CS351-1 Introduction to Computer Graphics

Test A: Shape
100 pts max.
Feb 02, 2018

NetID______________Name____
(netID is 6 letters + 6digits, e.g. jet861)

INSTRUCTIONS: Edit this file in Microsoft Word or in Google Docs to enter your HIGHLIGHTED answers. Upload your own file on Canvas before the end of the day Sunday, Feb. 04, 2018, 11:59PM.

Corrected typos on problems 6 &15 shown in blue highlight

GUI and User Interface:

HIGHLIGHT your choice to mark your answer.

1) (5 pts) What are the dimensions of the 'Canonical View Volume' (CVV) in OpenGL and WebGL?

A) Fixed—a unit cube centered at the origin whose x,y,z values span (+/-1, +/-1, +/-1)
B) Fixed—a unit cube, with x,y,z origin shown at the upper-left of display: (0 <= x,y,z <= 1)
C) Fixed—an on-screen rectangle whose limits vary and depend on the modelMatrix contents
D) adjustable—x,y origin set at upper left, and x,y max values set by canvas width and height
E) adjustable—a unit cube centered at the origin
whose visible x,y,z values span(+/- width/2, +/-height/2, +/-1)
F) Something else; none of the above.

GPU Communication:

2) (24pts) TRUE/FALSE: (copy-and-paste your choice of these highlighted answers "True" or "False")

a) _False__WebGL ***requires*** users to specify all vertex positions using real values (floats). This requirement ensures that limited precision won't introduces rendering flaws on-screen.

b) _False__GLSL supplies a standard set of functions that can create a 4x4 matrix for translation, for rotation, or for scale, each from a single function call.

c) _False_ Drawing commands for WebGL and drawing commands for HTML5 'canvas' elements share the same on-screen drawing axes; both span +/- 1, with origin at center.

d) _False_ ... color, buf

https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/getParameter
& look at list on left side of this page: browse all functions that begin with 'get' (e.g. getActiveAttribute() ...)

e) _False__ A 'Fragment Shader' is optional; without it, your WebGL/HTML5/JavaScript program can still draw single-color WebGL drawing primitives on-screen (e.g. TRIANGLES).

f) _True__ With the proper selection of 'stride' and 'offset', WebGL can render the contents of a vertex buffer object (VBO) that holds 100 vertex positions, followed 100 vertex colors, followed by 100 vertex surface normals. In this VBO, the vertex attributes are NOT interleaved!

g) _False__ WebGL prevents use of the same 'uniform' variable to send values to both the Vertex Shader and Fragment Shader. If JavaScript sets its value, only one shader can use it.

h) _False_WebGL itself provides built-in functions for mouse, keyboard, and window-system interactions. We use HTML and JavaScript functions instead because they are more convenient.

Vector-Matrix Math:

In a WebGL program of the sort developed and described in your textbook (e.g. starting with Week02 Vector Matrix Tests, Ch2, 'HelloMatrixOps.js', w/ cuon-matrix-quat.js library)
we find two 'Vector4' objects (or 'variables') named 'aVec' and 'bVec', and
one 'Matrix4' object named 'aMat' that holds this 4x4 matrix:

```
[a b c d]
[e f g h] == aMat
[j k m n]
[p q r s]
```

3) **(4 pts)** If we call `aMat.setTranslate()`, then apply `aMat` to transform `aVec` into `bVec` like this: `bVec = aMat.multiplyVec4(aVec);` then the new value of `bVec.elements[1]` must be:
   A) `j*aVec.elements[0] + k*aVec.elements[1] + m*aVec.elements[2] + n*aVec.elements[3];`
   B) `b*aVec.elements[0] + f*aVec.elements[1] + k*aVec.elements[2] + q*aVec.elements[3];`
   C) `e*aVec.elements[0] + f*aVec.elements[1] + g*aVec.elements[2] + h*aVec.elements[3];`
   D) `c*aVec.elements[0] + g*aVec.elements[1] + m*aVec.elements[2] + r*aVec.elements[3];`
   E) something else happens; none of the above.

3--C --efgh

4) **(4 pts)** For that same translation matrix `aMat`, when `aVec.elements[3]` value is 1.0, then:
   A) All four elements of `bVec` result will always be 1.0, for any and all translations.
   (e.g. `bVec.elements[0]=bVec.elements[1]=bVec.elements[2]=bVec.elements[3]=1`).
   `lements[0]` result will be 1.0; all other elements vary with translation amount.

4--C
all but [3]

   `ents[3]` result will be 1.0; all other elements vary with translation amount.
   ro translation distances, all of the elements of bVec will vary.
   E) Something else happens; none of the above.

5) **(4 pts)** For that same translation matrix `aMat`, when `aVec.elements[3]` value is 0.0, then:

5--E something else;
bVec == aVec: only POINTS can translate!

   0.0 for any and all translation amounts.
   `ements[2]=bVec.elements[3]=0`).
   ments vary with translation amount.
   other elements vary with translation amount.
   D) For all non-zero translation distances, all of the elements of bVec will vary.
   E) Something else happens; none of the above.

(SURPRISE! bVec == aVec, because you can't translate a vector (whose w==1, e.g. elements[3]=0))

6) **(4 pts)** If we call `aMat.setScale()` then apply `aMat` to transform `aVec` into `bVec` like this: `bVec = aMat.multiplyVec4(aVec);` and then find that `bVec.elements[1]` == 0 despite an non `lements[1]`, then we know something about the `aMat` matrix:

6--A-sam. Be sure you know how to make a 'scale' matrix!

   lements are nonzero; '`s`', '`a`', and '`m`'
   ents are nonzero; '`m`', '`f`', and '`s`'
   ments are nonzero; '`s`', '`a`', and '`f`'
   rix elements are nonzero; '`m`', '`f`', and '`a`'
   E) Something else happens; none of the above.

7) **(4 pts)** If we call `aMat.setRotate(-90,0,0,-1);`then apply `aMat` to transform `aVec` into `bVec` : `bVec = aMat.multiplyVec4(aVec);` we can be certain that:

A) ... `ents[0]` equals `aVec.elements[0]`

7--C: -y axis
rotates to +x

B) ... `]` equals -`aVec.elements[0]` (note minus sign)
C) ... equals `aVec.elements[1]`
D) ... `s[0]` equals -`aVec.elements[1]` (note minus sign)
E) `bVec.elements[0]` equals `aVec.elements[2]`
F) `bVec.elements[0]` equals -`aVec.elements[2]` (note minus sign)
G) Something else happens; none of the above.

8) **(4 pts)** For that same rotation matrix, when `aVec.elements[3]` value is 0.0, then:
A) All four elements of `bVec` result will always be 0.0 for any and all rotations we specify. (e.g. `bVec.elements[0]=bVec.elements[1]=bVec.elements[2]=bVec.elements[3]=0`).
B) Only ... `Vec.elements[1] == bVec.elements[1]`

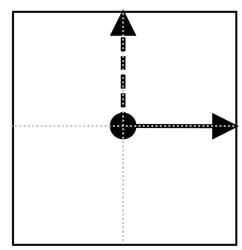
8--E – aVec is NOT a point, but a vector;
yet it DOES rotate, but around z axis:
thus the z value is fixed.

C) ... `ents[0] == bVec.elements[0]`
D) ... `ts[3] == bVec.elements[3]`
E) ... `ents[2] == bVec.elements[2]`
F) Something ...

Matrix Duality & Scene Graphs

Suppose that:

- Our HTML5 Canvas is square on-screen (height==width), and it displays WebGL output.
- We wrote a Javascript `drawAxes();` function that causes WebGL to:
  --draw a solid arrow from the origin to (+1,0,0) to depict the x axis, and
  --draw a dashed arrow from the origin to (0,+1,0) to depict the y axis.
  --draw a large, solid round 'dot' at the origin
- In JavaScript we send the 4x4 'modelMatrix' as a uniform to the GPU,
- Our Vertex shader applies that uniform matrix to transform all vertex position attributes before drawing them.
- This code: `modelMatrix.setIdentity(); drawAxes();`
- Causes our program to draw the picture shown, with both arrows drawn entirely within the canvas.



If our program executes this sequence of statements instead, smoothly varying 0 <= myAngle <=90°

```
modelMatrix.setTranslate(-0.5,-0.5,-0.5);   // 3D move
modelMatrix.scale(0.5, 1.0, 0.5);           // non-uniform scaling
modelMatrix.rotate(-myAngle,0.0,0.0,1.0);   // rotate (animated)
drawAxes();                                 // draw it!
```

then:

9) **(4 pts)** The central 'dot' for the arrows will be drawn:
A) in the center of the upper right quarter (or 'quadrant') of the HTML-5 canvas;
B) in the center of the upper left quarter (or 'quadrant') of the HTML-5 canvas;
C) ... right quarter (or 'quadrant') of the HTML-5 canvas;

9--D –translate drawing axes
down,left (-0.5,-0.5…)

D) ... (or 'quadrant') of the HTML-5 canvas;
E) ... the same 'dot' location shown in the drawing above)
F) ... of the above

10) **(4 pts)** As the 'myAngle' value varies smoothly between 0 and 90, where will we find the center of rotation (e.g. the 'hinge point') in the on-screen drawing?
   A) in the center of the upper right quarter (or 'quadrant') of the HTML-5 canvas;
   B) in t… …ter (or 'quadrant') of the HTML-5 canvas;
   … …uadrant') of the HTML-5 canvas;
   … …rant') of the HTML-5 canvas;
   … …ame 'dot' location shown in the drawing above)
   F) Some… …one of the above

10--D –translate drawing axes down, left (-0.5,-0.5…), then rotate there

11) **(4 pts)** The smoothly-changing 'myAngle' variable animates the drawing. As its value changes,
   A) all parts of both arrows stay entirely on-screen;
   B) … …shed arrow is always fully visible;
   … …rrow is always fully visible;
   … …me time;
   E) Something else happens; none of the above

11--A –careful! The axes are stretched, but we rotate to MINUS 90 degrees, not +90!

12) **(4 pts)** When '`myAngle`' == 0, does the solid arrow have the same length as the dashed arrow on-screen?
   A) Ye… …exactly the same length as the dashed arrow
   B) No… …a longer length than the dashed arrow
   C) No; the soli… …creen with a shorter length than the dashed arrow
   D) Something else happens; none of the above.

12--C –note the non-uniform scaling.

13) **(4 pts)** When '`myAngle`' varies smoothly from 0 to 90, does solid-arrow length change on-screen?
   A) No; the solid arrow…
   B) Yes; … …0.
   C) Yes; s… …=90.
   D) Something else happens, no…

13--C –careful! Non-uniform scaling! at 90 degrees, solid arrow aims in the stretched +/- y direction.

14) **(4 pts)** When '`myAngle`' varies smoothly from 0 to 90, does dashed-arrow length change on-screen?
   A) No; the dashed ar…
   B) Yes; d… …`le`=0.
   C) Yes; das… …`gle`=90.
   D) Something else happens; …

14--B –careful! Non-uniform scaling! at 0 degrees, dashed arrow aims in the stretched +/- y direction.

Draw your own 'scene-graph' for our set of 4 statements (listed above question 9).

15) **(4 pts)** The graph will contain a node for each of the 4 statements, and
   A) The transform node for **`scale()`** is a descendant of the transform node for **`rotate()`**
   B) The transform node for **`rotate()`** is a descendant of the transform node for **`scale()`**
   C) The **`rotate()`** and **`scale()`** nodes are siblings – both are child nodes of the same group node
   D) The **`rotate()`** and **`scale()`** nodes are unrelated—they do not share the same parent node
   E) Something else happens; none of the above.

16) **(4 pts)** Our sequence of 4 statements illustrates how to traverse a scene-graph to draw the animated, jointed objects it describes. In general, we traverse a scene-graph (not just ours) to generate statements:
   A) In breadth-first order, starting from the top (root) of the graph, ending at the leaf nodes
   B) In breadth-first order, starting with the leaf nodes, and always ending at the top (root)
   C) In depth-first order, starting from the top (root) of the graph, ending at the leaf nodes

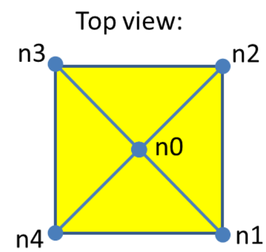
D) In depth-first order, starting from each leaf node in turn, and always ending at the top (root).
E) Something else happens; none of the above.

17) **(5 pts)** If we use [T] to represent the 4x4 matrix that performs our translation, and
[R] to represent the 4x4 matrix that performs our rotation, and
[S] to represent the 4x4 matrix for scale,
The modelMatrix variable contains a 4x4 matrix. Just before we call drawAxes(), the modelMatrix values are the result of combining several transformation matrices. How were they combined?
A) By matrix multiply: [modelMatrix] = [T][R][S]
(HINT: if you multiply a Vector4 by modelMatrix, we could get the same result if we multiplied the Vector4 first by S matrix, then R matrix, then T).
B) By matrix multiply; [modelMatrix] = [T][S][R]
C) By matrix multiply: [modelMatrix] = [R][T][S]
D) By matrix multiply; [modelMatrix] = [R][S][T]
E) By matrix multiply; [modelMatrix] = [S][T][R]
F) By matrix multiply; [modelMatrix] = [S][R][T]
G) Something else happens; none of the above.

Vertex Sequencing

18) **(4 pts)**If we use WebGL's gl.TRIANGLES drawing primitive (not TRIANGLE_STRIP), how many vertices do we need to draw a 3D cube?
A) 8: one vertex per corner; however, this limits each corner to 1 color only (where 3 faces meet)
B) 20: 8 corners + 6 faces*2 more vertices to split the face into 2 triangles
C) 24: 8 corners * 3 vertices/corner (one vert for each face)
D) 36: 6 faces * 2 triangles/face * 3 vertices/triangle.
E) Something else happens; none of the above.

19) **(6 pts)** There are many ways to tesselate 4-sided pyramid with these nodes (square pyramid base is facing away from you) into one single triangle strip (gl.TRIANGLE_STRIP, not gl.TRIANGLES). A 'good' tessellation makes a strip that covers all surfaces with no redundancies, no degenerate triangles, and always-correct winding order, such as this sequence of vertex-locations:



A) n2, n0, n3, n4, n2, n1, n0, n4
B) n0, n2, n0, n3, n4, n2, n1, n0, n4
C) n3, n0, n2, n1, n3, n4, n0, n1 **(The fold-up model shown in class Fri Feb 2, 2018)**
D) n2, n3, n0, n4, n1, n2, n0, n2, n2, n4, n3
E) Something else. None of these sequences tesselate the shape correctly.