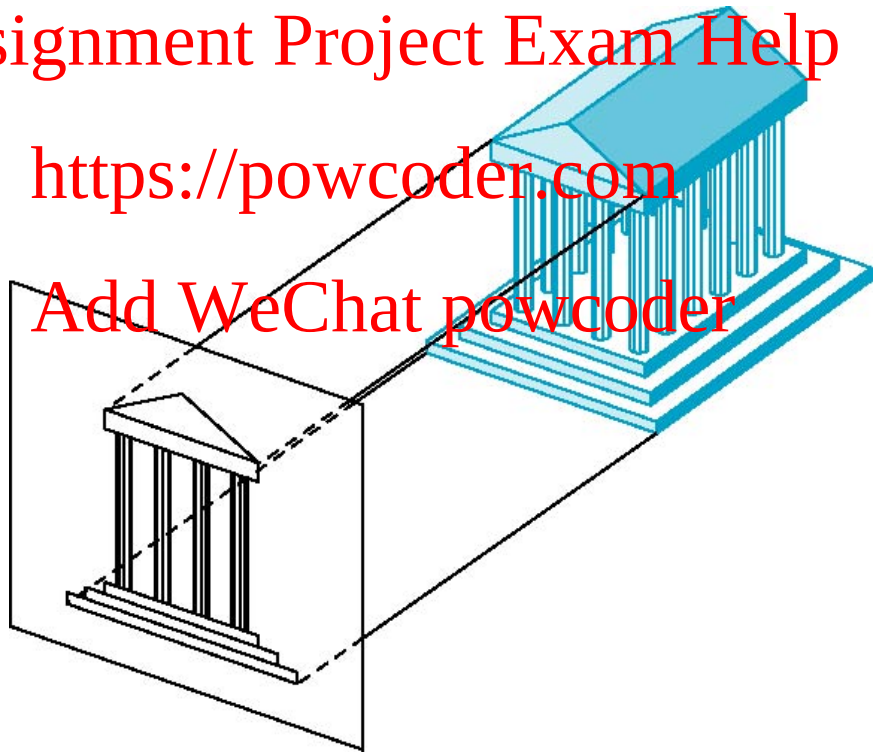


Parallel Projection



Orthographic Projection

Projectors are orthogonal to projection surface

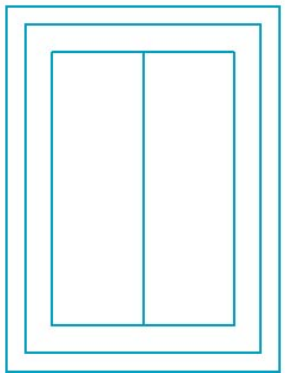


Multiview Orthographic Projection

- Projection plane parallel to principal face
- Usually form front, top, side views

isometric (not part of mv orthographic)

In CAD and architectural design often display three multiviews plus isometric

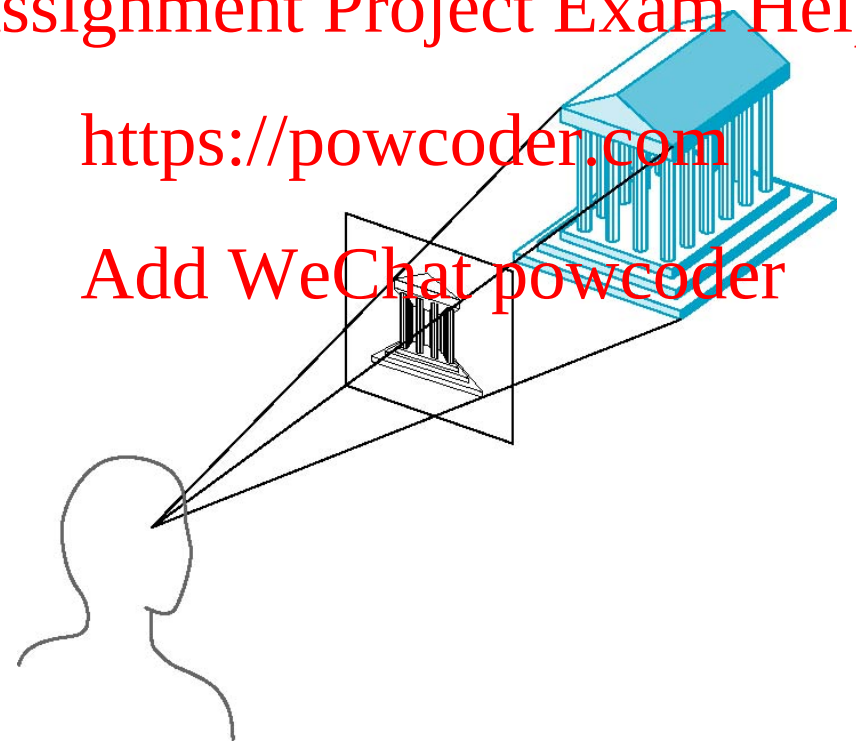


Advantages/Disadvantages

- Preserves both distances and angles
  - Shapes preserved
  - Can be used for measurements
    - Building plans
    - Manuals
- Cannot see what object really looks like because many surfaces hidden from view
  - Why the isometric view is often added

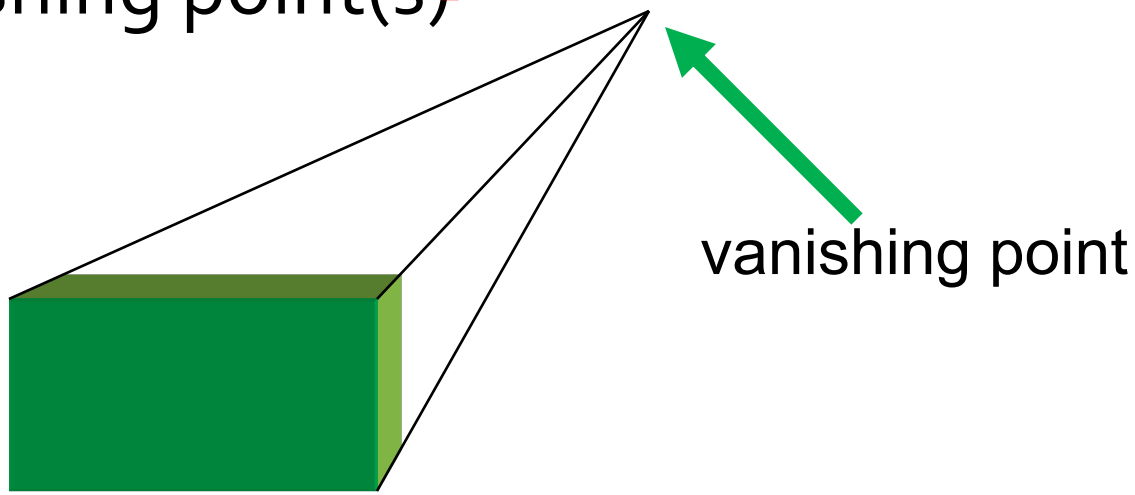
Perspective Projection

Projectors converge at center of projection



Vanishing Points

- Actual parallel lines within objects converge at a single point in the projection (the *vanishing point*)
- Drawing simple perspectives by hand uses these vanishing point(s)



Three-Point Perspective

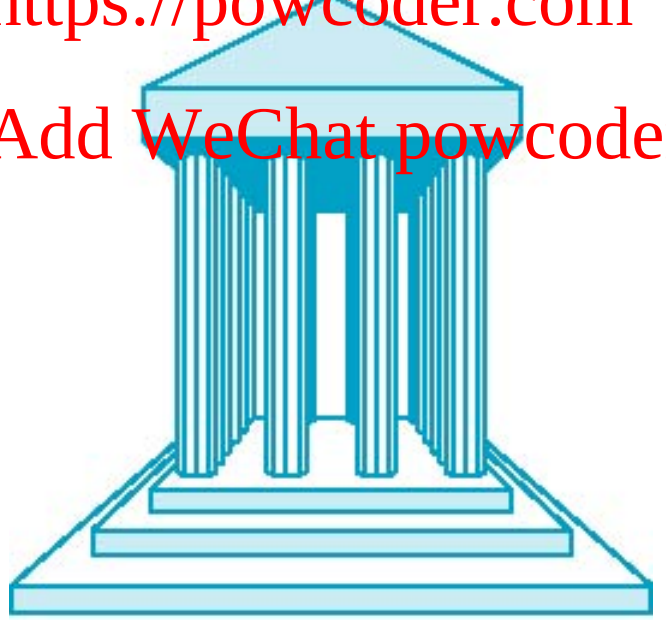
- No principal face parallel to projection plane
- Three vanishing points for cube

Two-Point Perspective

- On principal dir parallel to projection plane
- Two vanishing points for cube

Two-Point Perspective

- One principal face parallel to projection plane
- One vanishing point for cube



Advantages/Disadvantages

- Objects further from viewer are projected smaller than same sized objects closer to the viewer (*diminution*)
  - Looks realistic
- Equal distances along a line are not projected into equal distances (*nonuniform foreshortening*)
- Angles preserved only in planes parallel to the projection plane
- More difficult to construct by hand than parallel projections (but not more difficult by computer)

Computer Viewing

- Three aspects of viewing process:
  - Positioning the camera
    - Setting the model-view matrix
  - Selecting a lens
    - Setting the projection matrix
  - Clipping
    - Setting the view volume
- All implemented in the pipeline

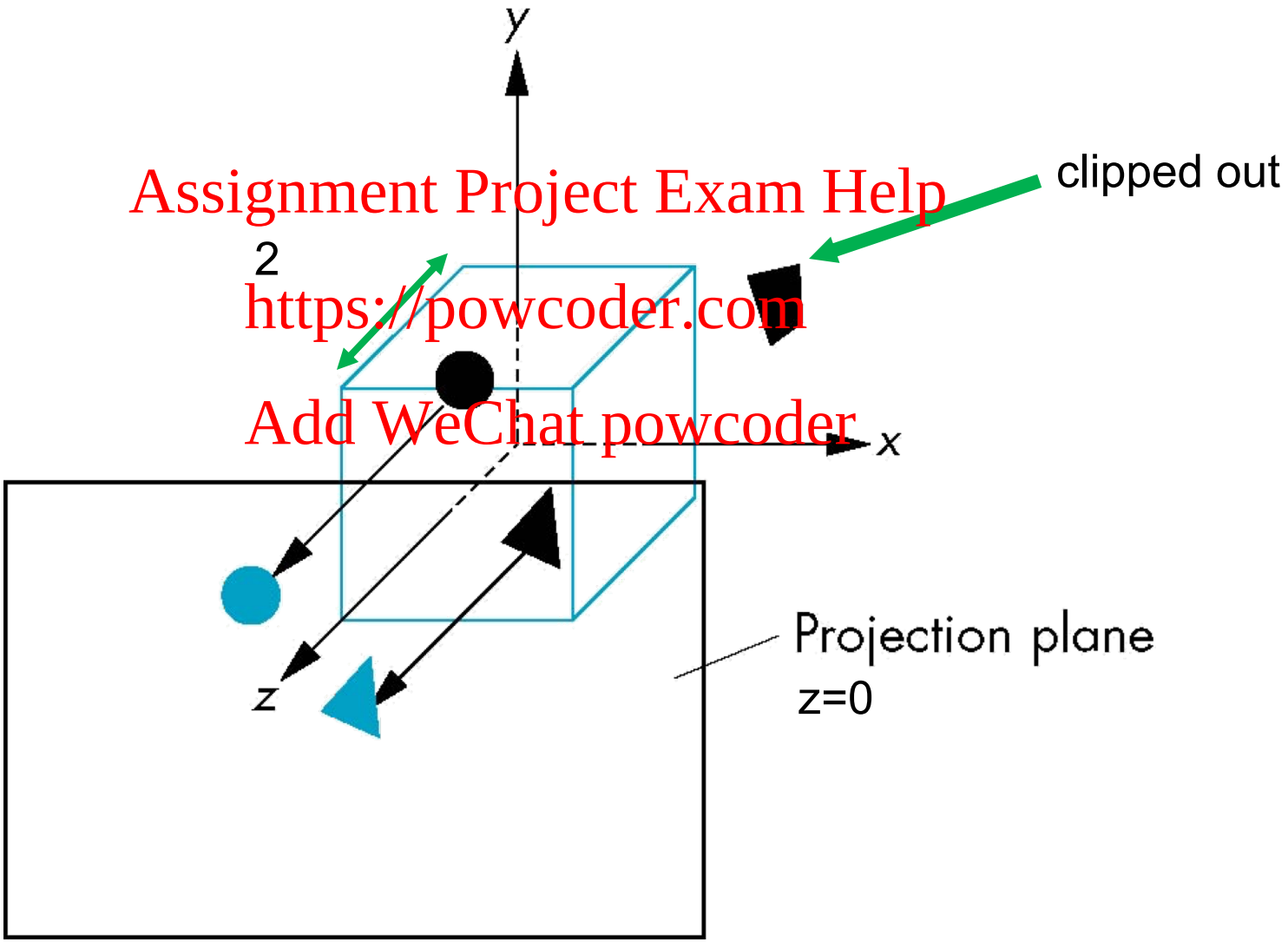
The WebGL Camera

- Technically everything is fixed so that:
  - Camera is located at the origin looking down the negative z direction
  - Orthogonal projection
  - Clipping volume is the cube with sides of length 2 centered at the origin
    - So from (-1,-1,-1) to (1,1,1)
- We implement the model-view, projection, and clipping matrices to force our model in into the fixed WebGL frame
  - All of them get multiplied together and applied in the vertex shader

The WebGL Camera

So far we have been mainly using the identity matrix for each one resulting in all models needing to be in the clipping box, shown with a parallel projection, and more negative z values being further away (and positive z values being behind the camera)

Default Projection



Projections and Normalization

- WebGL (and most graphics systems) use *view normalization*

  - All views must be converted to the default view via transformations

  - Means a single pipeline regardless of view

Default Orthographic Projection

$$x_p = x$$
$$y_p = y$$
$$z_p = 0$$
$$w_p = 1$$

$$\mathbf{p}_p = \mathbf{M}\mathbf{p}$$

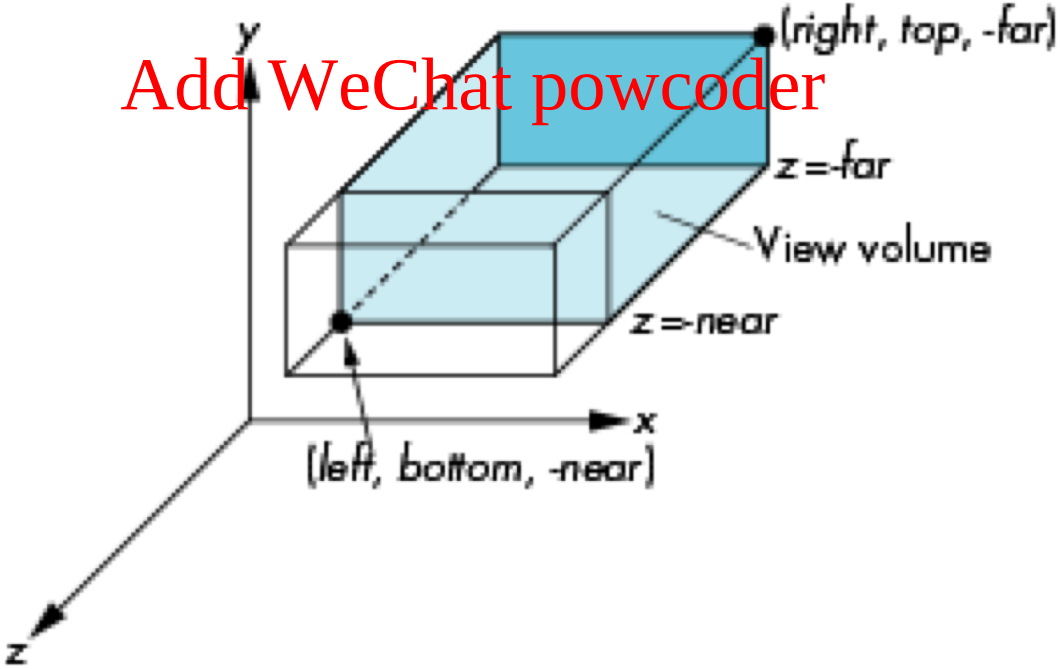
$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Why is $z_p = 0$?

In practice we can use $\mathbf{M} = \mathbf{I}$ and let the GPU set z_p to zero so it can record depth of fragments

Orthogonal Viewing

- Ultimately we translate and scale the clipping box to make the view the same as the default view
- We can develop a system where we specify the faces of the parallelepiped of the clipping box



Orthogonal Normalization Matrix

- Two steps
  - Move center to origin
  - Scale to have sides of length 2

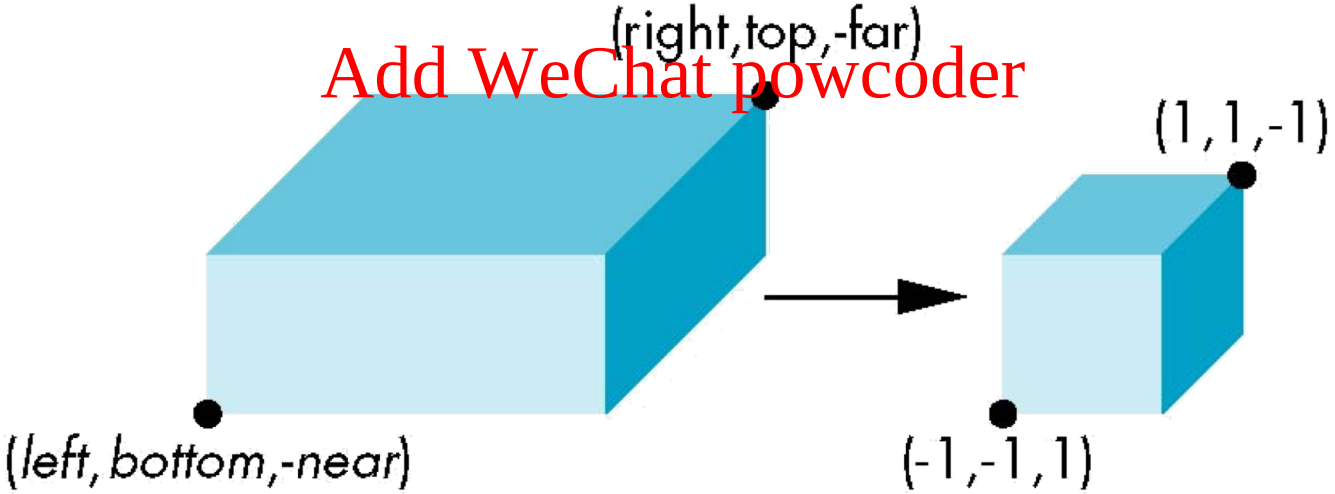
$$\mathbf{P} = \mathbf{ST} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where r, l, t, b, n, f are the `right`, `left`, `top`, `bottom`, `near`, `far` values of the orthogonal region

Orthogonal Normalization


```
glMatrix.mat4.ortho(out,
      left, right, bottom, top, near, far)
```


normalization means using the transformation to convert specified clipping volume to default



Adding Projection

- Need to add a new matrix to our vertex shader
  - It only applies to the `gl_Position`, no other computations
- Need to update the value
  - In this case in the `updateProjectionMatrix()` which is called in response to the boxes changing
  - Make an orthographic projection matrix:

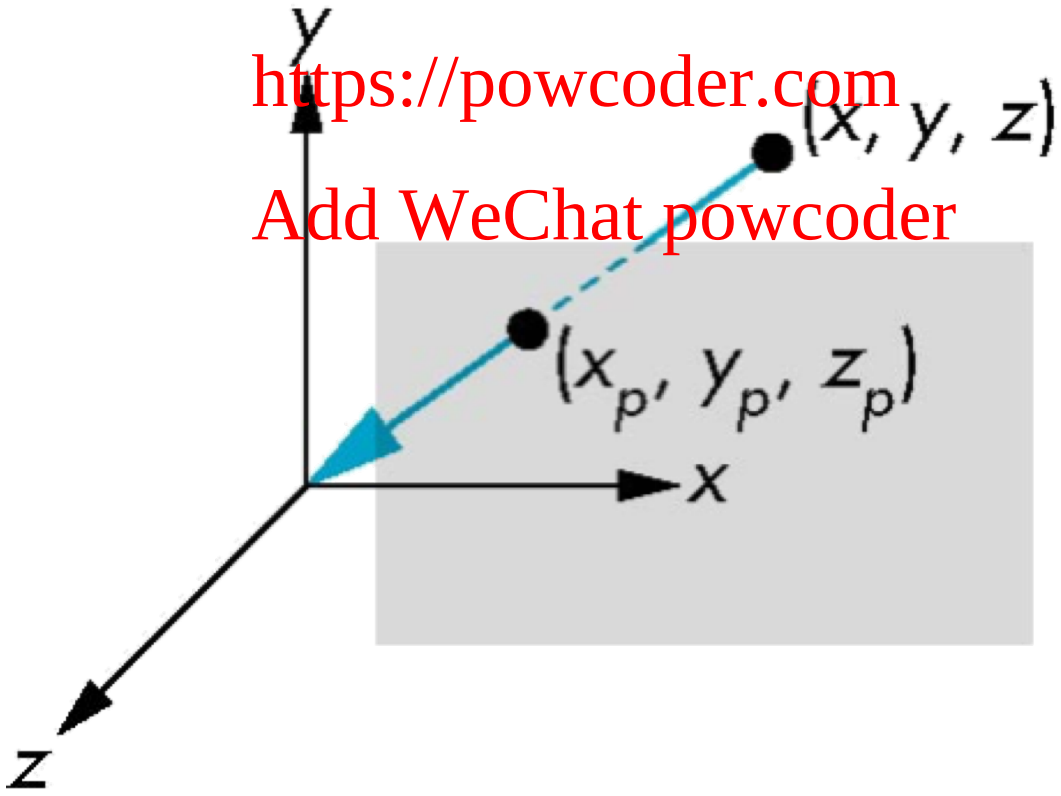
```
mat4.ortho(out,
   left, right, bottom, top, near, far)
```

Not "square" though…

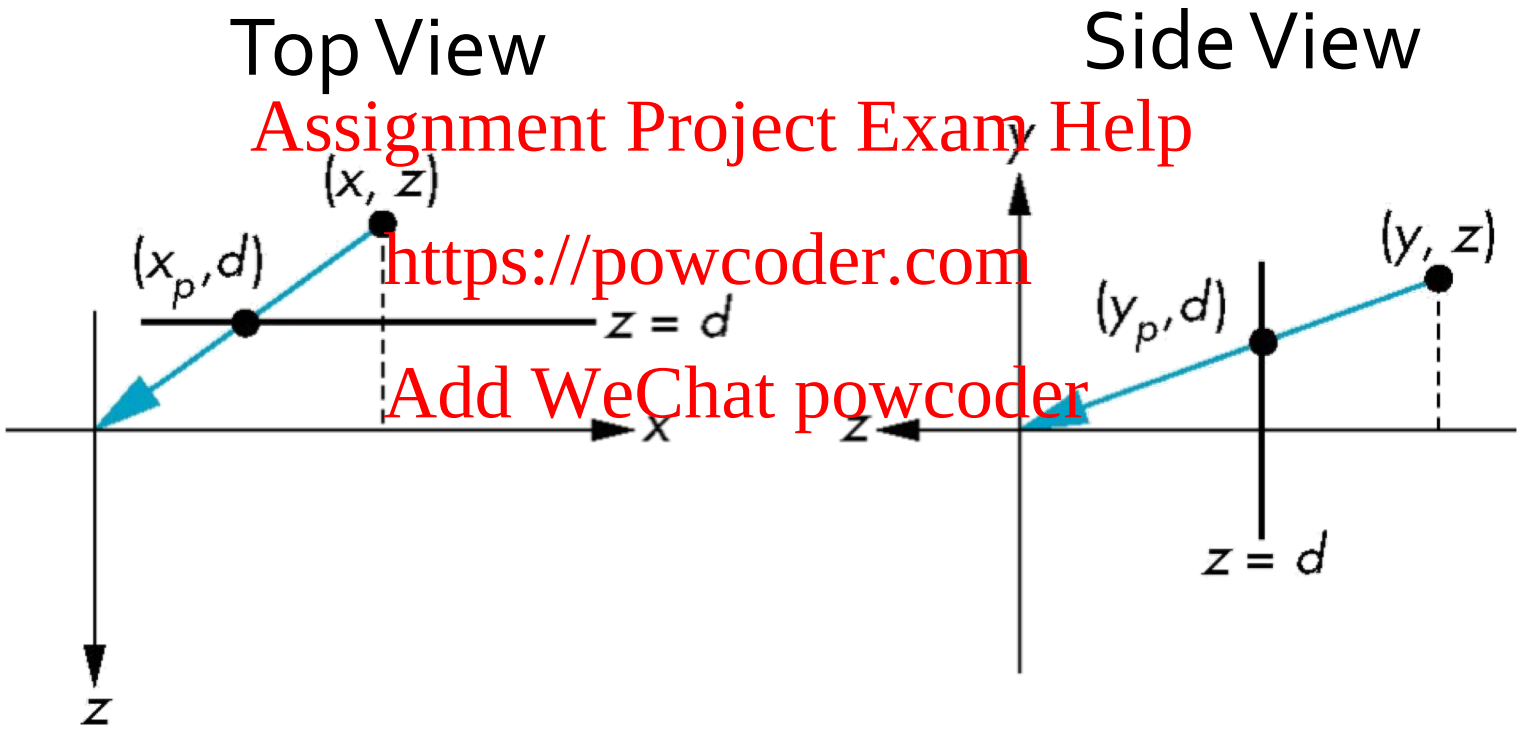
- Want to maintain "aspect ratio"
  - What is aspect ratio?
  - How can we control it?

Simple Perspective Viewing

- Center of projection at the origin
- Projection plane $z = d$ with $d < 0$



Perspective Equations



$$x_p = \frac{x}{z/d} \qquad y_p = \frac{y}{z/d} \qquad z_p = d$$

Perspective Homogeneous Form

$$\mathbf{p}_p = \mathbf{M}\mathbf{p} \text{ where } \mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

$$\text{Thus } \mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \text{ is transformed to } \mathbf{p}_p = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix}$$

Perspective Division

$[x \quad y \quad z \quad 1]^T$ **is transformed to** $\left[x \quad y \quad z \quad \frac{z}{d}\right]^T$

- However $w \neq 1$ so no longer have a point
  - Also $w \neq 0$ so not a vector either…

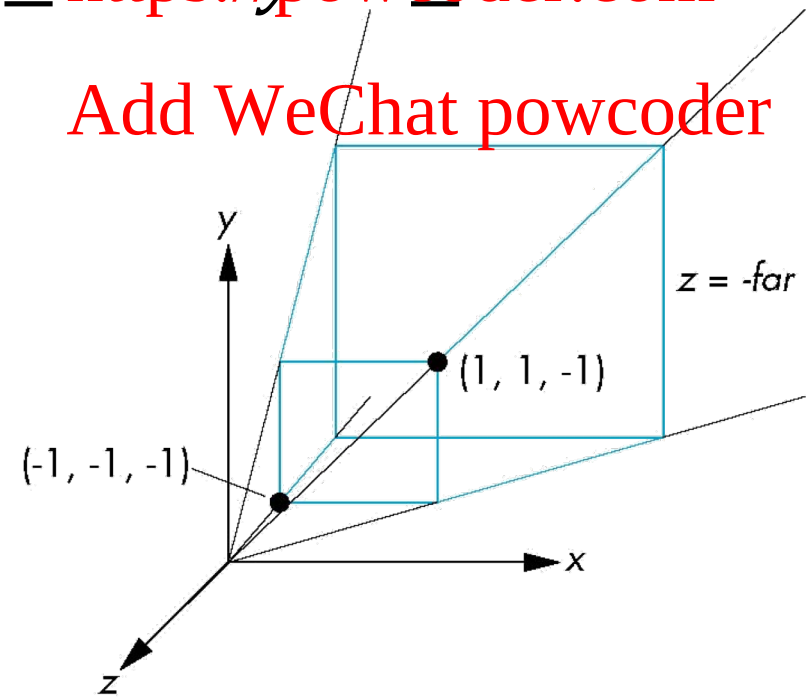
- Must divide by w to get homogeneous coords back
- The ***perspective division*** yields:

$$\left[\frac{x}{z/d} \quad \frac{y}{z/d} \quad d \quad 1\right]$$

  - Which is the desired perspective equations derived earlier
  - The projected values of x and y are now inversely dependent on the point's depth (z)

Simple Perspective

Consider a perspective with the COP at the origin, the near clipping plane at $z = -1$, and a 90° field of view (FOV) determined by the side planes making $x = \pm z$ and $y = \pm z$



Simple Perspective Matrix

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

The matrix is independent of the far clipping plane location

Generalized Perspective Matrix

$$N = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- After perspective division $(x, y, z, 1)$ goes to

$$x'' = x/z$$
$$y'' = y/z$$
$$z'' = -(\alpha + \beta/z)$$

- Which projects orthogonally to the desired point regardless of α and β

Picking α and β

- If we pick

$$\alpha = -\frac{far + near}{far - near} \qquad \beta = -\frac{2 * near * far}{far - near}$$

- Then:
  - the near plane is mapped to $z = -1$
  - the far plane is mapped to $z = 1$
  - the sides are mapped to $x = \pm 1, y = \pm 1$
- Hence the new clipping volume is the default clipping volume

Generalized Perspective Matrix

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -\dfrac{f+n}{f-n} & -\dfrac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

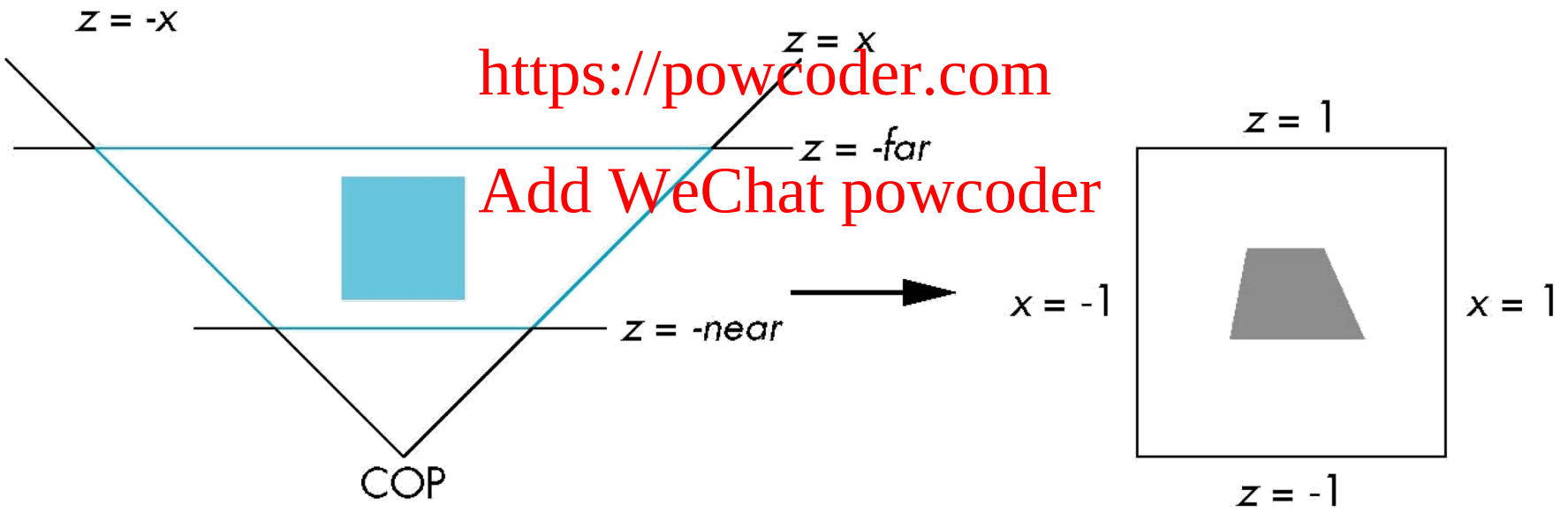
After perspective division $(x, y, z, 1)$ goes to

$$x'' = x/z$$
$$y'' = y/z$$
$$z'' = \frac{1}{f-n}\left(f + n + \frac{2nf}{z}\right)$$

Normalization Transformation

Original Clipping Volume and Original Object

New Clipping Vol. and Projected Object



Fully Generalized Perspective Matrix

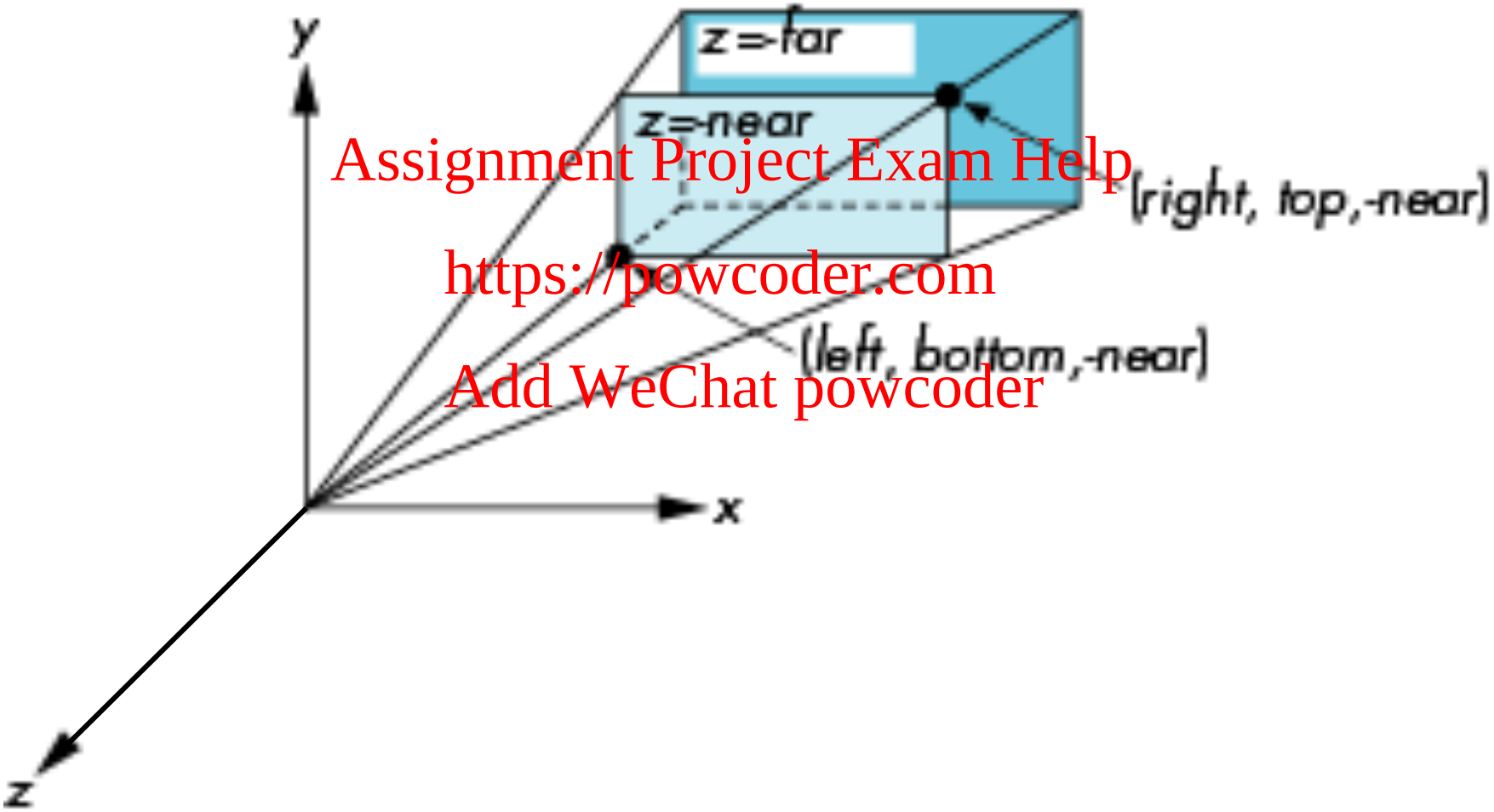
- Previous example still fixed the right, left, top, and bottom clipping planes
- How can we make these adjustable?
  - Need to incorporate shearing and scaling transformations

Fully Generalized Perspective Matrix: *Frustum*

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

r, l, t, b are the right, left, top, bottom coords of near plane and n, f distances to the near and far planes

Perspective Frustum: Truncated pyramid



glMatrix Perspective: Frustum

- `glMatrix.mat4.frustum` function:
  - `out` argument: a `mat4` to save to
  - `left, right, bottom, top`: specify corners of near plane
  - `near, far`: distances from the COP (fixed to the origin) and must be $far > near > 0$

Simplification

- This is a bit too general for most situations and hard to get all the parameters just right
- If we assume the near plane is symmetrical about the z axis :

$$left = -right$$
$$bottom = -top$$

then we can simplify the matrix

  - This eliminates the shearing component of the matrix

Perspective Matrix

$$\mathbf{P} = \mathbf{NS} = \begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

More Natural Perspective Matrix

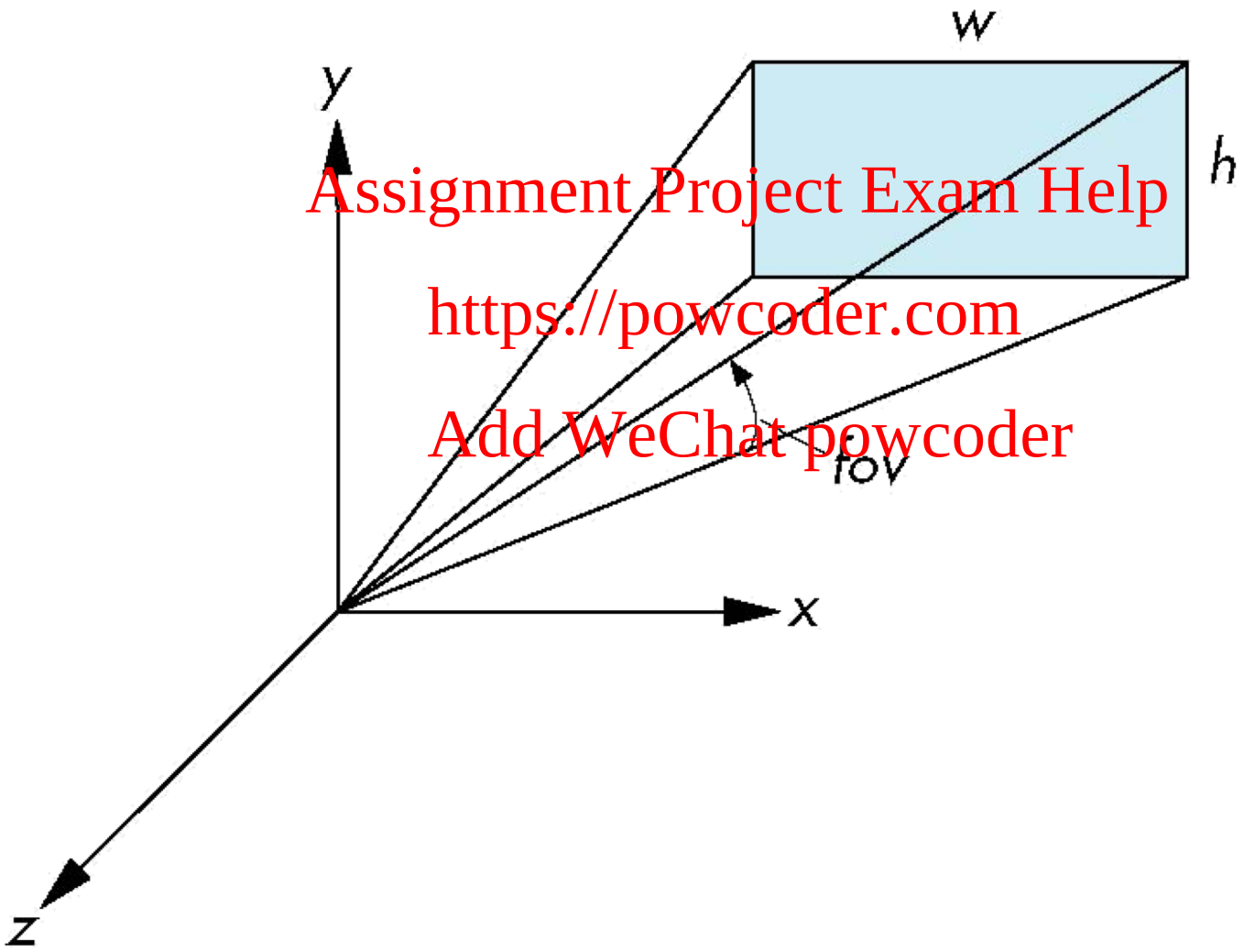
- Having to determine the positions of the top and right clipping planes is still a bit unnatural
- We can redefine top and $right$ in terms of more logical values:

$$t = n * \tan(fovy)$$

$$r = t * aspect$$

- where $fovy$ is the angle between the top and bottom planes and $aspect$ is the ratio of width to height of the near (or far) plane

Perspective with FOV



near plane

aspect = $\frac{w}{h}$

Simplified Perspective Matrix

$$\mathbf{P} = \mathbf{NS} = \begin{bmatrix} \frac{\cot(fovy)}{aspect} & 0 & 0 & 0 \\ 0 & \cot(fovy) & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

`glMatrix.mat4` function:

```
perpsective(out, fovy, aspect, near, far)
```

glMatrix Perspective with FOV

`perpective(out, fovy, aspect, near, far)`

- `out` is the mat4 to save to
- `fovy` is the FOV angle in the y (or up) direction (angle between top and bottom planes)
- `aspect` is the ratio of width to height of the clipping planes
- `near` and `far` are distances from COP to `near` and `far` planes

Assignment Project Exam Help

Exercise: Changing the View

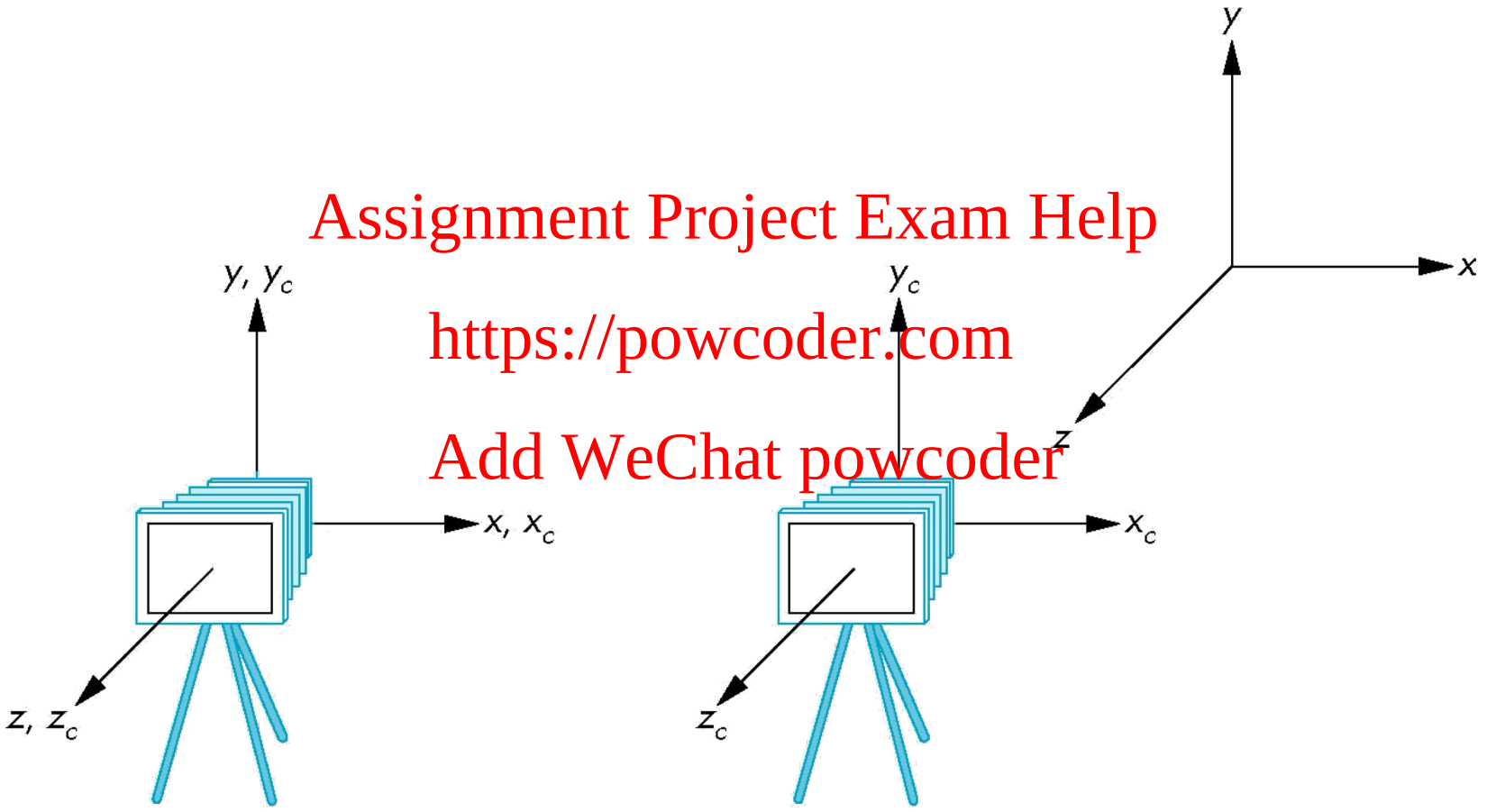
- Start with the static cube
- Add two `uniform mat4` variables to the vertex shader for the model-view and perspective matrices
  - Then set `gl_Position` to the projection matrix times the model-view matrix times the position
- Grab the uniform locations of the matrices in JS
- Add sliders to control the camera position using `lookAt`:
  - theta (-90 to 90), phi (-90 to 90), distance (0.05 to 10)
- Add sliders that control the `perspective` matrix:
  - near (0.01 to 3), far (3 to 10), aspect (0.5 to 2), FOV (10 to 120)
- Whenever a slider changes re-render the scene with the new matrices

Moving the Camera

- If we want to visualize objects with both positive and negative z values we can either:
  - Move the camera in the positive z direction
    - Translate the camera
  - Move the objects in the negative z direction
    - Translate all objects
- These are equivalent since everything is relative so we want to incorporate this into our model-view matrix

`translate(0, 0, -d)` with `d` > 0

Moving the Camer

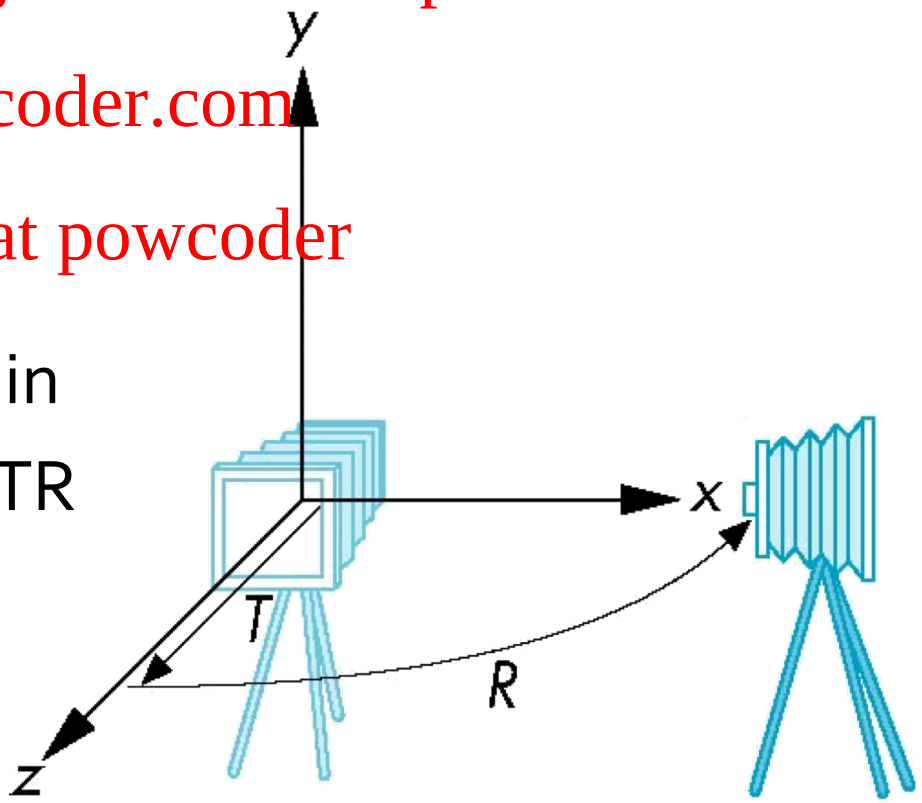


default frame

frames after translation by $-\mathbf{d}$

Moving the Camera

- We can move the camera to any desired position by a sequence of rotations and translations
- Example: side view
  - Rotate the camera
  - Move it away from origin
  - Model-view matrix C = TR



Moving the Camera

- Difference between moving/rotating the object or moving rotating the camera?
  - No
- Results in the same view

Look-At Viewing Specification

- We can modify the PHIGS/GKS-3D viewing specification slightly to get `lookAt` which takes:
  - Position of eye/camera
  - Position to look at
  - The vector describing the up direction
- Used to be included in OpenGL but now we have a method in MV.js to create the model-view matrix:


```
let eye = vec3(1, 1, 1);
let at  = vec3(0, 0, 0);
let up  = vec3(0, 1, 0);
let mv  = lookAt(eye, at, up);
```

  - This is an isometric view of cube aligned with axis
  - Internally still just a translation and rotation

Other Viewing Specification

- Depending on situation others might be useful:
  - Flight simulator: yaw, pitch, and roll
  - Sky coordinates: elevation, azimuth, twist
  - Direction angles
- In the end they all just create a translation and rotation for our camera/object, difference is in what values are given

Exercise: Rotating Cube

- Copy the full rotating cube example
- Add a `<div>` to the HTML that has the `style`:

```
background-color:rgba(255,255,255,0.5);left:0;right:0;bottom:0
```


- Add a checkbox to this `<div>` that when checked causes the display to be perspective and when unchecked has an orthographic view

  - The cube should minimally move when switching views
- The canvas is always the full height and width as the HTML document but the cube stays a cube
  - This will require changing the view, using `aspect` (for perspective) and the ratio of side length (for orthographic)