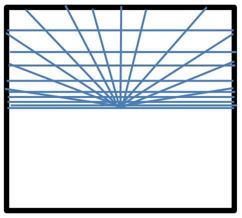
NetID_______________Na **!!!SOLUTION!!** Views of grade 25, 2018

True/False: (copy-and-paste TRUE or FALSE, not just T or F) **(64 pts total – 4pts each)**

__ FALSE*(look-at: 0,0,0) _____1. Suppose we construct a 3D scene that contains a nearly-infinite 'ground plane' made of a grid of lines parallel to x and y axes at z=0 in 'world coordinates.' The ground-plane grid-spacing (the world-space distance between adjacent parallel lines) is 1.0 in both the x and y directions. We then view that 3D scene with a 3D **perspective**-projection camera whose frustum is defined by (left, right, top, bottom == (0,10,0,10) and a very close 'near' clipping plane, and a very far-away 'far' clipping plane. IF we position the camera at (x,y,z) = 0,0,5 in world coordinates, AND IF we set the camera's 'look-at' point anywhere on the ground-plane grid (z=0; any nominal/non-infinite x,y values), THEN we our on-screen result will ALWAYS show a horizon-line on-screen, formed by converging ground-plane lines.



___ FALSE*(view-frustum shape REQUIRES look-at point ABOVE the horizon! Look-at point appears in lower-left corner of on-screen image)____2. Using that same 3D perspective camera, located at that same world-space (0,0,5) position, viewing that same 3D scene of parallel ground-plane lines, we can ALWAYS aim our camera to create an on-screen image of an 'inverted horizon' (like the one I crudely & approximately illustrated here) IF we again restrict our look-at point to a ground-plane location (z=0; nominal/non-infinite x,y values).
Note that ground-plane lines appear ONLY in the top half of the image, and those lines converge to form perfectly-level horizon-line with exactly half the image above the horizon, and half the image below.

____ TRUE(Our look-at point is at lower-left corner. Rotate up vector by 90 degrees CW and you'll see only sky!)___3. Suppose we used the same 3D perspective camera, located at that same world-space (0,0,5) position, viewing the same 3D scene of parallel ground-plane lines, and successfully aimed the camera to produce the 'inverted horizon' image described & crudely illustrated in the previous question. CLAIM: We can make all grid-lines vanish from the on-screen displayed image by adjusting the 'UP' vector. True or false?

__ FALSE*(camera position is in world coordinates; +x moves 'world' leftwards in upright camera's image, and rightwards in upside-down camera's image) _____4. Suppose we again used the same 3D camera at world-space (0,0,5), the same 3D scene, and the same camera aiming described in the previous question. Once again, we see the 'inverted horizon' image described and illustrated above. CLAIM: if we translate the camera in the +x direction without rotating the camera at all, then the grid-lines move LEFTWARDS in the on-screen image (-x direction in screen coordinates)

__ TRUE; only rotations change vanishing points_____5. As shown in the crude illustration, an artist would say that the ground-plane grid has a 'vanishing point' at the center of the on-screen image (see Lecture Notes & reading). IF we translate the camera coordinate system without any rotation (its U,V,N vectors don't change), THEN the vanishing point location in on-screen image DOES NOT CHANGE. This is true for ANY translation direction and

___ TRUE(view-volume is always 10x10; holds >=10 x 10 lines for look-at of (0,0,0), and more for look-at further away. 'up' vector COULD put view volume entirely ABOVE the eye->lookAt line, but view volume STILL intersects ground-plane!) _____6. Suppose we instead capture that same 3D ground-plane scene with a 3D **orthographic**-projection camera, a camera that also has a very nearby 'near' clipping plane, and a very far-away 'far' clipping plane, and (left, right, top, bottom) values of (0,10,0,10). If we position the camera at (x,y,z) = 0,0,5 in world coordinates and set its look-at point anywhere on the ground-plane (z=0), then our on-screen result will ALWAYS be completely covered by a grid-like-feature, one that shows at least 10 non-overlapping lines in one direction, and at least 10 more non-overlapping lines drawn in a visibly different direction. HINT: don't forget the 'up' vector!

___ TRUE (aim camera parallel to ground plane (e.g. any look-at point (x,y,5) with nonzero x,y), with camera turned sideways (e.g. look-at (0,1,5); up vector (1,0,0)) the off-center viewing volume gets split by the ground-plane; ____7. Now suppose we view that same 3D scene with that same 3D **orthographic** projection camera with the same very nearby 'near' clipping plane, and same very far-away 'far' clipping plane. If we position the camera at (x,y,z) = 0,0,5 in world coordinates, then we can ALWAYS find a camera aiming direction where the entire ground-plane appears on-screen as a single vertical line that passes through the exact center of the on-screen image.

___ TRUE (take the answer from previous question, rotate 'up' vector to -Z direction (0,0,-1), Again, the off-center viewing volume gets split by the ground-plane. ____8. Now suppose we view that same 3D scene with that same 3D **orthographic** projection camera with the same very nearby 'near' clipping plane, and same very far-away 'far' clipping plane. If we position the camera at (x,y,z) = 0,0,5 in world coordinates, then we can ALWAYS find a camera aiming direction where the entire ground-plane appears on-screen as a single horizontal line that passes through the exact center of the on-screen image.

___ FALSE ____9. For our textbook's cuon-matrix.js library (which exactly mimics the OpenGL functions gluPerspective(), glFrustum(), glOrtho), the `setPerspective()` and `setFrustum()` functions make camera matrices that form a viewing frustum in the +Z half space: these cameras 'gaze down the +z axis'.

___ TRUE (by definition. See Scene Graph)____10. The 'viewing' transformation, no matter how it is made, converts the 'Eye' or 'camera' coordinate system or **drawing axes** into the 'world' coordinate system or **drawing axes**, *e.g.* the world coordinate system gets 'pushed out from' eye coordinate system. Equivalently, the viewing transformation matrix converts vertex coordinate numbers from their 'world-space' numerical values to their 'eye-space' numerical values.

___ FALSE(NO!!!! backwards! Takes the camera frustum/view volume contents, transforms to fit into the CVV) ___ 11. The 'projection' transformation, no matter how it is made, converts the coordinate-value contents of a +/-1 cube centered at the origin of the 'Eye' or 'camera' coordinate system so that they fill a 'camera-viewing volume' or 'camera frustum'.

____ FALSE(NO!!!! backwards!___12. Any matrix (and thus any viewing frustum) created by a call to the textbook's cuon-matrix.js function `setFrustum()` can also be created by a call to the `setPerspective()` function. However, `setPerspective()` can create matrices that the `setFrustum()` function cannot match.

_____ TRUE (by definition. See Scene Graph)__13. In our WebGL programs that use separate model, view, and projection matrices, the 4x4 matrix created by the `setPerspective()` function is usually the LAST matrix applied to vertex position coordinate values before the vertex shader sends them onwards to the fragment shader.

___ TRUE (see 'Pseudo-depth' in readings.____14. Orthographic camera matrices created by the `setOrtho()` function will simply scale and translate z-coordinates, ***without*** distorted z values, unlike the `setFrustum()` function.

(For 15 and 16): Suppose your convert your Project B WebGL program to use the 'frustum()' function instead of 'perspective()' to create its projection matrix, and replace the side-by-side, two-camera-view display with just a single camera's image. As before, when users re-size their web-browser your WebGL program keeps the browser window filled, but does not distort the displayed 3D perspective image (does not 'squash' or 'stretch' the on-screen picture or its contents).

___ FALSE(You did it yourself in Project B! Top/Bottom stay same to keep same fovy; only left/right changed____15. For an arbitrary browser re-sizing that changes width & height by different non-zero amounts, your revised Project B then MUST change the values of all 4 of these `frustum()` arguments: left, right, top, bottom – none of them remain unchanged.

___ FALSE(We don't need to call viewport() at all—its default values will fill the canvas) ____16. For an arbitrary browser re-sizing that changes width & height by different non-zero amounts, your revised Project B then

MUST call **gl.viewport()** and use it to change the default viewport matrix contents.
HINT: Think carefully. Is this call absolutely necessary? We only have 1 camera now, not two!

Suppose a WebGL program includes a Vertex-Buffer Object (VBO) that contains a long list of vertices with position and color attributes. If we draw the buffer's contents using the WebGL **`GL_LINES`** drawing primitive with model, view, and projection matrices set for drawing on-screen using with the 'world' drawing axes, we get a vast, seemingly endless grid of lines in the x=0 plane. It draws world-space lines parallel to the y axis at z=0, +/-10, +/-20, +/-30, … etc., and world-space lines parallel to the z-axis at x=0, +/-10, +/-20, +/-30, … etc. THEN:

17 a) **(4 points)** We want the buffer's contents on-screen (**`GL_LINES`** drawing primitive again) to appear as a 'ground plane,' but we must somehow draw its lines in the world-space z=0 plane, not the x=0 plane. What is the best, most-sensible action to take before just before draw this buffer's contents? (HIGHLIGHT YOUR ANSWER)

a) change both the 'model' and the 'view' matrix

b) change both the 'view' and the 'projection' matrix

d)push the 'view' matrix onto stack, change only the view matrix, draw lines, pop the view matrix.

[✓ c) push the 'model' matrix onto stack, change only the model matrix, draw lines, pop the model matrix.]

e) change none of the matrices; instead, you MUST change the vertex-buffer contents.

f) Something else – none of the above.

17 b) **(4 points)** Which function call (or function calls) will you apply to the matrix (or matrices) you selected in 4a) so that the grid's +x direction matches the world coordinate systems' +z direction?
(HIGHLIGHT YOUR ANSWER(S): If more than one matrix, you may need to circle more than one answer.)

a) myMatrix.rotate(90.0, 1,0,0); // +90 degree x-axis rotation,

b) myMatrix.rotate(90.0, 0,1,0); // +90 degree y-axis rotation,

c) myMatrix.rotate(90.0, 0,0,1); // +90 degree z-axis rotation,

d) myMatrix.rotate(90.0, -1,0,0);

[✓ e) myMatrix.rotate(90.0, 0,-1,0); **// -90 degree y-axis rotation**]

f) myMatrix.rotate(90.0, 0,0,-1);

g) Something else – none of the above.

For all parts of problem 18, write q = (x,y,z,w) to represent the quaternion q = w + x**i** + y**j** + z**k**.
HINT 1: see lecture notes posted on Canvas, and Lengyel readings assigned on quaternions.
HINT 2: You have starter code that can compute and print your answers to basic quaternion calculations.
You may wish to use the 'printMe()' functions you find there for Quaternion and Matrix objects.

18 a) **(4 points)** What is the length (the magnitude) of this quaternion?
(numbers only, please—no algebra or expressions!)

qa = (2,3,4,-2) LENGTH == **+5.74456**

18 b) **(4 points)** This quaternion has magnitude or length of 5. Find it's normalized version:
(numbers only, please—no algebra or expressions!)

qb = (0,3,0,4). NORMALIZED qb = (0 , 0.6 , 0 , 0.8)

18 c) **(4 points)** Find the unit-length quaternion that results from rotation of -90 degrees around the 0,0,1) axis: (numbers only, please—no algebra or expressions)

q1 = (0 , 0. , -0.707107, 0.707107)

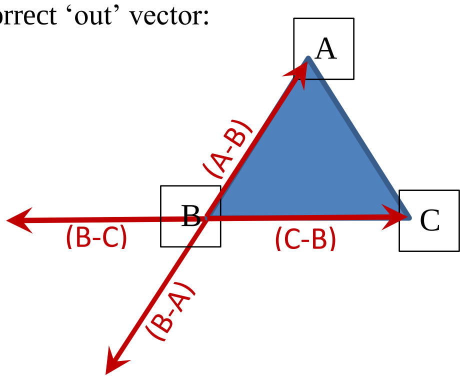
18 d) **(4 points)** Find the unit-length quaternion that results from rotation of 46 degrees around the (-3,2,1) axis: (numbers only, please—no algebra or expressions!)

q2 = (-0.31328, 0.208855, 0.104427, 0.9205047)

19(8 points) Normal Vector for a Triangle: Suppose we write a 3D triangle's vertices symbolically, as points A,B, and C, given in CCW (counter-clockwise) order. We can then calculate a vector called 'out' that is perpendicular to the front face of that triangle using the cross-product operator 'x' and vectors along the edges of the triangles. HIGLIGHT ALL of the expressions below (if any) that compute a valid & correct 'out' vector: (careful! No credit for sign errors!)

a) Out = (A-B)x(B-C)
b) Out = (A-B)x(C-B)
c) Out = (B-A)x(B-C)
d) Out = (B-A)x(C-B)
e) Out = (C-B)x(B-A)
f) Out = (B-C)x(B-A)
g) Out = (C-B)x(A-B)
h) Out = (C-C)x(A-B)

Correct sign, wrong magnitude.
Wrong sign, correct magnitude.
Wrong sign, correct magnitude.
Correct sign, wrong magnitude.
Wrong sign, wrong magnitude.
YES!
YES!
(C-C)==zero length! (typo) OR: (*B*-C)x(A-B): Wrong sign, wrong magnitude



20(4 points) Fill in A, B, or C in the blanks below to create another, different yet also-correct expression for the 'out' vector that is not already listed in the answer to question 19:

Out = (______ - ______) x (_____ - _____)

Question 19 explored all choices for edge-vectors from vertex B, and shows just 2 correct results exist. Similarly:
For vertex A, correct forms are:
(B-A)x(C-A)
(A-B)x(A-C)
For vertex C, correct forms are:
(A-C)x(B-C)
(C-A)x(C-B)
(No other forms are correct!)