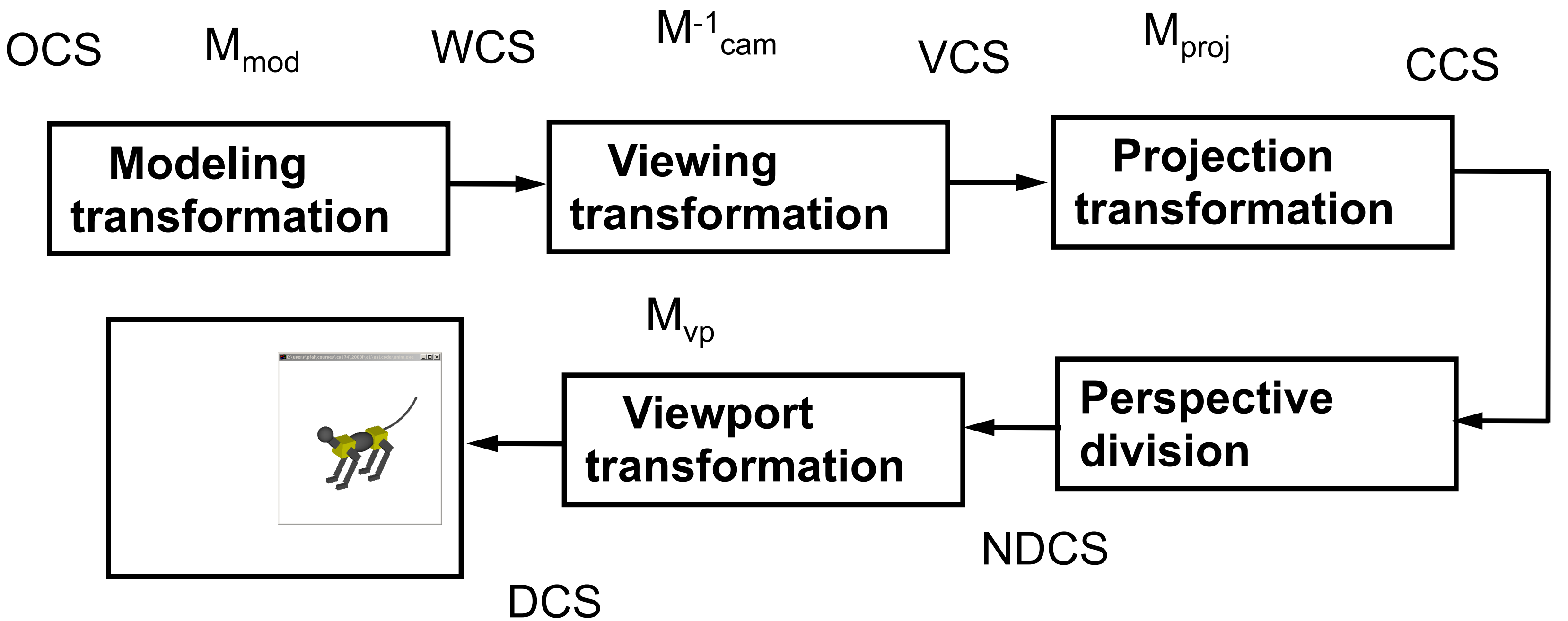
Projection transformation



Matrix Order

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

- $M = M_{proj}M_{modelview}$ or
- $M = M_{proj}M_{view}M_{model}$

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).

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?

Important

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

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?

- Transformed by inverse(Mvcs)
- i.e. (0,0,40)

Nonlinearity of perspective transformation

Tracks:

Left: x= -1, y = -1

Right: x = 1, y = -1

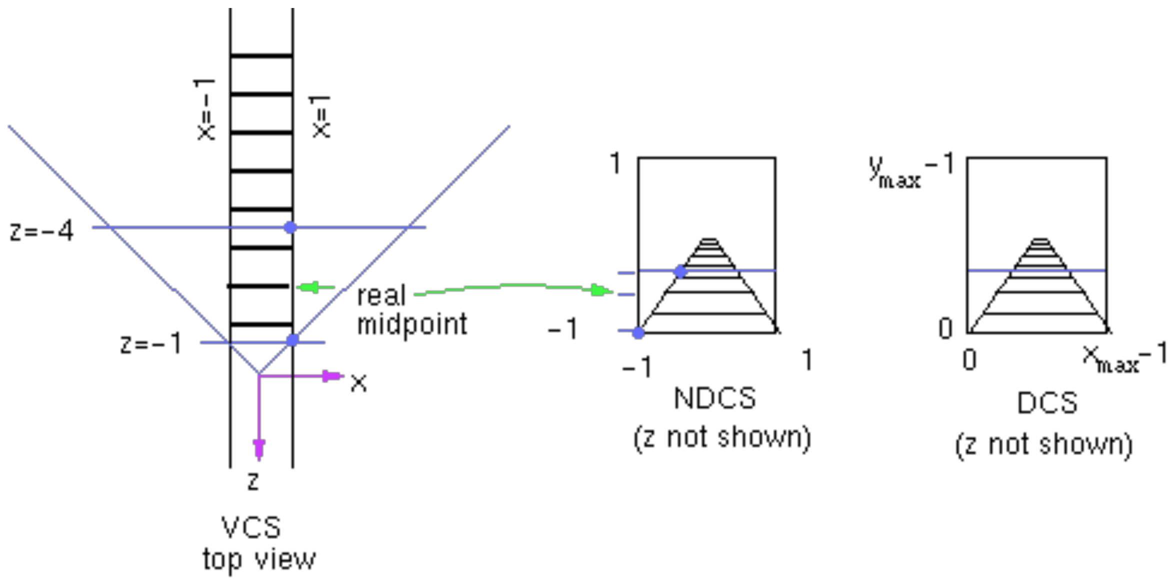
Z = -inf, inf

View CS:

z= -1, z = -4

Midpoint at z = -2
(second track)

NDCS and DCS midpoint is not the projection of the VCS midpoint



Z in NDCS vs -Z in VCS

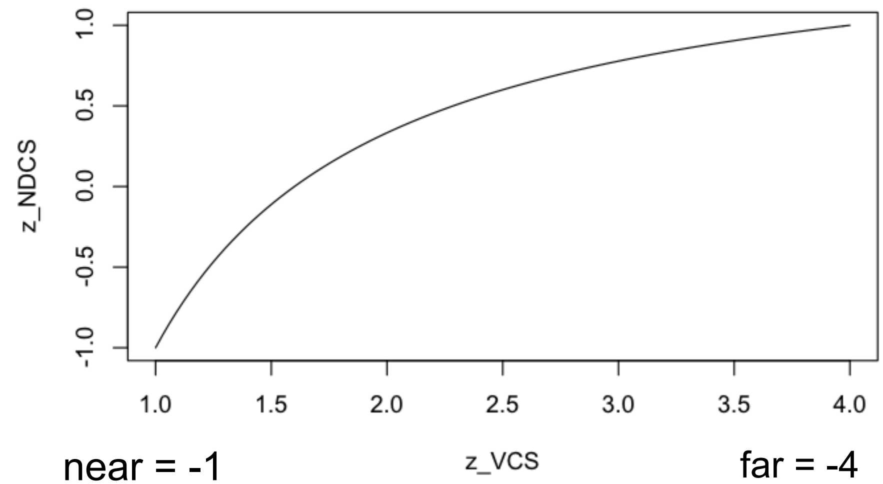
$Z_{NDCS} = 5/3 + 8/(3\,(-Z_{VCS}))$

Reminder: Z_{VCS}:=-near --> Z_{NDCS} = -1
-far --> 1

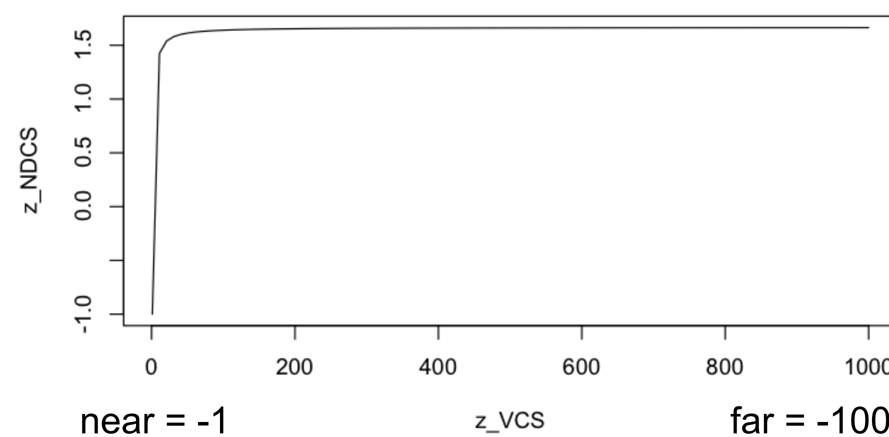
Notice that the curve flattens as $-Z_{VCS}$--> far

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



near = -1 far = -4

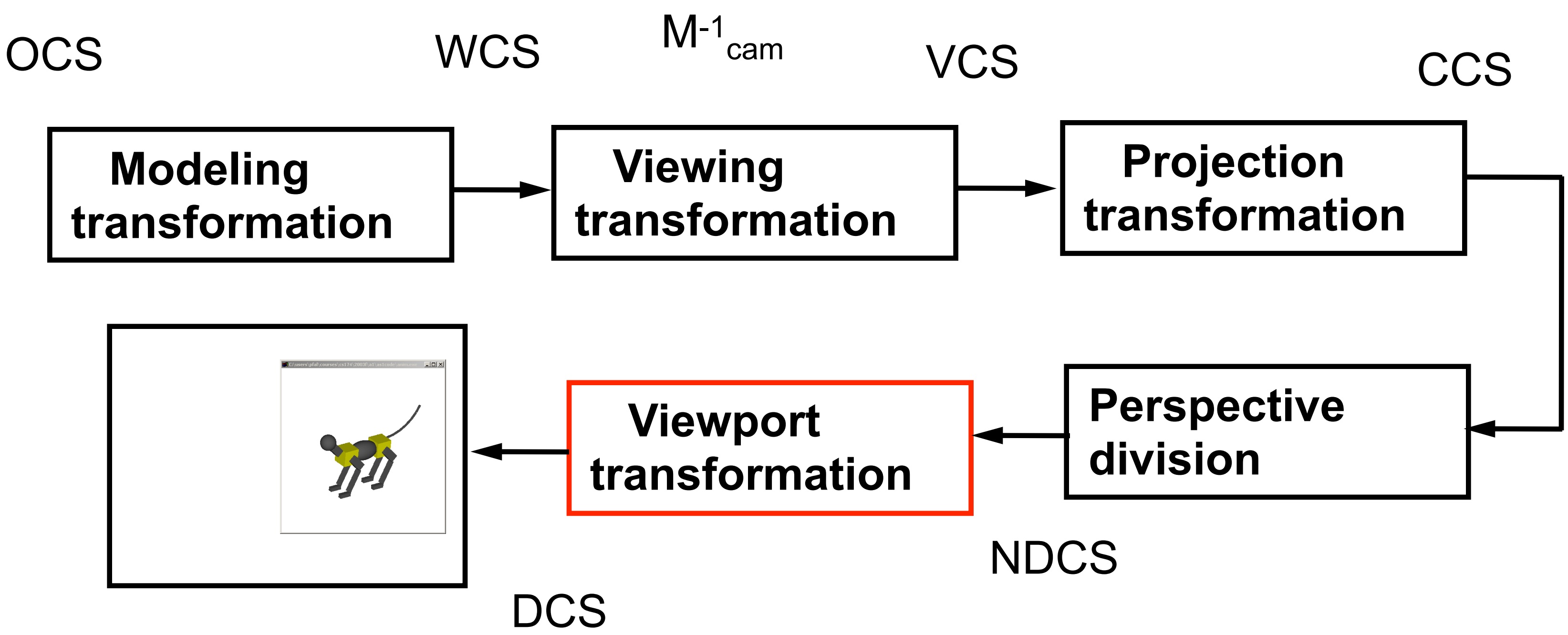


near = -1 far = -1000

Vertex Shader

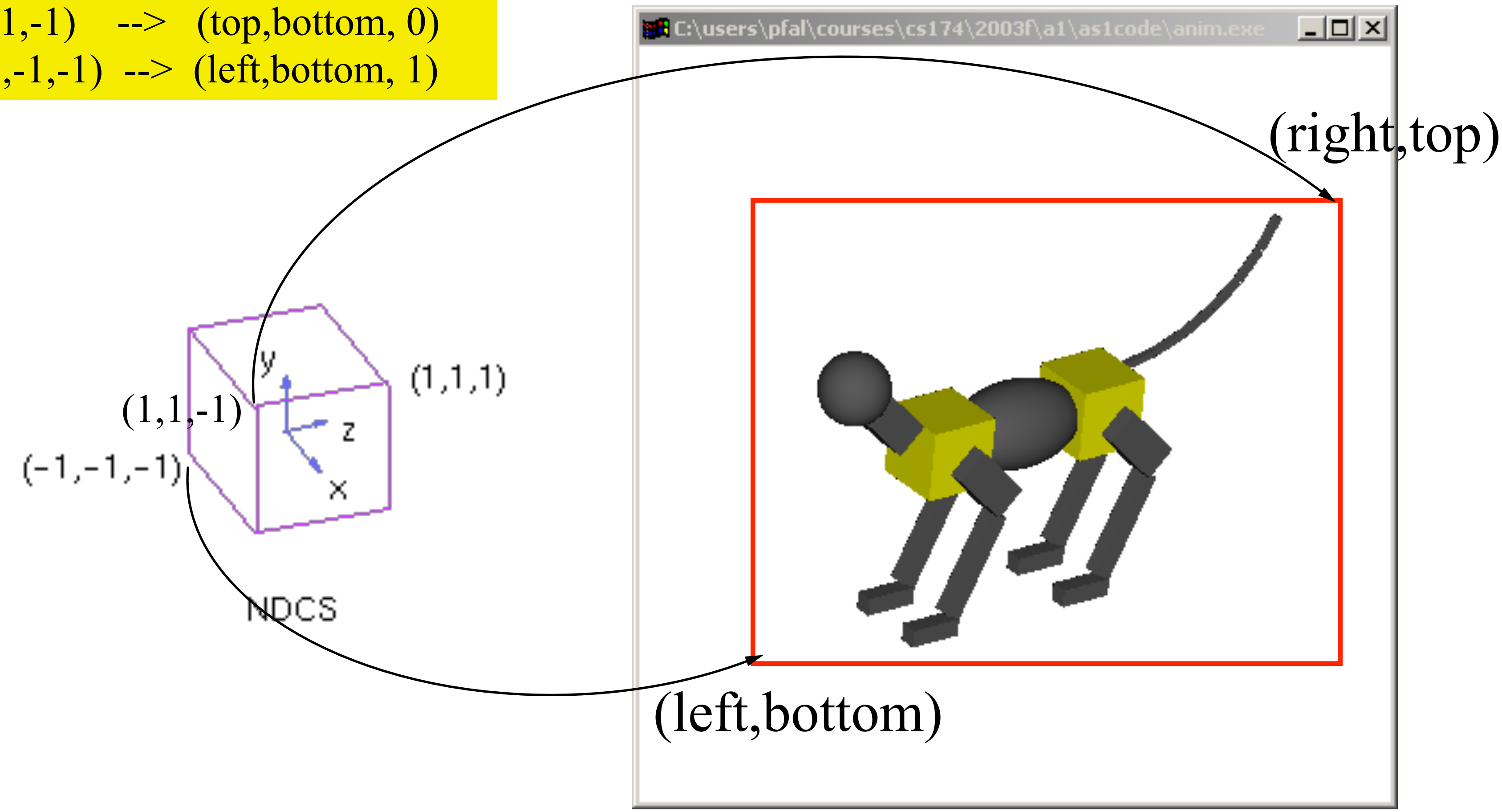
```
in vec4 vPosition;
in vec3 vNormal;

uniform mat4 projectionMatrix, modelViewMatrix

out vec4 fColor;


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 CLIPPING Coordinates
```

Viewport transformation



Viewport

NDCS **Screen Space**
(1,1,-1) --> (top,bottom, 0)
(-1,-1,-1) --> (left,bottom, 1)



Example: Full window coverage

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

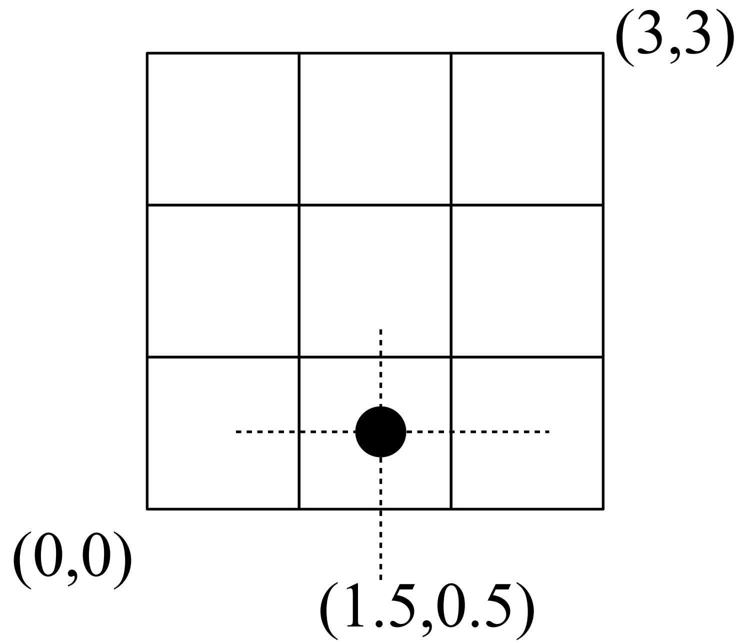
$$
\mathbf{M}_{VP}^{Full} = \begin{bmatrix} 1 & 0 & 0 & W/2 \\ 0 & 1 & 0 & H/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{W}{2} & 0 & 0 & 0 \\ 0 & \frac{H}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5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

- 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 WebGL 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

In Fragment Shader

The location of the pixel is in the built in variable, .xy and depth related values .zw:

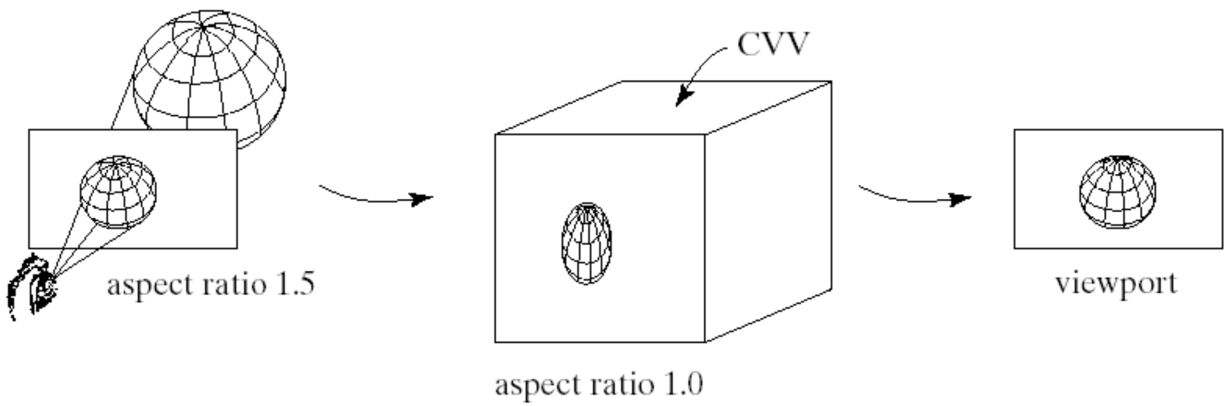
- in vec4 gl_FragCoord
  Aside:
  gl_FragCoord.w = 1 / gl_Position.w
- Again, in what coordinate system ?
  .xy in widow relative coordinates
  .z clipping coordinates (unless a shader changed it)
  .w = 1 / gl_Position.w   from clipping coordinates
- See reference page

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?

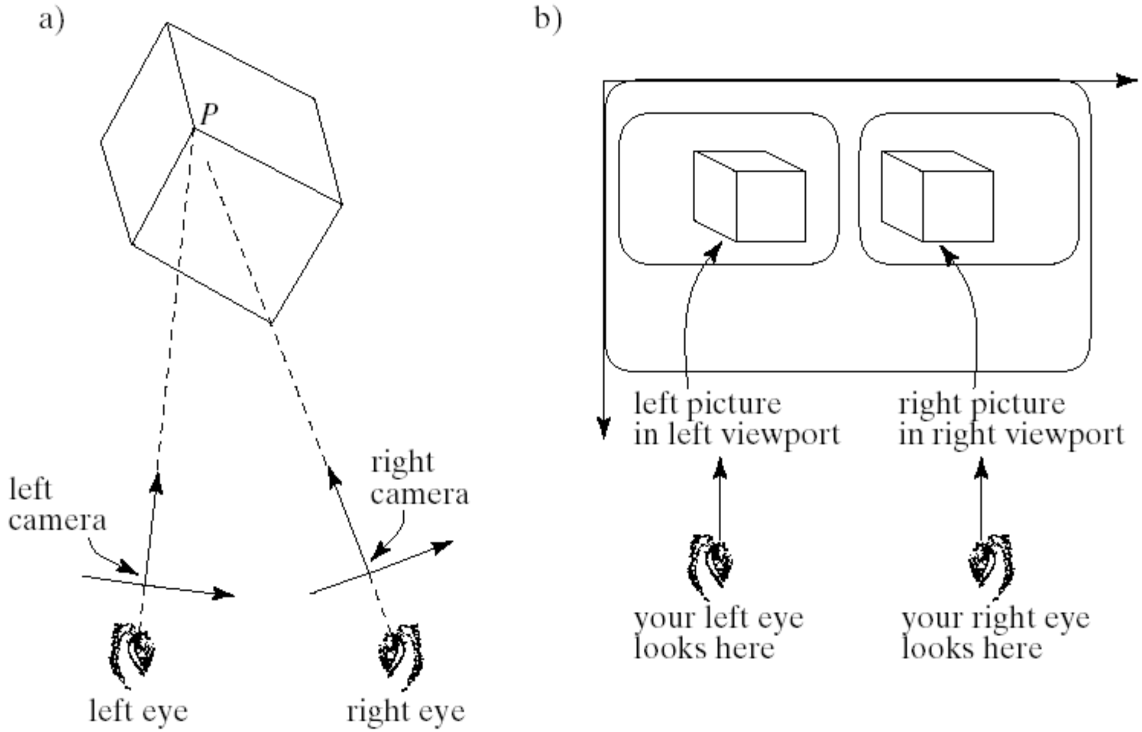
- Undo the distortion of the projection transformation



- Control distortion when window changes aspect ratio or when camera aspect ratio and window aspect ratio are different.
  Exercise: How can you map a square camera viewport to non-square canvas?

Stereo views

Render the scene twice from different points of view



Example: Two viewports

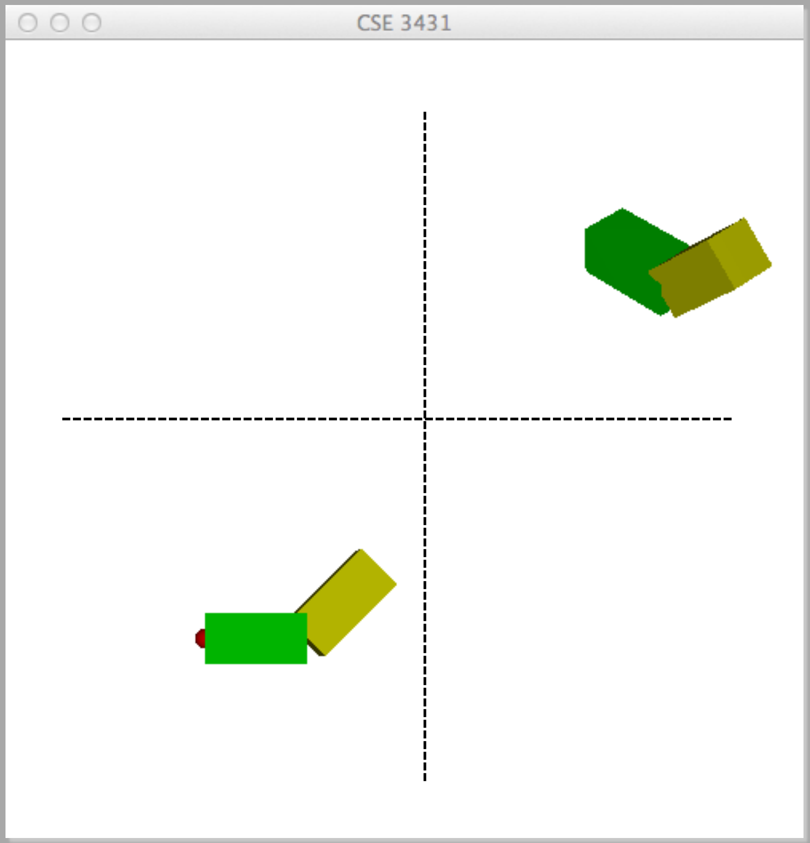
```
void render()
{
     gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

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

     drawObjects() ;
}
```

Example: Two viewports

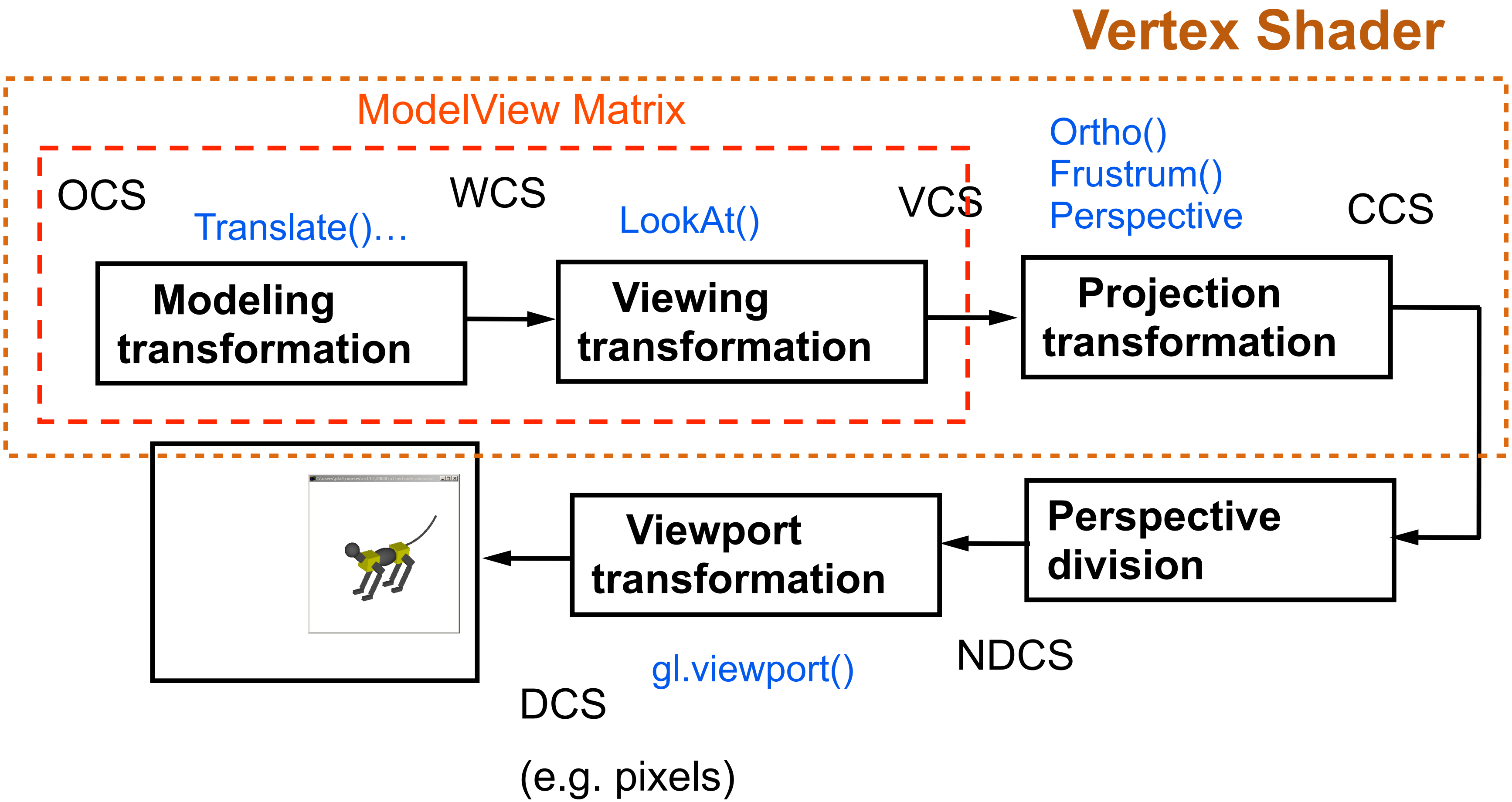
- Viewport one: lower left quadrant
- Viewport one: top right quadrant

Width: 500 pixels

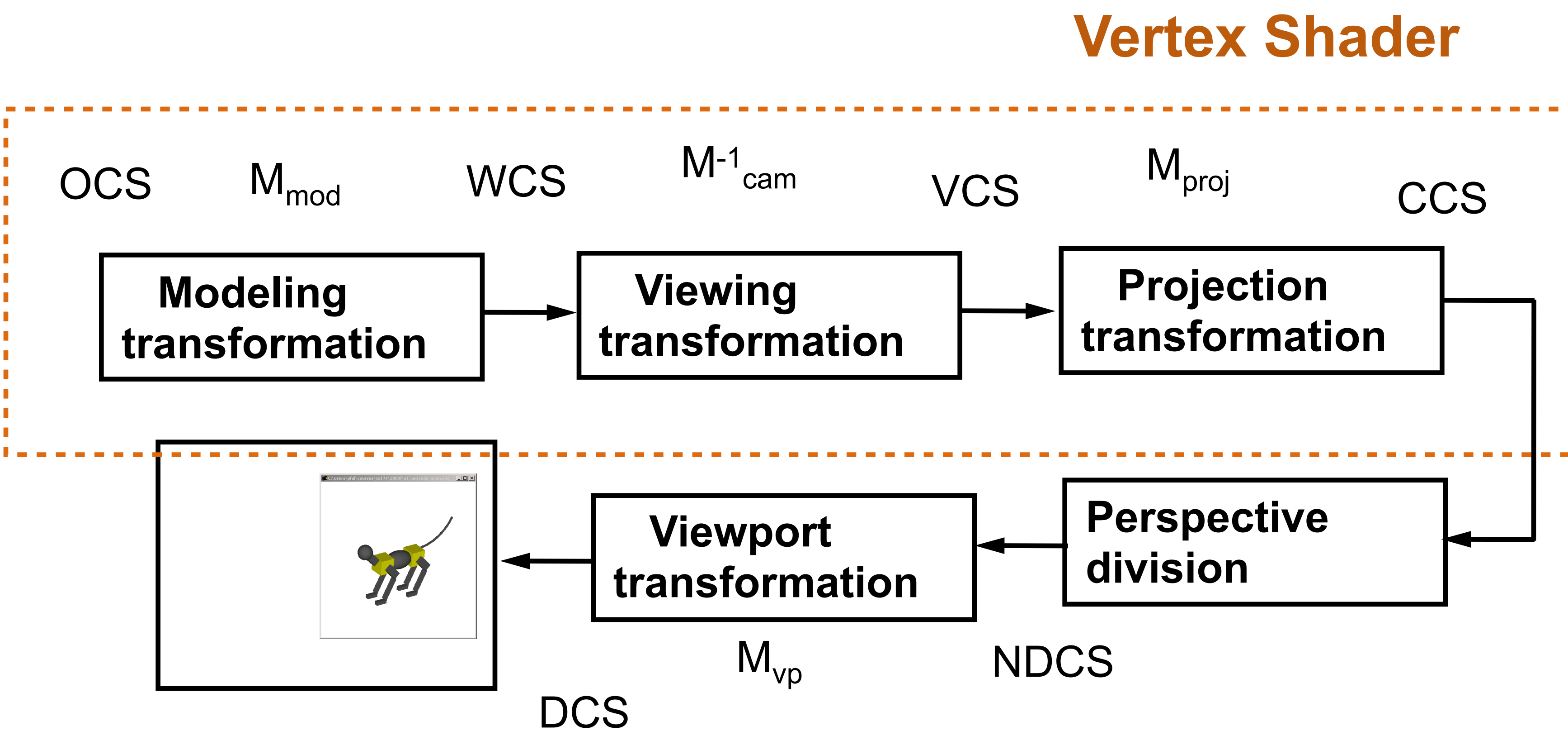
Height: 500 pixels



Transformations in the pipeline



Matrices in the Pipeline



Vertex Shader

```
in vec4 vPosition;
in vec3 vNormal;

uniform mat4 projectionMatrix, modelViewMatrix

out vec4 fColor;


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 CLIPPING Coordinates
```

Line Rendering Algorithm

vertex shader

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

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

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

Compute $\mathbf{M}_{O}$

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

Compute $\mathbf{M}_{proj} = \mathbf{M}_{O}\mathbf{M}_{P}$

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

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

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 4th coordinates of P,Q

end for

rasterizer calls fragment shader