CS351 Introduction to Computer Graphics

Test A: Shape
100 pts max.
Jan 31, 2016

NetID
(netID is 6 let

* * * SOLUTION * * *

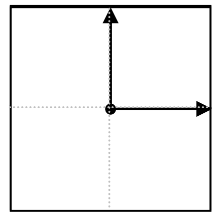
INSTRUCT ... **ter your HIGHLIGHTED answers. Upload your own file on Canvas before the end of the day Sunday, Jan 31, 2016, 11:59PM.**

1) Suppose that: ... ut, ... ng

RECALL: In OpenGL/WebGL (and nearly all other modern graphics systems)
we name transform functions by their effect on drawing axes, not on vertices:
--make a new copy of the current 'drawing axes'
(or 'reference frame' or 'coordinate system'; the origin point and coord axes), then
--transform the new drawing axes (as measured in 'current' drawing axes),
--don't change vertex coordinates; just draw them in the new drawing axes.

The on-screen result of this sequence of statements is:

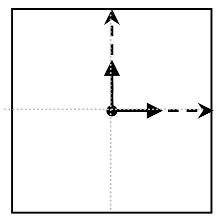
```
modelMatrix.setIdentity(); // set to identity matrix.
drawAxes();                // draw it!
```



HINT: All arrows stay entirely within the canvas

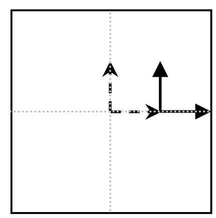
1a) **(5pts)** Sketch the on-screen result if you used these statements instead:

```
modelMatrix.setIdentity();
modelMatrix.scale(0.5, 0.5, 0.5);       // shrink to 50%
drawAxes();                             // draw it!  ANS: →→→
```



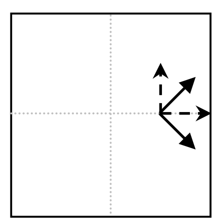
1b) **(5pts)** Sketch the on-screen result if you used these statements instead:

```
modelMatrix.setIdentity();                         ANS:
modelMatrix.scale(0.5, 0.5, 0.5);       // shrink to 50%
modelMatrix.translate(1.0, 0.0, 0.0);   // move +x by 1
drawAxes();                             // draw it!  ANS: →→→
```



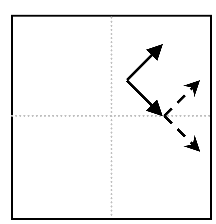
1c) **(5pts)** Sketch the on-screen result if you used these statements instead:

```
modelMatrix.setIdentity();
modelMatrix.translate(0.5, 0.0, 0.0);   // move +x by .5
modelMatrix.scale(0.5, 0.5, 0.5);       // shrink to 50%
modelMatrix.rotate(-45.0,0.0,0.0,1.0);// z-axis rotate
drawAxes();                             // draw it!  ANS: →→→
```



1d) **(5pts)** Sketch the on-screen result if you used these statements instead:

```
modelMatrix.setIdentity();                         ANS:
modelMatrix.translate(0.5, 0.0, 0.0);   // move +x by .5
modelMatrix.scale(0.5, 0.5, 0.5);       // shrink to 50%
modelMatrix.rotate(-45.0,0.0,0.0,1.0);// z-axis rotate
modelMatrix.translate(-1.0, 0.0, 0.0);// move +x by 1
drawAxes();                             // draw it!  ANS: →→→
```



2) (16pts) TRUE/FALSE: (copy-and-paste your choice of these highlighted answers "True" or "False")

a) True____ Drawing commands for WebGL and drawing commands for HTML5 'canvas' elements use different on-screen drawing axes; one has its origin at the upper left corner.

b) False___WebGL ***requires*** users to specify all vertex positions using real values (floats). This requirement ensures that limited precision never introduces rendering flaws on-screen.

c) False___WebGL provides its own built-in functions that can create a 4x4 matrix for translation, for rotation, or for scale, each from a single function call, as well as a push-down stack for them.

d) False___All the many parameters kept as state variables by WebGL, such as background color, buffer bindings, depth testing, etc., are 'write-only': you can't read any of their current values.

e) True____Every WebGL/HTML5/JavaScript program that can draw WebGL drawing primitives on-screen (e.g. TRIANGLES) must include both a 'Vertex Shader' and a 'Fragment Shader'.

f) True______ With 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 attributes are NOT interleaved!

g) True_____ WebGL permits you to use the same 'uniform' variable to send values to both the Vertex Shader and Fragment Shader. If JavaScript sets its value, then both shaders 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're more conv

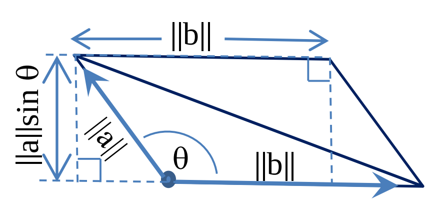
HIGHLIGHT your choice to mark your answer.

3) (4pt) What are the dimensions of the 'Canonical View Volume' (CVV) in openGL?

a) adjustable—specified indirectly HTML commands for 'canvas' object where WebGL
b) adjustable—call the gl.setCVV() command. Default is unit cube: (+/-1, +/-1, +/-1))
c) Fixed—a unit cube centered at the origin: (+/-1, +/-1, +/-1)
d) Fixed—a unit cube, with x,y,z origin shown at the lower-left corner of the display wi
e) Fixed—a unit cube, with x,y,z origin shown at the upper-left corner of the display wi

4)(3pt) Calculate the area of a triangle whose sequence of vertices lie at these three (x,y,z) posi
(1,2,3), (-3,3,-4), (3,-4, 5). Briefly, show your work in the space bel
(HOW? see assigned readings: Chap 2.3 Lengyel " Math for 3D…" posted on Canvas)

A == (3,-4,5) – (1,2,3) = (2,-6, 2)
B== (-3,3,-4) – (1,2,3) = (-4, 1, -7)
||**AxB**|| = ||**A**|| ||**B**|| cos θ = 2*AREA= ||-40, -6, 22|| = 46.04;
AREA = 23.02 Handy cross-product calculator:
http://onlinemschool.com/math/assistance/vector/multiply1/

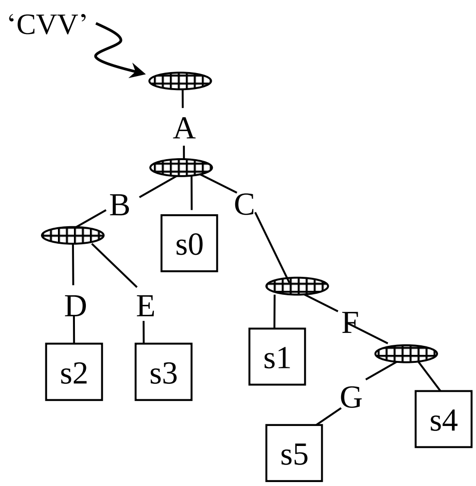


Or choose another of the 3 edge pairs:
C == (3,-4,5) – (-3,3,-4) = (6,-7, 9)
D== (1,2,3) – (-3,3,-4) = (4, -1, 7)
||**CxD**|| = ||**C**|| ||**D**|| cos θ = 2*AREA
= ||40,- 6, 22|| = 46.04. AREA = 23.02

5)(3pt) Calculate the surface normal vector for that same triangle: be sure to 'normalize' the vector to ensure its length is 1.0. (Again, see Lengyel "Math for 3D…" reading posted on Canvas).

CCW vertex ordering (v0,v1,v2) means normal points in (v2-v1) x (v0-v1) direction:
v2-v1 = **C** = (3,-4,5) – (-3,3,-4) = (6,-7, 9); **v0-v1**= D = (1,2,3) – (-3,3,-4) = (4, -1, 7);
CxD = (-40, -6, 22). Normalize: divide each element by ||**CxD**|| = sqrt(40^2+6^2+22^2) = 46.04
unit normal vector n = (-0.8687, -0.1303, 0.4778)

Scene Graph: This 'scene graph' describes a jointed 3D object:
--Each **letter** is a transformation node (*e.g.* holds a 4x4 matrix that combines rotation, translation, scale, etc.)
--Each **gridded ellipse** is a 'group' node, where we have uniquely defined 'drawing axes' (a coordinate system)
--Each **square** holds vertices (fixed, in a VBO) & shape-drawing fcns (*e.g.* 'drawAxes()', 'drawCube()', etc.)



The root of the tree shown begins in the 'CVV' coordinate system, and each leaf of the tree ends in its own separate coordinate system where we draw a shape (s0,s1,s2, etc.), defined by fixed sets of vertices.

RECALL THAT:
All our matrix transformation commands presume:
a) Vertices are 4-element column vectors;
b) When the current transform matrix [M] multiplies a given vertex v to make v', we get v' = [M]v, and
c) Any transform command (*e.g.* **`rotate(), translate(), scale()`**) multiplies the current matrix M with a new matrix that 'precedes' it's effect on vertices. Thus a call to **`rotate()`** will replace current matrix [M] with the result of this matrix multiply: [M][R], which we write as [MR]. If applied to the *coordinate values* for a vertex, the result is the same as multiplying by [R], and then by [M].
BUT we could also interpret these same calculations as: "Starting from the CVV, we first transform the drawing axes by [M], then transform these new drawing axes by [R]."

6) (8pts) Suppose we write software that traverses the tree and draws the entire scene, using pushMatrix() and popMatrix() as necessary. When we issue the drawing commands contained in scene-graph node s5, what are the contents of our current matrix?

a) [A]
b) [G]
c) [ACFG]
d) [GFCA]
e) [AGCF]
f) [GAFC] (HIGHLIGHT YOUR ONE ANSWER)
g) Something else:____________

7) (6pts) List *all* transform nodes that can modify the on-screen result for each shape. For example, shape s2 *might* change on-screen if we modified transform D, but G can't change s2.
Write your answers as comma-separated lists of letters, such as (A,B,C,D,E).

s0:_________ s3:_________

s1:_________ s4:_________

s2:_________ s5:_________

s0: 1 (A)	s3: 3 (A,B, E)
s1: 2 (A, C)	s4: 3 (A,C,F)
s2: 3 (A,B, D)	s5: 4 (A,C,F,G)

VERTEX / VECTOR MATH:

Use these x,y,z coordinates for the 3D points **P0**, **P1**, the origin, and the 3D vectors **V0**,**V1**:

NAME:	x, y, z, w	NAME:	x, y, z, w
P0:	1, 2, 3 ?	**V0**:	3, 2, 1 ?
P1:	-1, 1, 0 ?	**V1**:	0, 4, 3 ?
Orig:	0, 0, 0 ?		

for points, w == 1
for vectors, w == 0
Point-Point = Vector:
thus w==0
Point-Vector = Point:
thus w==1

8)(4pts) What are the correct 'w' values for **P0** and **P1**? **w ==**____

What are the correct 'w' values for **V0** and **V1**? **w ==**____

What is the correct 'w' value for (**P0 – P1**)? **w ==**____

What is the correct 'w' value for (**P0 – V0**)? **w ==**____

9) Find your answer using 3-D homogeneous coordinates (e.g. a 4-tuple; a column of 4 numbers):

a) **(3pts)** Find a new vector that points from P1 to P0: (P1→P0)

(, , ,) (write

P0 – P1
x=(1) – (-1) = 2
y=(2) – (1) = 1
z=(3) – (0) = 3
w=(1) – (1) = 0

b) **(3pts)** Find the length of vector V1 given above:

(.) (write
not a

|V1| = sqrt($x1^2$ + $y1^2$ + $z1^2$ + $w1^2$)
= sqrt(0^2 + 4^2 + 3^2 + 0^2)
= sqrt(16 + 9)
= 5.0

c) **(3pts)** Find the point halfway between points P0 and P1:

(, , ,) (write 4 real numbers, not an expression)

Geometrically, we scale vectors, not points: P0 + (P1-P0)*0.5.
But this simplifies to: P0*0.5 + P1*0.5 =
x=(1)*0.5 + (-1)*0.5 = 0.0
y=(2)*0.5 + (1)*0.5 = 1.5
z=(3)*0.5 + (0)*0.5 = 1.5
w=(1)*0.5 + (1)*0.5 = 1

d) **(3pts)** Find the

(.)

V0 . V1: x: (3)*(0) +
y: (2)*(4) +
z: (1)*(3) +
w: (0)*(0) = 11.0

e) **(4pts)** Find vector perpendicular to both v0 and v1, with z <0

(, ,

Hmm. a cross product, but which one? (V0 x V1)?

```
| i  j  k |      i  j  k |i  j  k |i  j  k
|x0 y0 z0 | ::   3  2  1 |3  2  1 |3  2  1 ::
|x1 y1 z1 |      0  4  3 |0  4  3 |0  4  3
```

= (2*3 – 1*4)**i** + (1*0 – 3*3)**j** + (3*4 - 0*2)**k**
= (2)**i** + (-9)**j** + (12)**k**

Whoops! V0 x V1 points the wrong way! We need to find V1 x V0,
but that's easy: remember V0 x V1 = -(V1 x V0). Change the signs.

x= -2
y= +9
z= -12
w= 0

10) **(20pts) True/False** (copy-and-paste your choice of these highlighted answers "True" or "False")

a)__False____Support for programmable shaders written in GLSL has always been available on almost all implementations OpenGL since 1992, and part of the standard from version 1.0 onwards.
GLSL implementation first appeared in version 2.0, and was not mandatory until 3.1.

b)__False___Each GLSL vertex shader program receives ***sets*** of vertices as inputs. For example, when a WebGL program draws a TRIANGLES primitive, it calls the vertex shader once, and supplies three vertices as input.
GLSL vertex shader programs are parallel, SIMD, and each operates on attributes of only one vertex.

c)__False__GLSL vertex program performs only geometric calculations, and never any color calculations, because it has no way to affect the on-screen colors of any pixels: only the fragment shader can set pixel colors.
As shown in starter code, vertex shader can directly set gl_FrontColor (to red); it may compute new colors as well, or pass a 'varying' value to the fragment shader, which can modify, use, or ignore them.

d)__True___GLSL fragment shader programs cannot change the on-screen position of a vertex; only the vertex shaders can modify vertex positions.
(See GLSL Lang. Spec.)

e)__True____In a GLSL vertex shader, global variables preceded by the 'attribute' qualifier may be different for each vertex.

f)__True____The GLSL qualifier 'uniform' for a variable means the GLSL value cannot change while processing a single drawing primitive.

g)___True____The GLSL qualifier 'varying' used within a vertex shader means that variable's values get rasterized. The graphics hardware computes interpolated values for each pixel within the drawing primitive and each fragment shader receives a separately-computed value, which may be different for each pixel.

h)___True____The GLSL 'swizzling' capability lets you re-arrange and/or exclude vector elements; it can convert a color vector expressed in r,g,b,a order to a new color vector in b,g,r,a order.

j)___False____As GLSL offers built-in vector and matrix types, it does NOT permit fixed-sized arrays (e.g. you cannot declare: float myArray[7]; inside a GLSL shader).

k)__False______The GLSL language permits use of the 'varying' qualifier for both local variables and global variables in both the vertex shader and the fragment shader. Your JavaScript code can set their values, too.

NO! NO! NO! You can't make local 'varying' vars: they MUST be global!
The values of 'Varying' vars computed in the Vertex Shader get sent to rasterizing hardware.
Each instance of the Fragment Shader program receives 'varying' vars as inputs, with values interpolated between vertices to compute new values for each pixel; we can't set them in JavaScript!