Cory Zimmerman

(Q1)
B: While fov affects the apparent sizes of objects, it doesn't affect their perceived distance in relation to each other. As the front face scales, so does the back face. Because of this, option B, which displays zooming but not distortion, is most plausible.

(Q3: 11.2)
Notice that this is Q3. I'm doing them in book order, not pset order:
Let the projection matrix be defined as

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

With eye coordinates x_e, y_e, z_e, this yields clip coordinates

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_e \\ y_e \\ z_3 \\ 1 \end{bmatrix} = \begin{bmatrix} x_e \\ y_e \\ 1 \\ -z_3 \end{bmatrix}$$

This produces device coordinates

$$\begin{bmatrix} \frac{x_e}{-z_e} \\ \frac{y_e}{-z_e} \\ \frac{1}{-z_e} \\ 1 \end{bmatrix}$$

Using the projection matrix PQ instead generates

$$PQ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -3 & 0 \end{bmatrix}, PQ \cdot \begin{bmatrix} x_e \\ y_e \\ z_3 \\ 1 \end{bmatrix} = \begin{bmatrix} 3x_e \\ 3y_e \\ 1 \\ -3z_e \end{bmatrix}$$

This produces device coordinates

$$\begin{bmatrix} \frac{3x_e}{-3z_e} \\ \frac{3y_e}{-3z_e} \\ \frac{1}{-3z_e} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x_e}{-z_e} \\ \frac{y_e}{-z_e} \\ \frac{1}{-3z_e} \\ 1 \end{bmatrix}$$

Mathematically, it appears that the coordinates projected by PQ have identical x and y values but

are squished closer towards the origin in the z direction compared to coordinates projected by just P. I believe this has the effect of pushing a wider range of z values into the range of what will be rendered, producing the effect of seeing further. In other words, scene elements are less likely to be clipped.

I made a test example available here: `https://github.com/cfzimmerman/S24-CS175/blob/main/assignment-5-cory/mtx_proj.py`. With eye coordinates $(0, 0, -1, 1), (0.5, 0, -0.5, 1), (0, 0.5, 0.5, 1)$, the device coordinates projected by just P are $(0, 0, 1, 1), (1, 0, 2, 1), (0, -1, -2, 1)$, while the device coordinates projected by PQ are $(0, 0, 0.33, 1), (1, 0, 0.67, 1), (0, -1, -0.67, 1)$.

(Q2: 11.3)
Abbreviating some of the steps from the question above, consider the projection by PS:

$$PS = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & -3 & 0 \end{bmatrix}, PQ \cdot \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = \begin{bmatrix} 3x_e \\ 3y_3 \\ 3 \\ -3z_e \end{bmatrix}$$

This produces device coordinates

$$\begin{bmatrix} \frac{3x_e}{-3z_e} \\ \frac{3y_e}{-3z_e} \\ \frac{3}{-3z_e} \\ \frac{-3z_e}{-3z_e} \end{bmatrix} = \begin{bmatrix} \frac{x_e}{-z_e} \\ \frac{y_e}{-z_e} \\ \frac{1}{-z_e} \\ 1 \end{bmatrix}$$

The resulting device coordinates are identical to those after projection by just P. This shows up mathematically because all entries in the output are scaled by 3, and then dividing by w_c eliminates that scale from all entries. Visually, I believe this has the effect of evenly expanding and then condensing our view of the world by the same constant, producing the same picture. I added code for this to the script linked above as well, but, because the outputs are the same for both, they're not very interesting.

(Q4: 11.4)
Again, model mathematically:

$$QP = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = \begin{bmatrix} 3x_e \\ 3y_e \\ 3 \\ -z_e \end{bmatrix}$$

This produces device coordinates

$$\begin{bmatrix} \frac{3x_e}{-z_e} \\ \frac{3y_e}{-z_e} \\ \frac{3}{-z_e} \\ 1 \end{bmatrix}$$

Under this transformation, the x, y, and z coordinates are all scaled by a factor of 3. Elements will appear proportionally larger. With these expanded dimensions (especially the increased z value), fewer points will fit within the clipped viewing region, creating a zooming effect on the scene.

Again using the same script as before, the same eye coordinates $(0, 0, -1, 1), (0.5, 0, -0.5, 1), (0, 0.5, 0.5, 1)$ which map under just P to $(0, 0, 1, 1), (1, 0, 2, 1), (0, -1, -2, 1)$ now map under QP to $(0, 0, 3, 1)$, $(3, 0, 6, 1), (0, 3, 6, 1)$.

(Q5: 12.3)
Again, begin by analyzing this mathematically. For computational simplicity, let $W = 512$ and $H = 256$. With a known viewport matrix V, normalized device coordinates x_n, y_n, z_n, have the following map:

$$\begin{bmatrix} 256 & 0 & 0 & 255.5 \\ 0 & 128 & 0 & 127.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix} = \begin{bmatrix} 256x_n + 255.5 \\ 128y_n + 127.5 \\ 0.5z_n + 0.5 \\ 1 \end{bmatrix}$$

Using QV as the viewport matrix instead places the same coordinates like this:

$$QV = \begin{bmatrix} 768 & 0 & 0 & 766.5 \\ 0 & 384 & 0 & 382.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}, QV \cdot \begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix} = \begin{bmatrix} 768x_n + 766.5 \\ 384y_n + 382.5 \\ 0.5z_n + 0.5 \\ 1 \end{bmatrix}$$

From the comparison above, it seems using QV as the viewport matrix for computing window coordinates scales the x and y window dimensions by 3 without altering the z coordinate dimensions. This is different from question 4. When scaling the projection matrix, changes affected the zoomed perspective of the scene, and we zoomed into the center of it. When scaling the viewport matrix, however, we're just changing the size of the scene on the user's screen. Importantly, the zoom location of origin for the user's screen is almost certainly not the center, so performing this transformation zooms on a different part of the scene (the bottom left corner).

Using the same code, sample eye coordinates, and projection matrix from earlier, the window coordinates using just V are $(255.5, 127.5, 1.0), (511.5, 127.5, 1.5), (255.5, -0.5, -0.5)$. The window coordinates using QV are $(766.5, 382.5, 1.0), (1534.5, 382.5, 1.5), (766.5, -1.5, -0.5)$. These results are consistent with the mathematical approach.

(Q6)
I'll approach this question by mapping to and from window coordinates. For simplicity, assume the window itself is the same size as the texture, and the texture is positioned at the bottom left of the window. That means the texture has a unit bottom left coordinate of $(0, 0)$, a canonical bottom left coordinate of $(-1, -1)$, and a window coordinate of $(-0.5, -0.5)$. Because the canonical square is twice the size of the unit square and translated from the origin, map the given unit

coordinate $(0.45, 0.63)$ by the function to the canonical coordinate $(0.45, 0.63) * 2 - 1 = (0.9, 1.26) - 1 = (-0.1, 0.26)$. Using my code from question 5 on a 512 by 512 screen, that maps to window coordinates $(229.9, 322.06)$. Under the current model, pixels are centered at integer coordinates, with 0.5 window units of area to the left, right, top, and bottom. So, the given texture coordinate will map to the pixel at window coordinate $(230, 322)$. The x canonical coordinate of the pixel center is then at $\frac{230+0.5}{512} * 2 - 1 = -0.0996$, and the y canonical coordinate of the pixel center is at $\frac{322+0.5}{512} * 2 - 1 = 0.2598$. Converting that back to texture coordinates produces $(\frac{-0.0996+1}{2}, \frac{0.2598+1}{2}) = (0.4502, 0.6299)$.

Reasoning on the formulas:

- Unit to canonical: Canonical is twice as large as unit, so a unit coordinate must be multiplied by 2. Canonical is also shifted down and to the left. That produces the unit to canonical mapping $f(u) = 2u - 1$.
- Canonical to window: I used the matrix from question 5.
- Window to canonical: Find the ratio of the distance from the pixel to the axis against that window dimension. Add an extra 0.5 because the origin is at $(-0.5, -0.5)$. Then, multiply that by two for the size of the canonical square, and reposition the origin left and down by 1. That yields $f(w) = \frac{w+0.5}{D} * 2 - 1$ where D is the window dimension.
- Canonical to unit: Add 1 to shift the canonical square origin up to $(0, 0)$. Then, divide by 2 to resize: $f(c) = \frac{c+1}{2}$.

(Q7)
Note: following the structure of the question, my response omits z.

It's given that $f = ax_n + by_n + c$ for NDC x_n and y_n. However, these NDCs will not be properly interpolated into the fragment shader. We know from the derivation of clip coordinates that there exists some w_n such that $x_n w_n = x_c$ and $y_n w_n = y_c$. From the vertex shader, define fragment shader inputs $v_x = x_n w_n, v_y = y_n w_n, v_w = w_n$. These values match the definition of clip coordinates, and OpenGL affinely transforms with respect to eye/clip/object coordinates. So, the interpolated values of v_x, v_y, v_z in the fragment shader will be affinely transformed. The temporary modification must be undone through division by w_n in the fragment shader where all inputs have been affinely transformed. So, with a, b, c defined for f, the correctly interpolated value of f in the fragment shader with these inputs will be $f = \frac{av_x + bv_y + cv_w}{v_w}$.