

# Matrix Order

*Normally projection has to apply to all objects (i.e. the entire scene) thus it must pre-multiply the modelview matrix*

- $M = M_{\text{proj}} M_{\text{modelview}}$  Or
- $M = M_{\text{proj}} M_{\text{view}} M_{\text{model}}$

*However, with shaders you have absolute control of the matrices and the way they are multiplied*

# Important

***Projection parameters are given in CAMERA Coordinate system (Viewing).***

Assignment Project Exam Help

<https://powcoder.com>

***So if camera is at  $z = 50$ , is aligned with the world CS, and you give  $near = 10$  where is the near plane with respect to the world?***

Add WeChat powcoder

# Important

*Projection parameters are given in CAMERA Coordinate system (Viewing).*

Assignment Project Exam Help

<https://powcoder.com>

*So if the camera is at  $z = 50$ , is aligned with the world CS, and you give  $|near| = 10$  where is the near plane with respect to the world?*

Add WeChat powcoder

- Transformed by  $\text{inverse}(M_{vcs})$
- i.e.  $(0,0,40)$

# Nonlinearity of perspective transformation

## Tracks:

Left:  $x = -1$ ,  $y = -1$

Right:  $x = 1$ ,  $y = -1$

$Z = -\infty, \infty$

## View volume:

Left = -1, right = 1

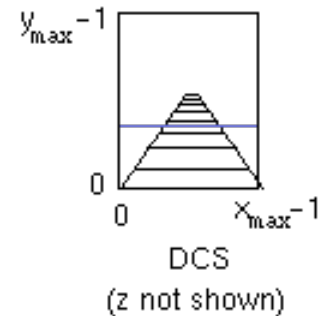
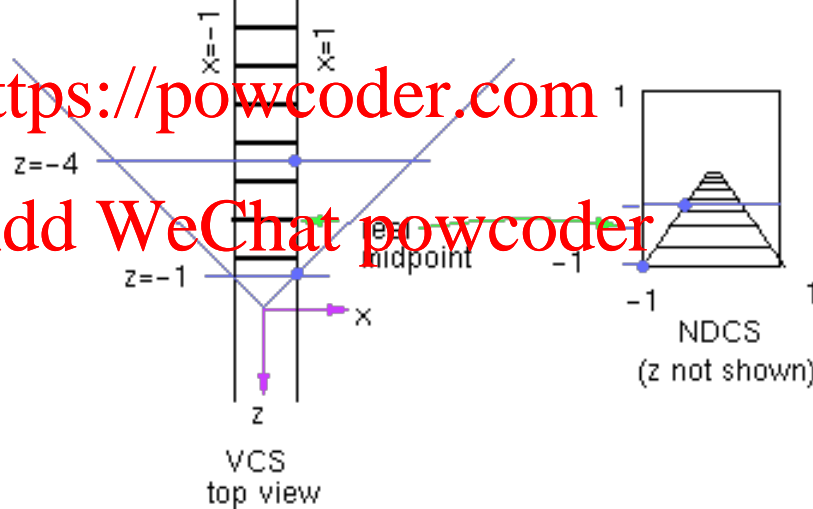
Bot = -1, top = 1

Near = 1, far = 4

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

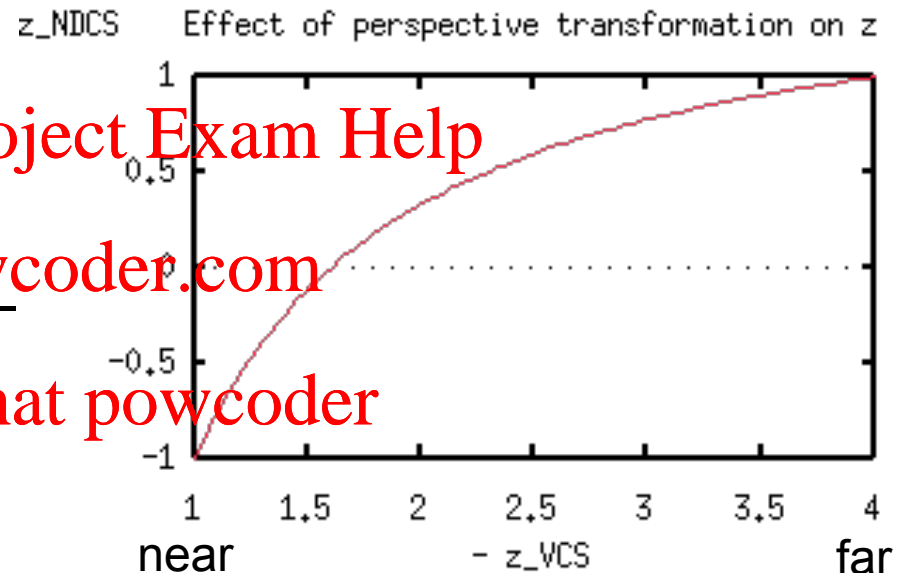


# Z in NDCS vs -Z in VCS

$$Z_{NDCS} = \frac{5}{3} + \frac{8}{3} \left( \frac{-Z_{VCS}}{\text{far} - \text{near}} \right)$$

Reminder:  $Z_{VCS} := -\text{near} \rightarrow Z_{NDCS} = -1$   
 $\quad \quad \quad -\text{far} \rightarrow Z_{NDCS} = 1$

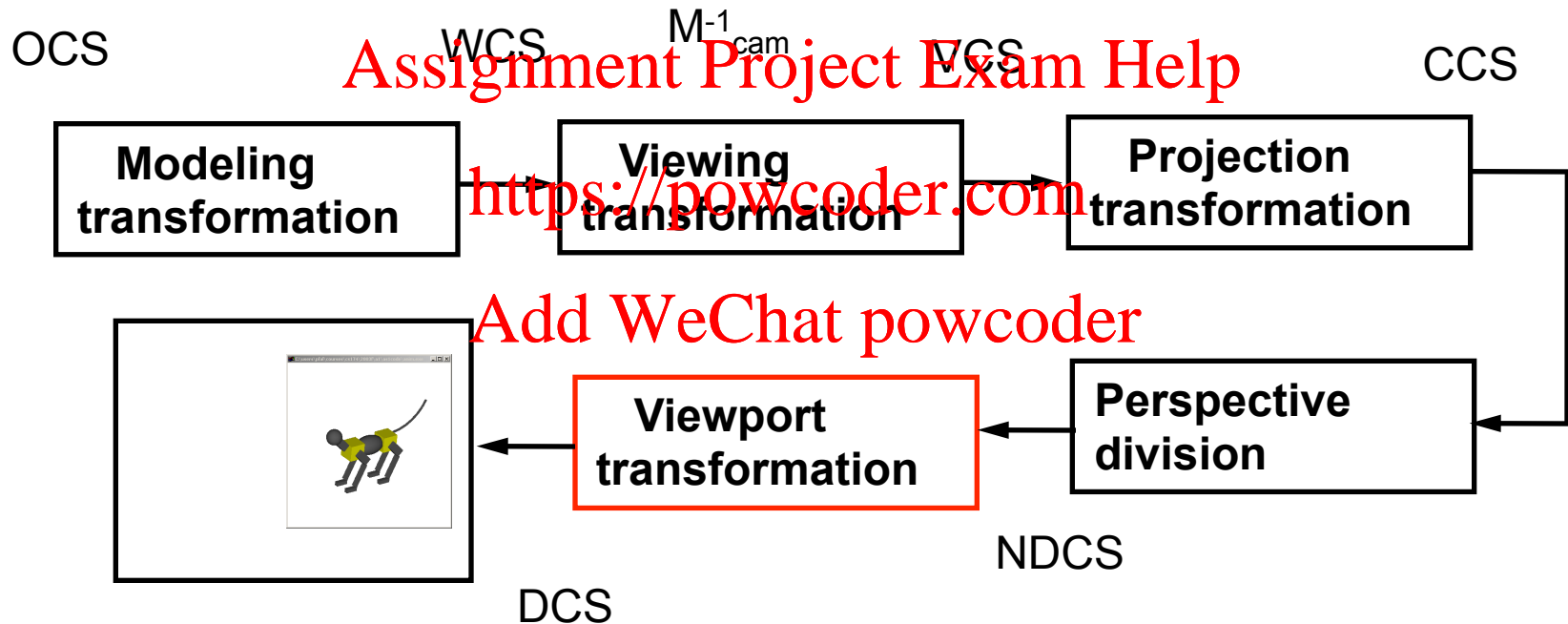
Notice that the curve flattens as  
 $-Z_{VCS} \rightarrow \text{far}$



On systems with limited numerical precision for the z-buffer (e.g. 8 bits) a large difference in near and far can result in multiple  $Z_{VCS}$  values to map on the same value in  $Z_{NDCS}$ . As a result the graphics system cannot resolve visibility correctly!

Rule of thumb: Limit the z-range as much as you can

# Viewport transformation



# Viewport

**NDCS**

**Screen Space**

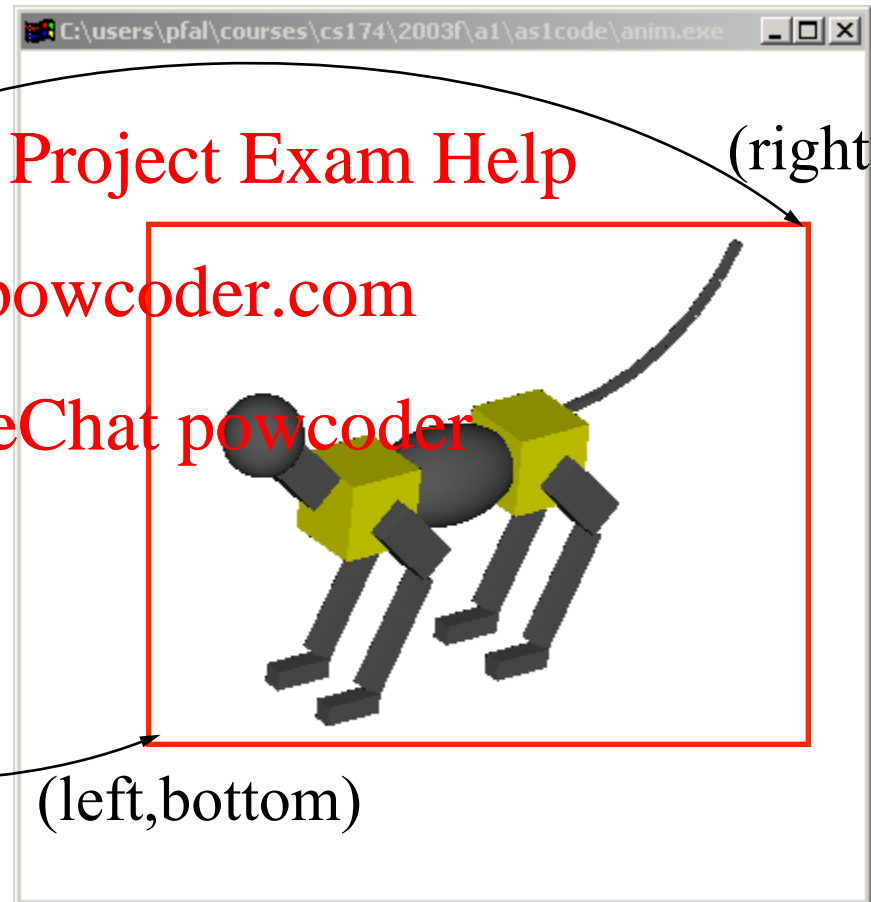
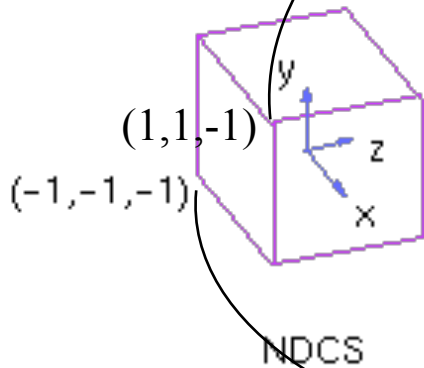
$(1,1,-1) \rightarrow (\text{top}, \text{bottom}, 0)$

$(-1,-1,-1) \rightarrow (\text{left}, \text{bottom}, 1)$

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder



# Example: Full window coverage

- Transforms the canonical coordinates to a viewport of size  $W \times H$  from  $(0,0)$  at lower left; thus, viewport is  $[0, W] \times [0, H]$
- Scales and translates  $z$  to be in  $[0, 1]$

$$\mathbf{M}_{VP}^{Full} = \begin{bmatrix} 1 & 0 & 0 & W/2 \\ 0 & 1 & 0 & H/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{W}{2} & 0 & 0 & 0 \\ 0 & \frac{H}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

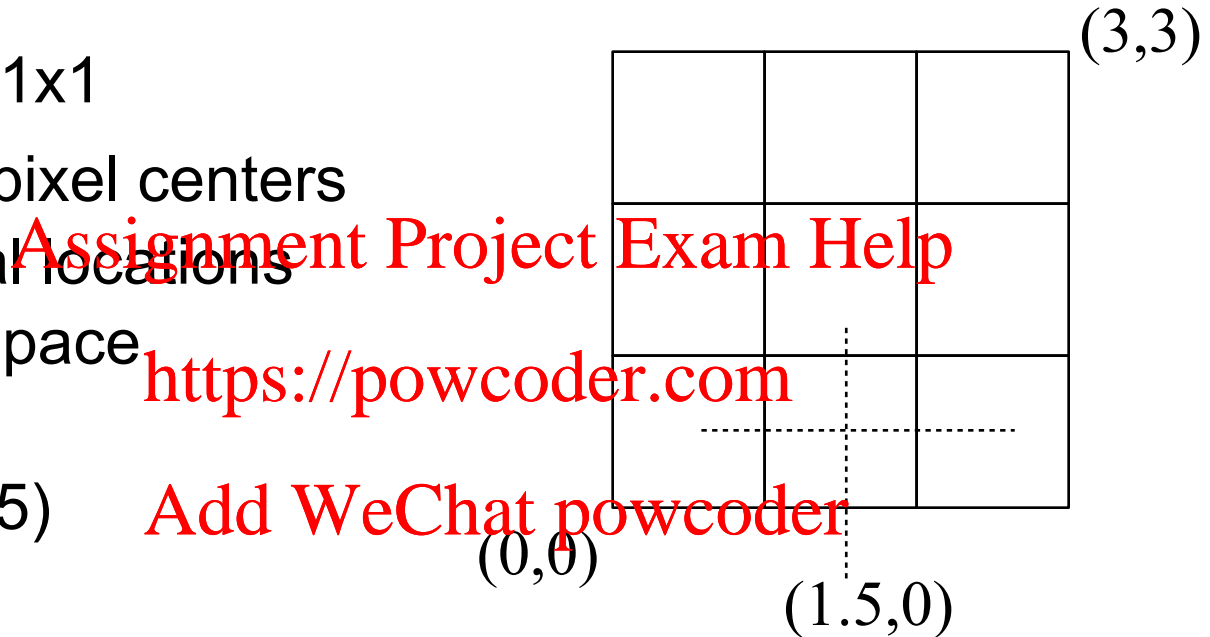
- How does a partial coverage matrix look like?



# So..Pixel Centers?

- Pixel size: 1x1
- Therefore pixel centers at fractional locations in screen space

$$p_{ij} = (i.5, j.5)$$



- In OpenGL the bottom left corner of the window is at (0,0)
- In some windowing systems the top left is at (0,0)
- When do you care about this?...When needing the location of the mouse from the windowing system

# Viewport in WebGL

- `gl.viewport( x, y, width, height );`
  - `(x,y)`: lower left corner of viewport rectangle in pixels.
  - `width, height`: width and height of viewport in pixels.
  - Generally put the code in a reshape callback.
  - Example: the whole window
- `gl.viewport(0,0,canvas.width, canvas.height);`

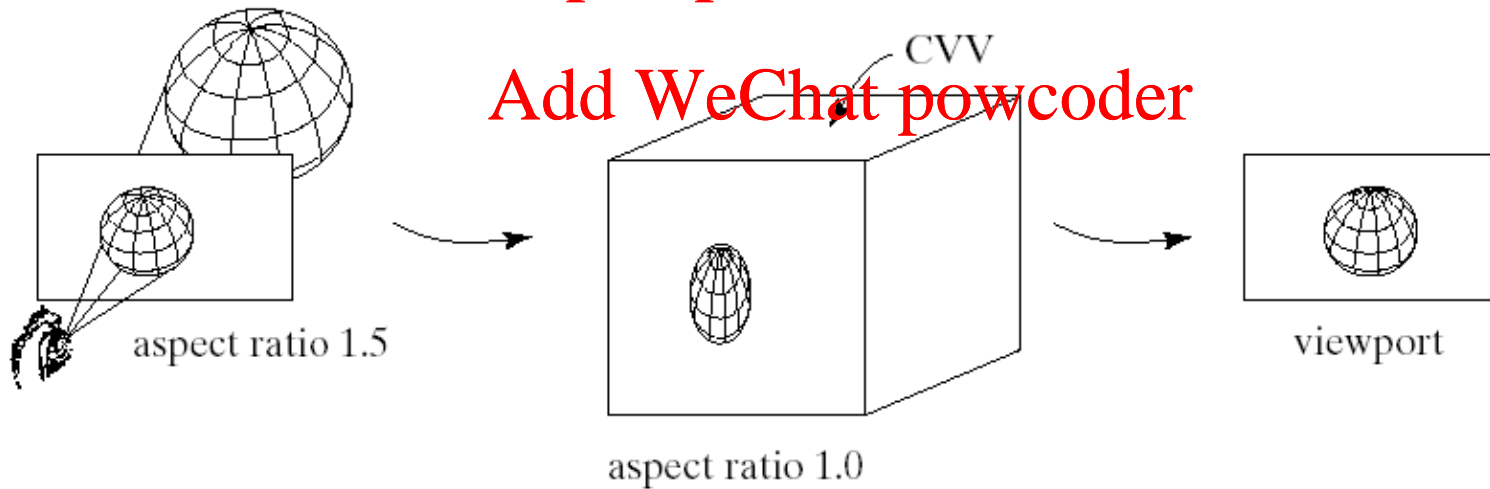
# Why viewports?

*Undo the distortion of the projection transformation*

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder



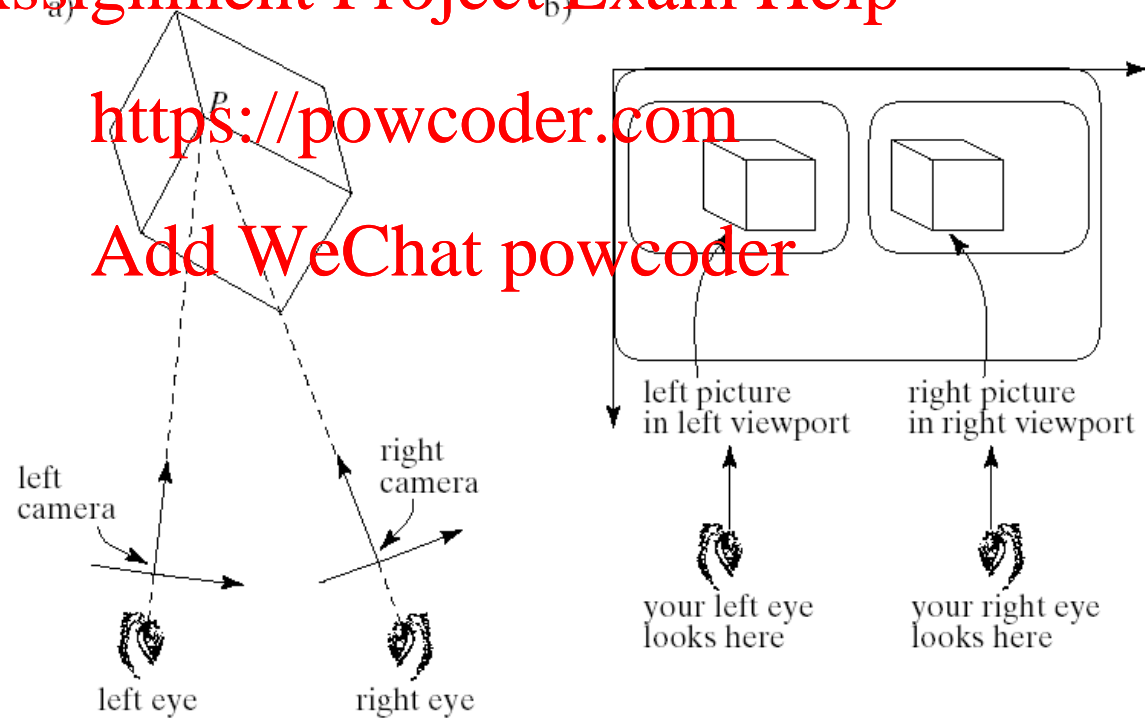
# Stereo views

*Render the scene twice from different points of view*

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder



# Example: Two viewports

```
void render()
{
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    // Set the first viewport
    gl.viewport(0, 0, canvas.width/2, canvas.height/2);
    // Set an orthographic projection matrix
    projectionMatrix = ortho(-3, 3, -3, 3, 1, 100);
    modelViewMatrix = mat4();

    var eye = vec3(0, 0, 10);
    modelViewMatrix = mult(modelViewMatrix, lookAt(eye, at, up));

    drawObjects();

    // Set the second viewport
    gl.viewport(canvas.width/2, canvas.height/2, canvas.width/2, canvas.height/2);
    // Set an orthographic projection matrix
    projectionMatrix = ortho(-3, 3, -3, 3, 1, 100);
    modelViewMatrix = mat4();
    eye = vec3(10, 10, 0);
    modelViewMatrix = mult(modelViewMatrix, lookAt(eye, at, up));

    drawObj();
}
```

# Example: Two viewports

- Viewport one: lower left quadrant
- Viewport one: top right quadrant

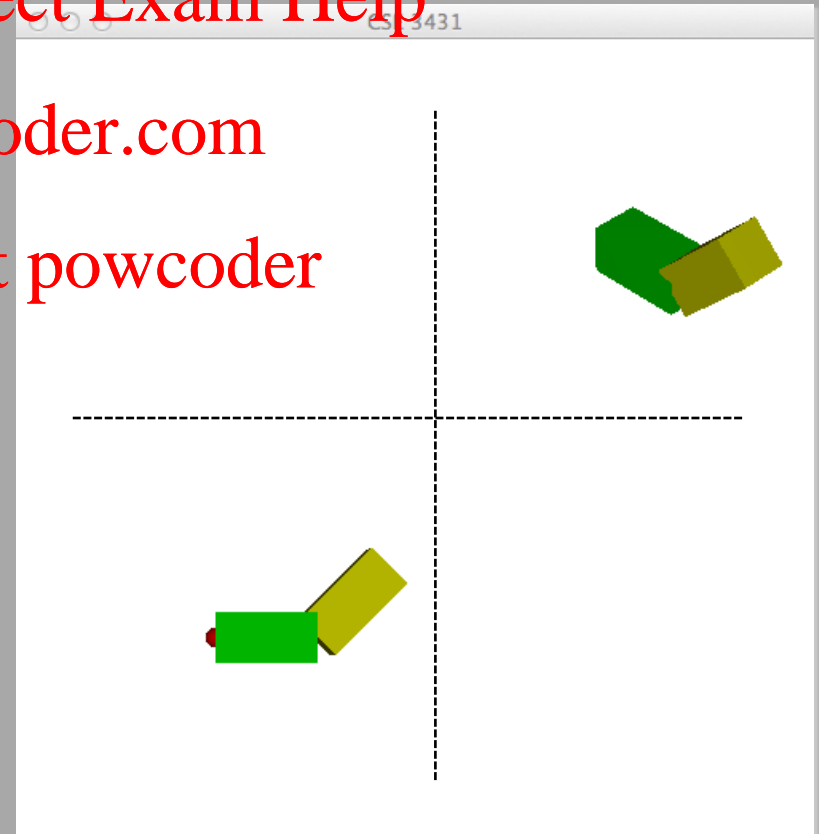
Width: 500 pixels

Assignment Project Exam Help

<https://powcoder.com>

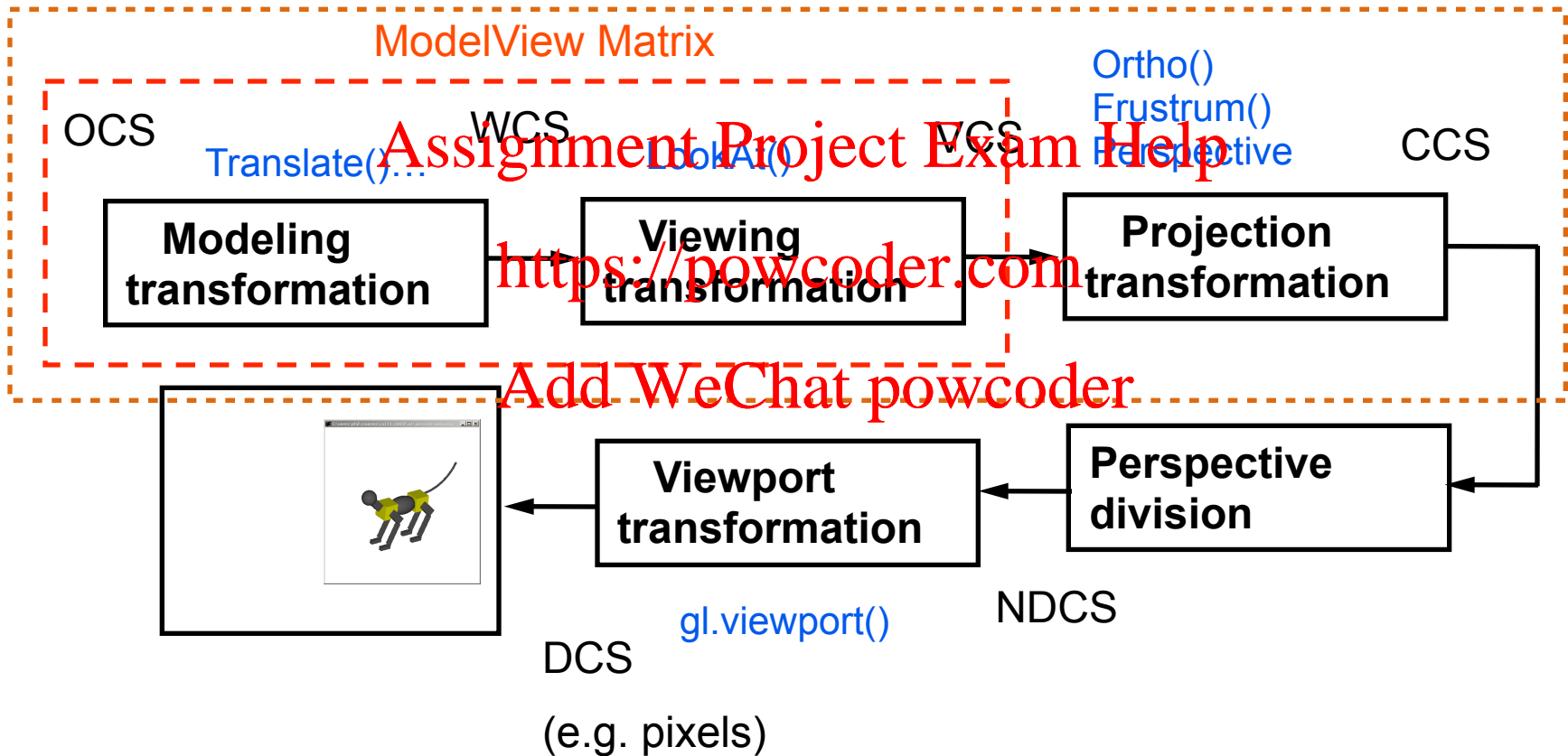
Add WeChat powcoder

Height: 500 pixels



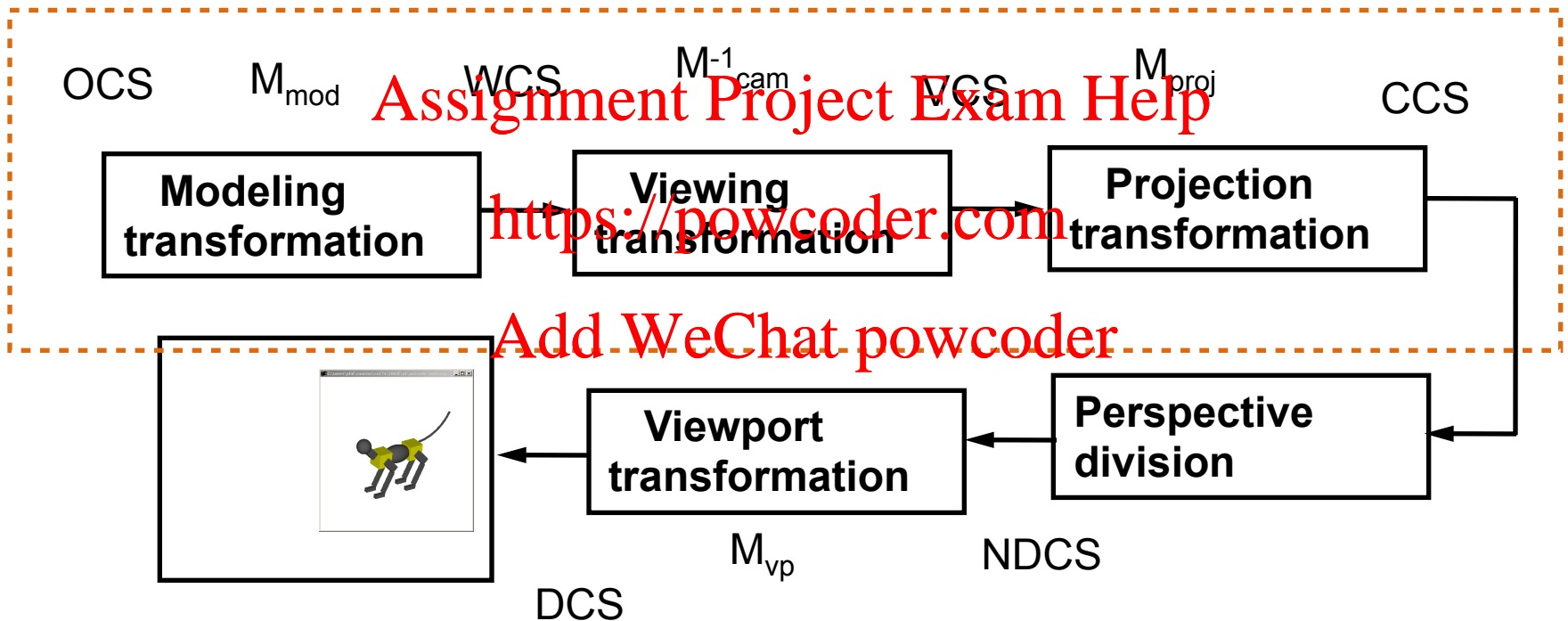
# Transformations in the pipeline

## Vertex Shader



# Matrices in the Pipeline

Vertex Shader





# Vertex Shader

```
attribute vec4 vPosition;  
attribute vec3 vNormal;  
  
varying vec4 fColor;
```

Assignment Project Exam Help

```
void  
main()  
{  
    gl_Position = projectionMatrix * modelViewMatrix * vPosition;  
    fColor = vec4(1.0f, 0.0f, 0.0f, 1.0f) ;  
}  
  
// Notice that perspective division happens later.  
// gl_Position is in homogeneous coordinates
```

<https://powcoder.com>

Add WeChat powcoder

# Line Rendering Algorithm

Compute  $\mathbf{M}_{\text{mod}}$

Compute  $\mathbf{M}^{-1}_{\text{cam}}$

Compute  $\mathbf{M}_{\text{modelview}} = \mathbf{M}^{-1}_{\text{cam}} \mathbf{M}_{\text{mod}}$

Compute  $\mathbf{M}_O$

Compute  $\mathbf{M}_P$  // disregard  $\mathbf{M}_P$  here and below for orthographic-only case

Compute  $\mathbf{M}_{\text{proj}} = \mathbf{M}_O \mathbf{M}_P$

Compute  $\mathbf{M}_{VP}$  // Viewport transformation

Compute  $\mathbf{M} = \mathbf{M}_{VP} \mathbf{M}_{\text{proj}} \mathbf{M}_{\text{modelview}}$

for each line segment  $i$  between vertices  $P_i$  and  $Q_i$  do

$P = \mathbf{M}P_i$ ;  $Q = \mathbf{M}Q_i$

drawline( $P_x/h_P$ ,  $P_y/h_P$ ,  $Q_x/h_Q$ ,  $Q_y/h_Q$ ) //  $h_P, h_Q$  are the 4<sup>th</sup> coordinates of  $P, Q$

end for