CS559 Computer Graphics – Fall 2015

Midterm Exam – Thursday October 27th 2015

Time: 2 hrs

Name	
University ID	

Part #1	
Part #2	
Part #3	
Part #4	
Part #5	
Part #6	
TOTAL	

1. [$5 \times 4\% = 20\%$] MULTIPLE CHOICE SECTION. Circle or underline the correct answer (or answers). You do not need to provide a justification for your answer(s).

   (1) The construction of the camera transform (e.g. via the TGWL `lookAt` method) requires us to provide an "up" vector as one of the inputs. Which of the following statements about this vector are true?

   Write the letters of ALL correct answers here: b

   (a) The "up" vector will be used as one of the basis vectors of the camera coordinate system.

   (b) The "up" vector will be perpendicular to at least one of the basis vectors of the camera coordinate system.

   (c) The construction of the camera transform requires that the "up" vector must be perpendicular to the gaze direction (i.e. the vector from the *eye* location to the *lookAt* or "target" location).

   (2) Any point P in the plane of a triangle ABC can be written as a linear combination

   $$P = \alpha A + \beta B + \gamma C$$

   where α, β and γ are the *barycentric coordinates* of P (to be clear, the variables P, A, B, C are the position vectors of the respective points in this expression). Let us now consider the special case when the point P actually lies *on the triangle edge* AB. Which of the following statements would be true in that case?

   Write the letters of ALL correct answers here: ab

   (a) It must be the case that $\alpha + \beta = 1$.

   (b) It must be the case that $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$

   (c) It must be the case that $\alpha = \beta = 0$.

   (3) If our shaders require per-fragment normals, it is quite typical for the vertex shader to assign the per-vertex normals (which would have been vertex *attributes*) to a *varying* variable so that the fragment shader can access this information (interpolated from vertices to fragment locations). It is also quite frequent that the fragment shader would start by *normalizing* the value it receives in this *varying* variable, prior to using it. Is this really necessary, and if so, why?

   Write the letter the ONE most correct answer here: b

   (a) The fragment shader needs to normalize the vector it receives, to ensure that it is actually perpendicular to the triangle.

   (b) We normalize to ensure that the length of the normal vector is equal to one, while keeping its direction unchanged.

   (c) We only need to normalize (to unit length) if the original per-vertex normals were not guaranteed to be unit-length.

(4) Which of the following statements about the *painter's algorithm* are correct?

Write the letters of ALL correct answers here: bc

(a) It prevents unnecessary drawing of triangles that will be fully hidden behind other triangles.

(b) It needs to repeat the sorting of triangles if the camera position moves.

(c) It can fail to produce the right result when some of the triangles to be drawn intersect one another.

(5) How can the host program communicate data to the fragment shader?

Write the letter the ONE most correct answer here: c

(a) The host program can specify the values of *varying* variables

(b) It is not possible. The fragment shader can only access data given to it by the vertex shader.

(c) The fragment shader can access uniform variables, which are set by the host program, and also receive information indirectly via the vertex shader.

2. [24%] SHORT ANSWER SECTION. Answer each of the following questions in no more than 1-3 sentences.

   (a) [6%] In many cases it is sufficient to pass a single, combined MVP (Model-View-Projection) matrix to the vertex shader, since the shader can directly apply this transform to vertex positions in local coordinates, to calculate the respective Normalized Device Coordinates (which is what `gl_Position` must be set to). Can you describe a set of circumstances where we would instead *need* to pass down to the shaders more transforms than just the MVP matrix? (for example, a case where we would need to pass separately the ModelView "MV" matrix in addition to the combined MVP transform?)
   The most common case is when the normal vector (passed as a vertex attribute) needs to be transformed within the shaders. The matrix that transforms the normals *could* be computed from the Model-View matrix (although it would have been much more efficient to just pass it in as a separate matrix), but not from the aggregate MVP matrix that includes the projection as well. Also, in certain instances we may want to pass the ModelView (MV) matrix separately to the shaders for transforming point locations to camera coordinates, e.g. for computing specular lighting.

   (b) [6%] Can the following be done in the **V**ertex shader, in the **F**ragment shader, in **E**ither the vertex or fragment shader, or in **N**either of them?
   (Write "V","F","E", or "N" next to each question below. You don't need to provide an explanation, unless you really feel that your answer comes with significant caveats.)

   - N Divide a triangle into smaller triangles
   - E Compute specular lighting
   - N Compute the texture coordinates for the *center* of a triangle (i.e. at the point with barycentric coordinates $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$).

   For the last question: Note that each instance of the vertex program will be given the information pertaining to a specific vertex; it will not have the information to combine several vertices (the three vertices of the triangle) to average their texture coordinates. The fragment shader does get access to interpolated data in the interior of a triangle, but we do not explicitly control which interior point (e.g. the center) will be processed by it.

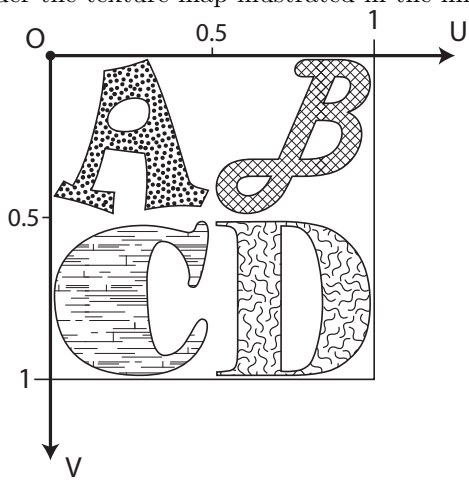
(c) [12%] Here is a list of some tasks that can be done in the vertex shader. Some of them could be done in the fragment shader. For each of the following, say if it would be possible to do this task in the fragment shader. If it is possible, explain how you would decide between which shader to do this in; if it is not possible to use the fragment shader, explain why not.

i. Do texture color lookups.
It is possible, and generally preferable to do texture lookups in the fragment shader, since doing that in the vertex shader would only interpolate the textured color from the vertices of each triangle (thus preventing the opportunity for texture to vary nontrivially within each triangle. That being said, if the size of the triangles is so small (and we have a lot of them) so that we will only render a few pixels per triangle, doing the texture lookup per-vertex might be acceptable.

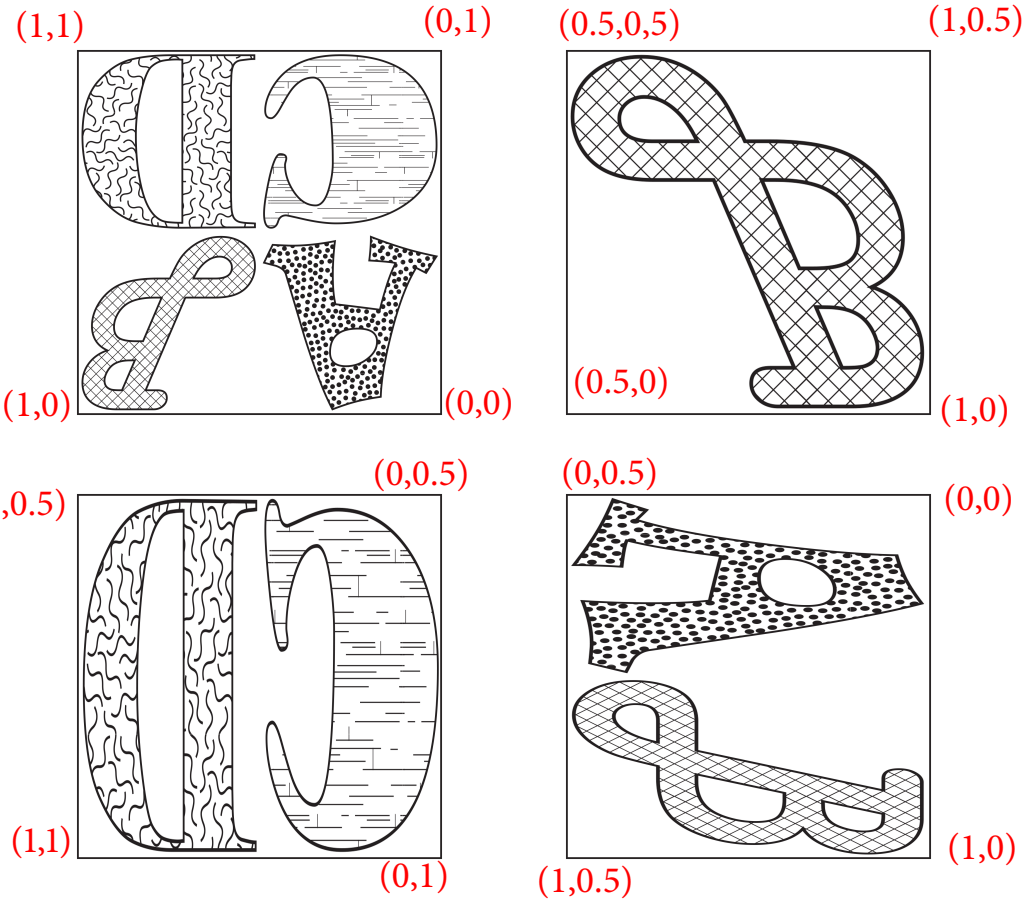
ii. Compute diffuse shading.
Can be done in either shader, the result will be higher quality if done in the fragment shader. However, if the 3 per-vertex normals of every triangle are exactly equal, the result would be the same regardless of whether the diffuse color was computed in vertex or fragment shader (diffuse color depends on normal and light direction).

iii. Change the shape of the object being drawn, for example by stretching or shearing it (independent of any transformations that have been handed down to the shaders by the host program).
Only the vertex shader can actually modify the positions of vertices being drawn (by setting the value of `gl_Position`)

3. [14%] Consider the texture map illustrated in the image below:

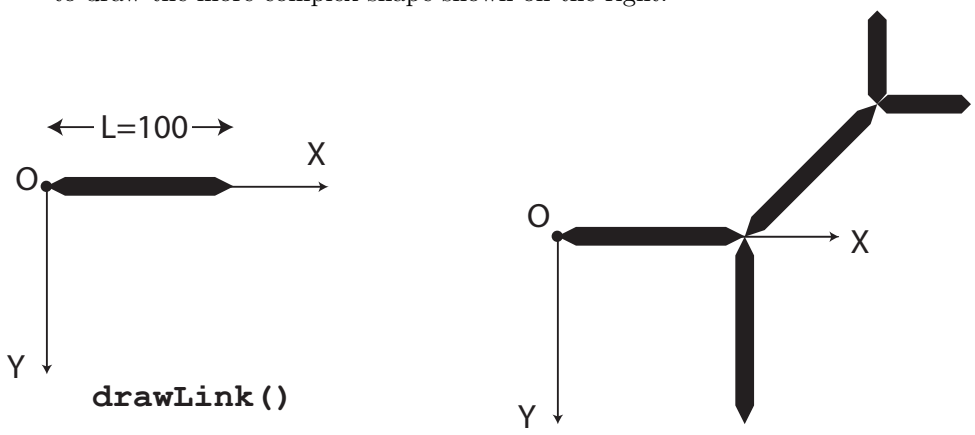


We want to apply this texture to four separate squares, such that the textured squares appear as follows:



For each vertex of each of the four squares, write the texture coordinates that we need to associate with this vertex in order to produce the desired apperance. Write the coordinates in the form (u, v) directly next to each vertex.

4. [14%] We have implemented a function `drawLink()` that draws the pointed stick-shaped object shown on the left. Subsequently we used this function to draw the more complex shape shown on the right:



This was done via hierarchical modeling, as shown in the code below:

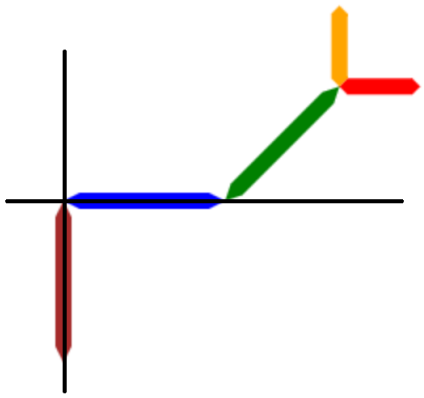
```
context.save();
drawLink();
context.translate(100,0);
context.save();              // Line "A"
context.rotate(-Math.PI/4);
drawLink();
context.translate(100,0);
context.save();
context.rotate(Math.PI/4);
context.scale(0.5,1);
drawLink();
context.restore();
context.save();
context.rotate(-Math.PI/4);
context.scale(0.5,1);
drawLink();
context.restore();
context.restore();           // Line "B"
context.save();              // Line "C"
context.rotate(Math.PI/2);
drawLink();
context.restore();           // Line "D"
context.restore();
```

(a) Draw the shape that the code above would produce, if we removed lines marked "A" and "D".

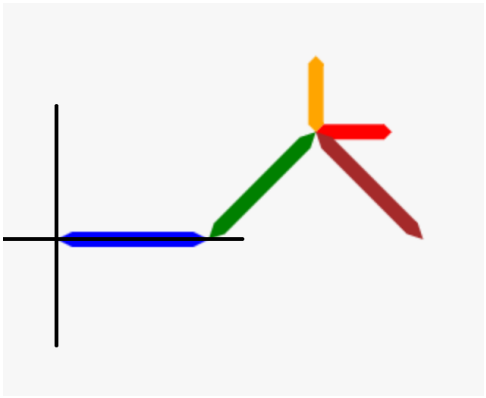
(b) Draw the shape that the code above would produce, if we removed lines marked "B" and "C".

Hints: Since the Y-axis is pointing down, positive angles will be oriented *clockwise*, and negative ones *counter-clockwise*. Also, a friendly reminder that `PI/4`=45 degrees.

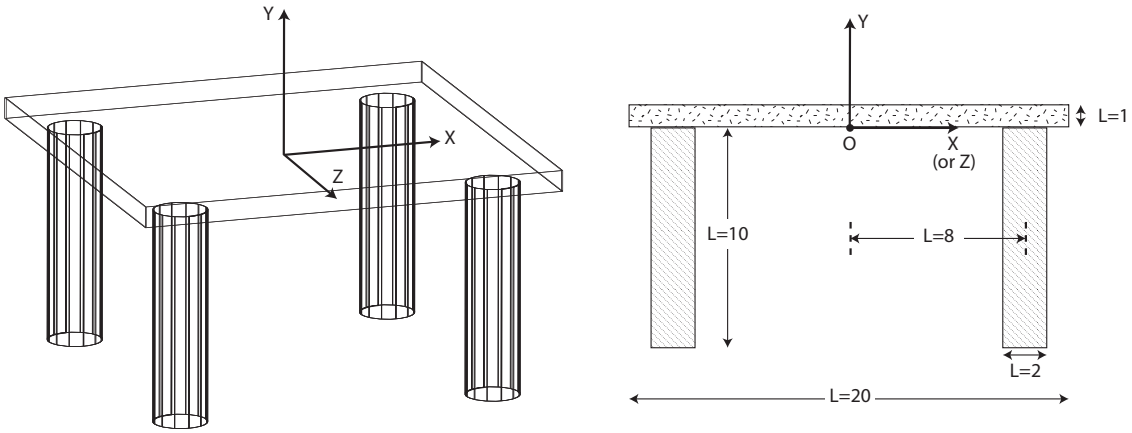
Commenting out lines “A” and “D” yields:



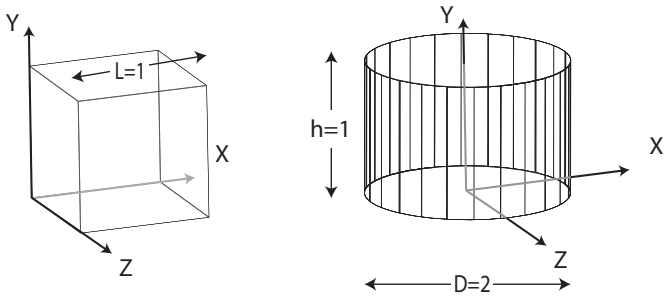
Commenting out lines “B” and “C” yields:



5. [14%] We want to draw a model of the four-legged table shown below (the Z-Y side view will look the same as the X-Y view shown here – just replace the X-axis with the Z-axis in the illustration to the right).



Let's assume that we have been given the two routines `drawCube()` and `drawCylinder()` that draw a unit-edge cube as seen on the left, and a cylinder with radius 1 (diameter 2) and unit height as seen on the right.



Write a program, in pseudo-code, to draw the table using a hierarchical modeling approach. Your code can call the functions `drawCube()` and `drawCylinder()` when needed, and you can also use the Canvas-like `save()/restore()` functions to manipulate the transform stack. You can use functions like `scale(a,b,c)`, `translate(a,b,c)` and `rotateX(angle)` (similarly for `rotateY/rotateZ`) to multiply the current (top-of-stack) transformation with the respective scale/translation/rotation.

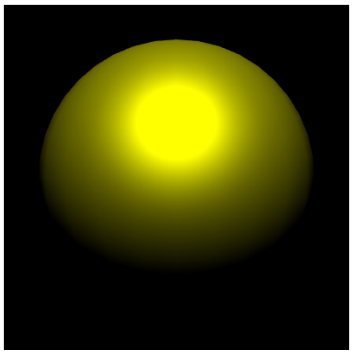
One of many possible correct solutions:

```
function drawTop(){
  save();
  translate(-10,0,-10);
  scale(20,1,20);
  drawCube();
  restore();
}

function drawLeg(){
  save();
  translate(0,-10,0);
  scale(1,10,1);
  drawCylinder();
  restore();
}

function drawTable(){
  drawTop();
  save();translate(-8,0,-8);drawLeg();restore();
  save();translate(8,0,-8);drawLeg();restore();
  save();translate(-8,0,8);drawLeg();restore();
  save();translate(8,0,8);drawLeg();restore();
}
```

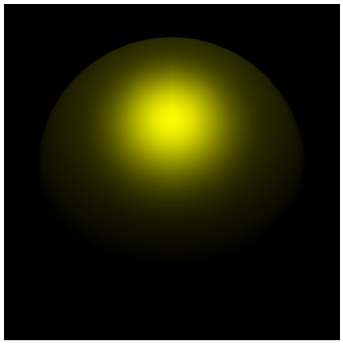
6. [14%] The image below has been produced by a certain set of parameters for ambient, diffuse and specular lighting.

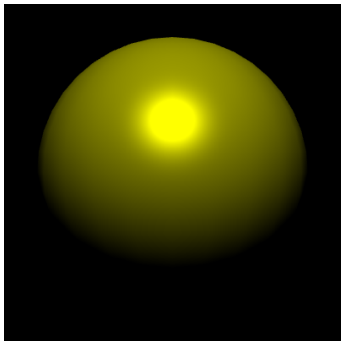


Each one of the following images was created by either increasing or decreasing just *one* of the following parameters, relative to the image above.

- The **A**mbient lighting coefficient.
- The **D**iffuse reflection coefficient.
- The **S**pecular reflection coefficient.
- The Specular **E**xponent.

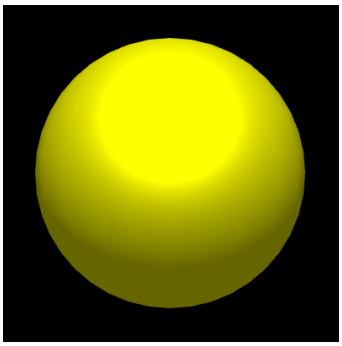
D↓

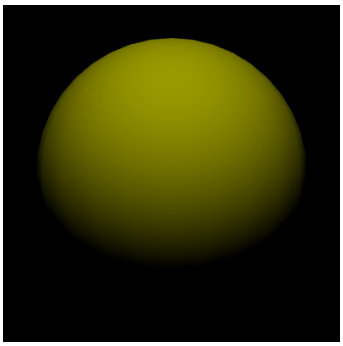




E↑

A↑





S↓

In each of the boxes provided, write "A","D","S" or "E" to indicate the parameter that changed, with an up- or down-arrow to indicate if the parameter was increased or decreased.