CS559 Computer Graphics – Fall 2015

Practice Midterm Exam

Time: 2 hrs

1. [$XX \times YY\% = ZZ\%$] MULTIPLE CHOICE SECTION. Circle or underline the correct answer (or answers). You do not need to provide a justification for your answer(s).

(1) When combining transforms by multiplying the respective 4×4 matrices, the *order* of multiplication (i.e. the order of composition) typically matters. In which of the following cases can we, as an exception, swap the order between two sequentially applied transformations?
(Circle or underline ALL correct answers)

(a) A translational and a scaling transformation.
[✓ (b) Two scaling transformations.]
(c) A translation and a perspective projection transformation.
(d) Two rotation transformations in 3D.

(2) Which of the following statements about homogeneous coordinates are accurate?
(Circle or underline ALL correct answers)

[✓ (a) Using homogeneous coordinates allows us to express more than just linear transformations as matrix multiplications.]
(b) With homogeneous coordiates the graphics pipeline can implement perspective transformations without requiring expensive division operations.
[✓ (c) When combined with 4×4 transformation matrices, they allow us to compose affine transforms by simple matrix multiplication.]

(3) Will a perspective transformation preserve the *angle* between two intersecting lines in space (i.e. will the projected lines on the display have the same angle?)
(Circle or underline the ONE most correct answer)

(a) Yes. Other properties like relative distances can change, but angles are preserved under perspective projection.
(b) The only thing that is preserved is that lines that are parallel in 3D space remain parallel (on the display) after perspective transformation. Angles, however, are not preserved.
[✓ (c) No; in the general case angles between lines, and even the property of lines being parallel can change after a perspective transformation.]

(4) Which of the following would be a good test for checking that points $\mathbf{p}, \mathbf{q}$, and $\mathbf{r}$ (in the 3-dimensional space) are *co-linear*?
(Circle or underline the ONE most correct answer)

(a) A dot product test : $(\mathbf{p} - \mathbf{q}) \cdot (\mathbf{q} - \mathbf{r}) = 0$

[✓ (b) A cross product test : $(\mathbf{p} - \mathbf{q}) \times (\mathbf{q} - \mathbf{r}) = \vec{0}$]

(c) A cross product test : $(\mathbf{p} - \mathbf{q}) \times (\mathbf{p} - \mathbf{r}) = \|(\mathbf{p} - \mathbf{q})\|\|(\mathbf{p} - \mathbf{r})\|$ (where $\|\mathbf{x}\|$ is the magnitude of the vector $\mathbf{x}$)

(5) Given that perspective projection is closer to how our human visual system works (as well as cameras, etc), why would we ever consider using orthographic projection?
(Circle or underline the ONE most correct answer)

(a) The cost of rendering images using orthographic projection is substantially lower.

(b) In the absence of perspective projection, all transforms can simply be described as 4 × 4 matrices.

[✓ (c) Orthographic projection keeps lines that are parallel in the world, parallel on the screen. This can be useful for, say, engineering and architectural applications.]

(6) If the normals given for the three vertices of a triangle are exactly equal, would we see any difference if we did per-vertex lighting vs. per-fragment lighting in a Phong model?
(Circle or underline the ONE most correct answer)

(a) No. In this case all the interpolated normals inside the triangle would be equal to the constant normal of every vertex. Thus, lighting computations would produce exactly the same result.

(b) Yes. Diffuse reflection would still produce different results in this case if done on a per-fragment basis (assume just a directional light).

[✓ (c) Yes. Specular reflection could produce different results if done on a per-fragment basis.]

(7) Which of the following statements is true, about the matrix that can be used to transform *normal* vectors?
(Circle or underline ALL correct answers)

(a) It is equal to the top 3 × 3 block of the 4 × 4 matrix used to transform positions.

(b) It is always a rotational matrix.

[✓ (c) If positions are transformed as $\mathbf{T}(x) = \mathbf{M}x + b$, the inverse-transpose matrix $(\mathbf{M}^{-1})^T$ is the one that will transform the normals.]

2. [$XX \times YY\% = ZZ\%$] SHORT ANSWER SECTION. Answer each of the following questions in no more than 1-3 sentences.

(a) Write a mathematical expression for a vector that is *perpendicular* (or *normal*) to the plane in 3D that passes through the three non-collinear points $\mathbf{p}, \mathbf{q}$, and $\mathbf{r}$.

$$(p-q) \times (q-r)$$

(This is just one possible correct expression)

(b) Name 2 reasons why hierarchical modeling is particularly convenient when creating models of articulated objects made from rigid parts (e.g. a robot that can move its limbs).

Note: More correct answers exist...

→ Each part can be independently modeled in a convenient coordinate system

→ Easy manipulation with few parameters (angles)

→ Easy to ensure parts are properly connected (no gaps...)

(c) The three vertices of a triangle $P_1P_2P_3$ have the texture coordinates $(u1, v1) = (0.2, 0.2)$, $(u2, v2) = (0.2, 0.4)$, and $(u3, v3) = (0.6, 0.2)$ associated with them. Let P be an interior point of the triangle, whose location is given by the following expression:

$$P = 0.25 \cdot P_1 + 0.25 \cdot P_2 + 0.5 \cdot P_3$$

What would be the texture coordinates associated with P?

$$\begin{aligned} & 0.25 \cdot (0.2, 0.2) \\ + \; & 0.25 \cdot (0.2, 0.4) \\ + \; & 0.5 \cdot (0.6, 0.2) \end{aligned} = \boxed{(0.4, 0.25)}$$

(d) The OpenGL Shading Language supports `uniform` variables, vertex `attributes` and `varying` variables. Give two (2) examples of vertex attributes (at least one vector-valued), one example of a uniform variable (of any type) and what purpose it serves, and one example of a varying variable with an explanation of why we would need it.

→ Attributes : Vertexposition, color, normal
tex.coordinate

→ Uniform : (mat4) Model View Proj transform.

→ Varying : Per-fragment color/normal/tex.coordinate

(e) In local illumination models (e.g. the Phong model we discussed in class) shading of a given point on a visible surface object is computed independently of how any other surface has been shaded. Describe two examples of scenes or phenomena that a local illumination model would not be able to realistically reproduce.

→ Non-transparent media (fog, smoke, etc)

→ Refraction

→ Color bleeding (appearance of an object depending on color of another).

(f) When rasterizing a triangle (in modern graphics hardware) barycentric coordinates are computed for an entire rectangular region of pixels surrounding the triangle in question – the barycentric coordinates are then checked to indentify if the triangle overlaps with any given pixel. Inevitably, computation will be spent for a number of pixels that will ultimately be determined to be outside the triangle. Why would this approach be followed (instead of scanline rasterization) even when there is the possibility of wasted computation?

Computing barycentric coordinates is a very regular operation (does not need branching/conditionals) and is very parallel (every fragment can be done independently). The performance gains due to these properties on modern, massively parallel graphics hardware outweigh any "wasted" computation

3. [XX%] A camera is positioned as follows (relative to the 3D world coordinate system):

- The center of projection (the "eye" position) is at : $\mathbf{e} = (3, 2, 2)$.
- The "lookAt" location is at: $\mathbf{l} = (2, 1, 1)$.
- The "up" vector is: $\mathbf{t} = (0, 1, 0)$.

Compute the three basis vectors $(\mathbf{u}, \mathbf{v}, \mathbf{w})$ of the *camera* coordinate system.

gaze vector $\vec{g} = \vec{l} - \vec{e} = (-1,-1,-1)$

$$\vec{g} \times \vec{t} = (1,0,-1)$$

$$\vec{u} = \frac{\vec{g} \times \vec{t}}{\|\vec{g} \times \vec{t}\|} = \boxed{\frac{1}{\sqrt{2}}(1,0,-1)}$$

$$w = -\frac{\vec{g}}{\|g\|} = \boxed{\frac{1}{\sqrt{3}}(1,1,1)}$$

$$\vec{v} = \vec{w} \times \vec{u} = \boxed{\frac{1}{\sqrt{6}}(-1,2,-1)}$$

4. [XX%] There are several errors in the following vertex/fragment shader pair, that would prevent it from compiling. Identify as many as you can.

Vertex shader:

```
precision highp float;
attribute vec3 position;
attribute vec3 normal;
attribute vec3 color;
uniform mat3 normalMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
varying vec3 fNormal;
varying vec3 fColor;

void main(){
    fNormal = normalize(normalMatrix * normal);
    vec4 pos = modelViewMatrix * position;
    fColor = color;
    gl_Position = (projectionMatrix * pos)~~.xyz~~;
}
```

Should be vec4(position, 1.0)

↳ gl_Position is $\boxed{\text{vec4}}$

Fragment shader:

```
precision highp float;
varying vec3 fNormal;

const vec3 ambientColor = vec3(1.0,0.5,0.5);
const vec3 lightDir = vec3(1.0,0.0,0.0);
const float ambientFactor = 0.3;
const float diffuseFactor = 1;

void main(){
    vec3 s = normalize(lightDir);
    vec3 n = normalize(fNormal);
    vec3 ambient = ambientColor*ambientFactor;
    vec3 diffuse = fColor*diffuseFactor*max(0.0,dot(n,s));
    gl_FragColor = ambient+diffuse;
}
```

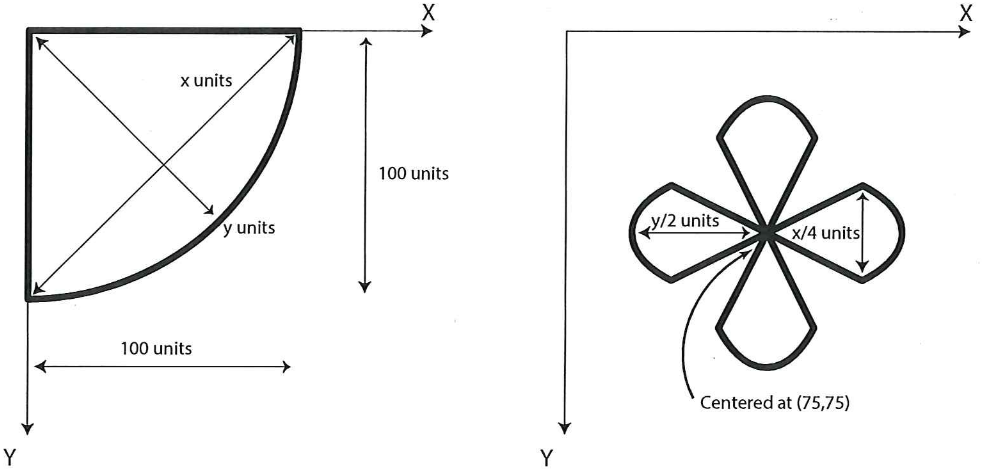
Missing
varying vec3 fColor

type mismatch ; should be 1.0

gl_FragColor expects vec4 value

Maybe vec4(ambient+diffuse, 1.0);

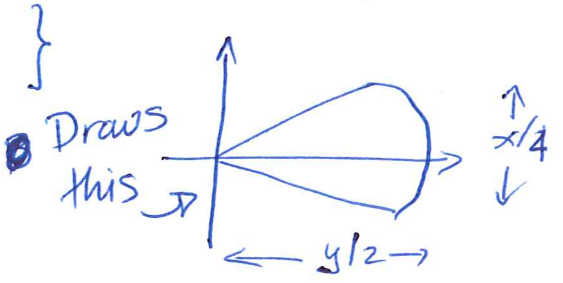
5. [XX%] Imagine that we have implemented a routine `draw()` that draws the shape outlined with a thick black line on the left of the following figure (i.e., the outline of one quadrant of a circle).



We want to use this basic shape to build the fan-shaped design on the right, consisting of four such quadrants that have been scaled, rotated and translated as shown. Describe the sequence of transforms that would be needed to form each of the 4 blades of this fan. You can either (a) Write down the individual transforms, as products of fundamental transforms (you can use notation similar to the HTML5 Canvas transform methods such as "`scale(0.5,2.0)`","`rotate(PI/4)`" and so on, to describe individual fundamental transforms), or (b) by writing pseudo-code that will use the `draw()` routine along with Canvas `save()/restore()` calls and the fundamental transforms above, to draw the desired shape. Be mindful of the *order* of transformations!!

[Note: The exact value of the "x" and "y" dimensions in the illustration above is not so important – the main point is that the "x" has been scaled by a factor of 1/4, and "y" has been scaled to 1/2 of the original length]

```
function drawBlade {
    save();
    scale(0.5,0.25)
    rotate(-PI/4);
    draw();
    restore();
}
```

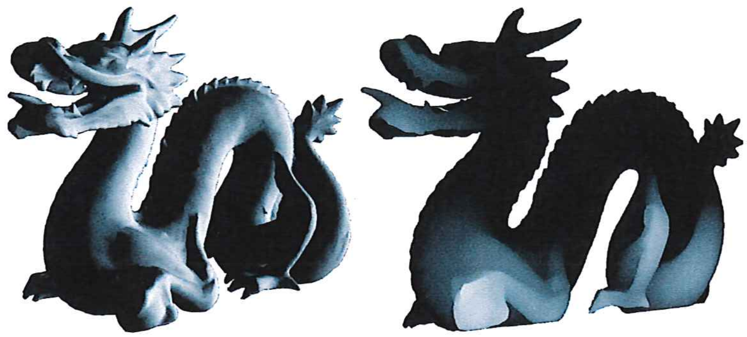


```
function drawFan() {
    translate(75,75);
    drawBlade();
    save(); rotate(PI/2);
      drawBlade(); restore();
    save(); rotate(PI);
      drawBlade(); restore();
    save(); rotate(-PI/2)
      drawBlade(); restore();
}
```

6. [XX%] Write the code of a simple vertex/fragment shader pair to produce the following appearance effect:

   - Objects to be drawn should get flattened along the z-axis, i.e. a vertex with coordinates (x, y, z) should be flattened into $(x, y, 0)$. This should cause 3-D objects to appear as if they have been compressed down to the plane $z = 0$. [Hint: Do this in the vertex shader].
   - The rendered color of the object should be a greyscale value, proportional to the z-value of the material point (that is, the z value that it had before it was collapsed down to the plane z = 0). [Hint: Do this in the fragment shader, with some help from the vertex shader].

   The illustration to the right shows an example of the effect we seek to reproduce (the left image is a standard shader applied to the same model).



Feel free to assume that the vPosition vertex attribute is available and provides the vertex position in local coordinates, along with (uniform) matrix variables modelViewMatrix, projectionMatrix etc. Alternatively, feel free to define your own variables for any information you need to pass down to the shaders.

VS:

```
attribute vec3 vPosition;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
varying float z;
void main () {
   z = vPosition.z;
   vec4 camera_pos =
     modelViewMatrix *
      vec4(vPosition.xy, 0.001*vPosition.z, 1.0);
  gl_Position = projectionMatrix * camera_pos;
}
```

To avoid z-fighting, flatten almost to a plane

FS:

```
varying float z;
void main () {
 gl_FragColor
      = vec4 (z, z, z, 1.0);
}
```