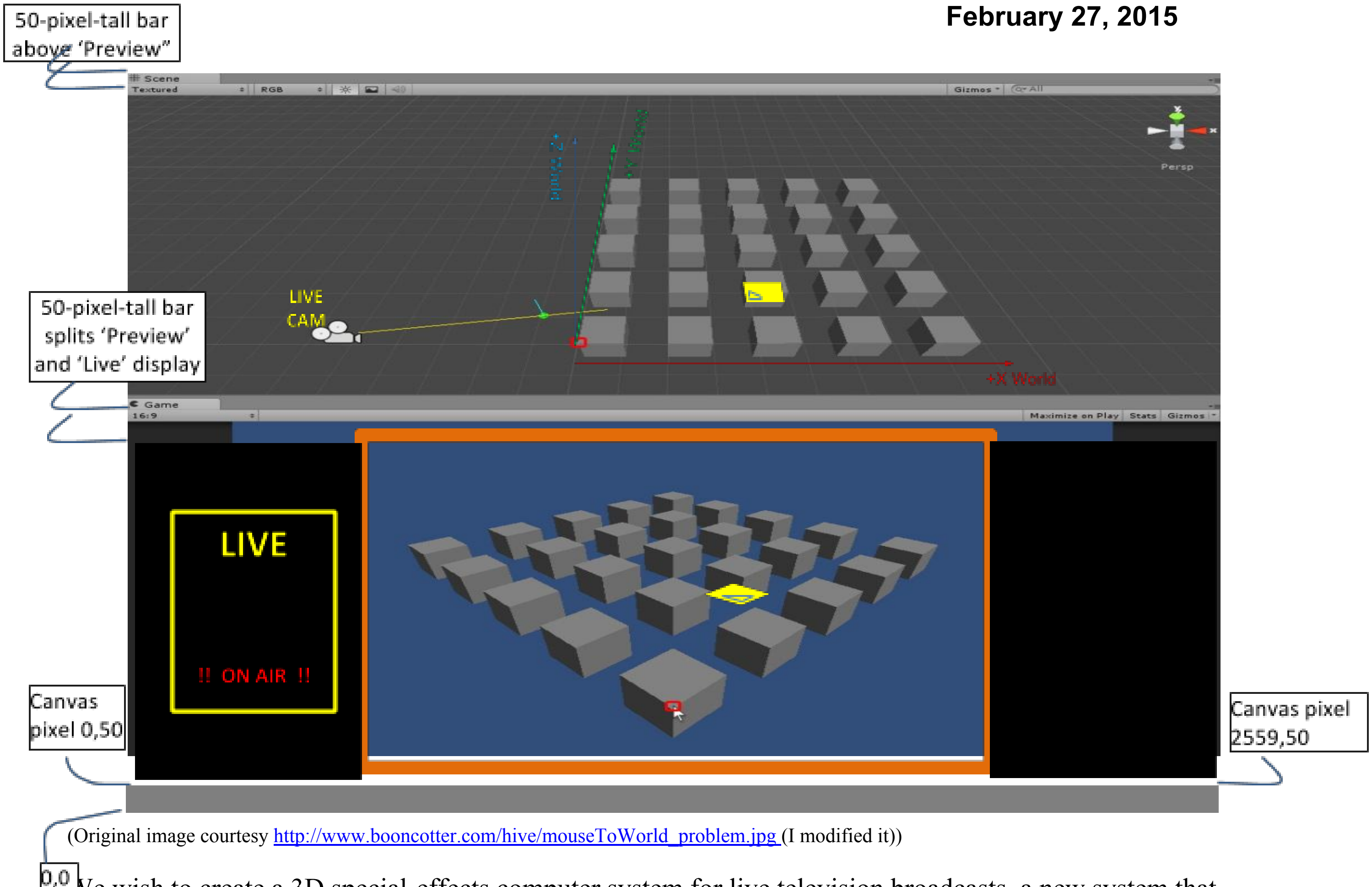
NetID_spb520__Name_Sean Bowen-Williams

Test B: Moving 3D Views
100 pts total, 7% of grade
February 27, 2015



(Original image courtesy http://www.booncotter.com/hive/mouseToWorld_problem.jpg (I modified it))

We wish to create a 3D special-effects computer system for live television broadcasts, a new system that usesWebGL to render synthetic 3D images like this one. The system fills a giant 2560x1600-pixel display to show two different 3D images from two different 3D cameras aimed at the same 3D scene.

- The 'preview' image at the top lets users construct an animated 3D scene and move a 'live' camera through that scene. The 3D perspective 'preview' camera has an unusually wide field-of-view, forms an undistorted preview image, made with a precisely-square grid of pixels on the camera's 'plane that splits the universe' (review your lecture notes and 2015.02.14.3DviewCamera04.pdf on Canvas). Equivalently, the preview camera's aspect ratio exactly matches its displayed-image aspect ratio (width/height = 2560/730).
  -- Preview-camera displayed image size: **2560 x 730 pixels (careful! does not match live-camera height!)**
  -- Preview-camera displayed image's **upper-right corner pixel is at address (2559,1549)** in the WebGL pixel coordinates, where lower-left corner pixel has address (0,0)).
  -- A 50-pixel-tall grey menu-and-tabs bar separates the preview-camera image from the top of the screen.
  -- Preview-camera FOV (edge-to-edge): **82.330 degrees wide, 28.000 degrees tall.**
  -- Preview-camera frustum is symmetrical: **left= -right, bottom = -top.** Its 'near' clipping plane is positioned in the 'camera' or 'eye' coordinate system at **z = -0.5**, and its 'far' plane at **z = -9987.0**.
  The preview image contains a 'LIVE CAM' icon that users can adjust to change the position and aiming of the live-camera as it gathers the 'live' image of the 3D scene.

- The live-camera displayed image, shown inside the orange rectangle, exactly matches (one of) the ATSC standards for HDTV broadcasting in the United States: exactly 1280 x 720 pixels. It was made by a 3D perspective camera that forms images with a 16/9 aspect ratio (1280pixels /720 pixels), and creates a grid of precisely-square pixels (the width of a pixel == the height of a pixel).

-- live-camera displayed image size: **1280 x 720 pixels (careful! does not match 'preview-camera' height!)**

-- live-camera displayed image's **upper-right corner pixel is at address (1919,769)** in the WebGL pixel coordinates, where lower-left corner pixel has address (0,0)).

-- A 50-pixel-tall grey menu-and-tabs bar separates the live-camera image from the preview-camera image

-- A 50-pixel-tall grey status bar separates the live-camera image from the bottom of the display screen.

--Live-camera FOV(edge-to-edge): **53.2570 degrees wide, 31.500 degrees tall.**

-- Live-camera frustum is symmetrical: **left= -right, bottom = -top.**

-- Live-camera 'near' clipping plane is positioned in the 'camera' or 'eye' coordinate system at **z = -1.3**, and its 'far' clipping plane is positioned in the 'camera' or 'eye' coord. system at z = **-4800.5**.

The special-effects computer system displays its images inside a specialized HTML-5 webpage that causes the web-browser to fill our entire giant 2560x1600-pixel LCD display with one giant, borderless canvas element of 2560x1600 pixels on-screen. Our WebGL program then writes its 3D WebGL images inside this canvas.

1 a) (4 pts) Write the gl.viewport() function call that will set the correct position and size of the **'Preview-camera'** display image on-screen, specified in pixels. **(Numbers only! no operators or expressions allowed in your answer!)**

```
gl.viewport( 0,769,2559,730);
```

1 b) (4 pts) Write the gl.viewport() function call that will set the correct position and size of the **'Live-camera'** display image on-screen, specified in pixels. **(Numbers only! no operators or expressions allowed in your answer!)**

```
gl.viewport(0,50,1279,769);
```

2 a) (4 pts) Complete this setPerspective() function call required to correctly specify the Preview-Camera frustum using our textbook's cuon-matrix.js library**: (Numbers only! no operators or expressions in your answer!)**

```
projPrvw.setPerspective(28.0,3.5, 0.5,9987.0);
```

2 b) (4 pts) Complete this setPerspective() function call required to correctly specify the 'Live-Camera' frustum using our textbook's cuon-matrix.js library**: (Numbers only! no operators or expressions in your answer!)**

```
projLive.setPerspective(31.5, 1.78, 1.3, 4800.5);
```

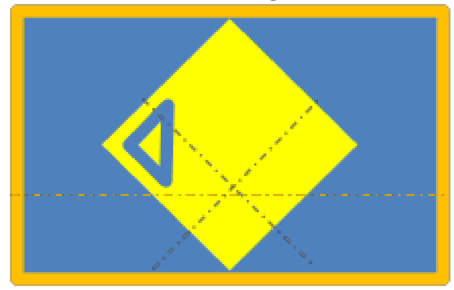
The 'preview-cam' image in our 3D special-effects system marks the 'world' coordinate system origin point (0,0,0) with a small red square, and its +x, +y, and +z axes with red, green, blue arrows respectively. It also marks the z=0 plane with an endless 'ground-plane' grid of light-grey lines in the z=0 plane. The grid holds:

--lines parallel to the +x axis (at y=0, +/-10, +/-20, +/-30, etc)
that converge towards the horizon past the top of the preview image, and

--lines parallel to the +y axis (at x=0, x=+/-10, x=+/-20, x=+/-30, … etc).

The preview image also shows that these 3D cube shapes extend BELOW the ground plane, with their top faces in the z=0 plane, and bottom faces at z=-10. One cube has a yellow top-face with a blue triangle in the z=0 plane.

3 a) (18 points) How should you position and aim the 'live' camera to exactly recreate this 'live' image of that face?



Complete this setLookAt() function call for the 'live' camera view matrix.
Note that:
-- the 10x10 yellow cube face is perfectly square, and also appears perfectly square in this image (camera aimed perpendicular to the yellow cube-face).
--the yellow cube face's center point is perfectly aligned with the 'Live-Camera' image center-point. The yellow cube face's corners touch the top and bottom edges of the 'live' image, at the exact center of those edges.
Also note the position of the blue triangle shown in the 'preview' image and the 'live' camera image at right.

(the dotted lines show centerlines:
they are not part of the image)

(numbers only, please—no algebra or expression s!)

```
viewLive.setLookAt(_45_______,__25_____,_25_______, //
eye,
                    ___45_____,__25______,___0_____, // at,
                    __-1______,___1_____,___0_____);// up.
```

3 b) **(4 points)** From the camera position specified in 3 a), what is the distance from the camera's center-of-projection (COP) in 'world' coordinates and the center of the one 3D cube with the yellow face? (numbers only, please)

```
Distance = 25
```

Suppose the special-effects systems' graphics processing unit (GPU) holds 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 with the 'world' drawing axes, we get a vast, seemingly endless grid of lines in the y=0 plane. It draws world-space lines parallel to the x 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:

4 a) **(4 points)**We want to draw this buffer's contents on-screen (**`GL_LINES`** drawing primitive again), to create the 'ground plane' for our 3D special effects system, but we must somehow draw its lines in the z=0 plane, not the y=0 plane. What is the best, most-sensible action before drawing 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

c) change both the 'model' and the 'projection' matrix

d) change the 'model' matrix

e) change the 'view' matrix

f) change the 'projection' matrix

g) change all 3 matrices

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

4 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 +y 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); // +90 degree x-axis rotation,

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

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

g) OTHER: ____________________(specify)

For all parts of problem 5, 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: Extend the starter code program 5.04jt.ControlQuaternion to compute your answers using the cuon-matrix-quat.js library. You may wish to use the 'printMe()' functions you find there for Quaternion and Matrix objects.

5 a) **(3 points)** What is the length (the magnitude) of this quaternion?
(numbers only, please—no algebra or expressions!)

qa = (5,4,3,2) LENGTH == __7.35_________________

5 b) **(3 points)** This quaternion has magnitude or length of 5. Find it's normalized version:
(numbers only, please—no algebra or expressions!)

qb = (3,0,0,4). NORMALIZED qb = (_.6_______, ___0.0_____, ____0.0____, ___0.8_____)

5 c) **(8 points)** Which of these quaternions, if any, have NOT been normalized? Given in (x,y,z,w) order:
(HIGHLIGHT ALL YOUR ANSWERS)

a) (1,1,1,0)

b) (1,0,0,1)

c) (0,0,0,1)

d) (0,1,0,0)

e) (0.5, sqrt(2)/2, 0, 0.5);

f) (sqrt(3)/3, 0, sqrt(3)/3, sqrt(3)/3)

g) (0.5, 0.5, 0.5, 0.5)

h) (0.5, 0.5, 0, 1)

5 d) **(3 points)** Find the unit-length quaternion that results from rotation of 20 degrees around the (-1,1,2) axis: (numbers only, please—no algebra or expression s!)

q1 = (__-0.071______, _0.071_______, ___.142_____, __.985______)

5 e) **(3 points)** Find the unit-length quaternion that results from rotation of 46 degrees around the (-2,5,6) axis: (numbers only, please—no algebra or expressions!)

q2 = (__-.097_____, ___.242_____, ___.291_____, __.920______)

5 f) **(4 points)** If we apply the rotation described by quaternion q1 to the 'world' coordinate axes to make new drawing axes A1, then apply the rotation described by quaternion q2 to axes A1 to make new drawing axes A2, how can we find the quaternion qTot that will rotate 'world' drawing axes to make drawing axes A2?

HIGHLIGHT ONE ANSWER ONLY:

a) qTot = (q1 A2 $q1^{-T}$)*(q2 A1 $q2^{-T}$)   (where * means 'quaternion multiply')

b) qTot = q1 * q2

c) qTot = (q1 M $q1^{-T}$)*(q2 M $q2^{-T}$)

d) qTot = q2 * q1

e) QTot = (q1 A1 q2)*(q2 A2 q1)

f) OTHER: _(write your answer):________________________

True/False: (paste your highlighted answer: TRUE or FALSE,) **(30 pts total – 3pts each)**

__TRUE_____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'. We then view that 3D scene with a 3D **perspective** projection camera with 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, then we can ALWAYS find a camera position and aiming direction that will show a horizon-line formed by converging ground-plane lines.

___FALSE____2. 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'. We then view that 3D scene with a 3D **orthographic** projection camera with a very nearby '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, then we can ALWAYS find a camera aiming direction that will show a horizon-line formed by converging ground-plane lines.

___TRUE____3. By default, WebGL always puts the center-of-projection (COP) of perspective cameras at eye-space origin.

___FALSE____4. 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' (NOT the -z axis).

__FALSE_____5. The 'viewing' transformation, no matter how it is made, converts the 'Eye' or 'camera' coordinate system **axes** into the 'world' coordinate system **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.

___TRUE____6. If we use our textbook's cuon-matrix.js function **`setLookAt()`** function (which exactly mimics the OpenGL function gluLookAt()) to implement the 'glass-tube' analogy for camera navigation we discussed several times in class, we will set the function's 'centerX, centerY, centerZ' parameters to the location of the center of the 'glass tube' that encloses our camera's eyepoint.

___FALSE____7. Any matrix (and thus any viewing frustum) created by a call to the textbook's cuon-matrix.js function **`setPerspective()`** can also be created by a call to the **`setFrustum()`** function. However, **`setFrustum()`** can create matrices that the setPerspective() function cannot match.

___FALSE____8. In our WebGL programs that use separate model, view, and projection matrices, the 4x4 matrix created by the **`setLookAt()`** function is usually the LAST matrix applied to vertex position coordinate values before the vertex shader sends them onwards to the fragment shader.

__TRUE_____9. Perspective camera matrices created by **`setFrustum()`** and **`setPerspective()`** functions will compute non-linear 'pseudo-depth' values for z in the CVV. Orthographic camera matrices created by the **`glOrtho()`** command simply scale and translate z-coordinates, ***without*** nonlinear distortion of z.

___TRUE____10. If our WebGL program calls **`setPerspective()`** to create its 'projection' matrix, then we have defined a camera whose view frustum forms a truncated, 4-sided pyramid (JTumblin: this previous statement is true. Are BOTH of next 2 sentences true?). 'Truncation' means the pyramid's peak was 'cut away' by the "near" clipping plane. The location of the pyramid's missing peak is also the camera's center-of-projection.