CS475/675: Computer Graphics

Mid-Semester Examination (Solutions), Fall 2016

This paper has **4 (four)** pages and **6 (six)** questions, totalling **65** points **plus 15** points of extra credit. Make sure you see them all! You do NOT need to get a perfect score on the first 5 questions to be awarded extra credit for the 6th.
No calculators, notes, books, phones, tablets, laptops etc are allowed.
(Non-red) pens only – no pencils.

1. **(10 points)** Anita decides to replace the CIE XYZ system with her own three imaginary primaries:
   A = 2X – Y,
   B = 2Y – Z, and
   C = 2Z – X

   a. (5 points) Express the X, Y and Z primaries as linear combinations of the A, B and C primaries, or argue why this is impossible.

   Solving the system of linear equations, we obtain:
   X = (4A + 2B + C) / 7
   Y = (4B + 2C + A) / 7
   Z = (4C + 2A + B) / 7

   Rubric: *2 points for getting one correct, 4 for getting two correct, 5 for getting all three correct. All variants (including truncated decimal expansions) of the coefficients are acceptable. No justification or calculations need to be shown. Generally 0 points for saying the conversion is impossible, or 1 if there is an interesting (albeit flawed) effort in this direction.*

   b. (5 points) Is the gamut of the ABC system equal to, smaller than, or larger than that of XYZ? Explain your answer (you may use illustrations to aid your argument).

   It's exactly the same as that of XYZ. If you correctly answered part (a), you know that every linear combination of X, Y and Z with non-negative coefficients is also a linear combination of A, B and C with non-negative coefficients. So the ABC gamut must be at least as large as the XYZ gamut. But the XYZ gamut encompasses all colours visible to the average person! Remember that the gamut is not all possible non-negative combinations, it is the set of all physically realizable/visible *combinations. So the ABC space also encompasses all visible colours, and the two gamuts are equal.*

   Another way to show containment is to draw the chromaticity diagram. Using the formulae for x and y in terms of X, Y and Z, the ABC primaries have (x, y) coordinates (2, -1), (0, 2) and (-1, 2). The ABC triangle completely encloses the XYZ triangle.

   Rubric: *5 points for the algebraic argument (no need for calculations). -1 for each small but significant error.*

5 points for the illustration, if the correct ABC vertices are plotted and labeled. -1 for <u>each</u> incorrectly located pair of (x, y) coordinates. -1 (overall) for not indicating the coordinates on the plot (an axis marking is enough), plus -1 for each vertex not in the approximately correct position.

3 points for either of the correct arguments above, but claiming the ABC gamut is "greater than", or "at least as large as", or "greater than or equal to" (i.e. omitting to note that the gamut is the set of visible colours).

0 for stating/arguing that it is smaller than the XYZ gamut.

2. **(10 = 5 x 2 points)** True or false? No need to justify your answers.

   a. For a given camera-to-subject distance, the image captured by a telephoto lens is a crop of the image captured by a wide angle lens.

   True. We clarified during the exam that any difference in resolution because of the physical act of cropping is not relevant to the question.

   b. A sensor with a Bayer filter, and an otherwise identical one with no such filter, require the same exposure for the same scene.

   False. The Bayer filter cuts out 2/3 of the incident light on average, and so requires 3 times more exposure.

   c. The CMYK primaries can uniquely represent every color in the RGB gamut.

   False. The crucial word is "uniquely". Because of the extra black (K) component, the same RGB colour can have many different CMYK representations.

   d. A string of heads and tails produced by 1 million tosses of a perfectly random (unbiased) coin necessarily has high Kolmogorov complexity.

   False. The string almost *always has high complexity (see the practice midsem), but you can also occasionally produce a low-complexity string like all heads or all tails.*

   e. The wavelet transform allows perfect reconstruction of the original signal if there are no numerical inaccuracies.

   True.

   ***Rubric**: 2 points per question, all or nothing. No explanations required.*

3. **(10 points)** Basit observes that discrete convolution can be used to implement many interesting image processing operations.

   a. (4 points) Write a 3x3 convolution kernel which will detect vertical edges in an image, but not horizontal edges. (There are many possible correct answers, all are ok.)

   Here's one possible solution: $\begin{bmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{bmatrix}$

   Or if you want to be more fancy, here's the Sobel operator for the X-derivative with Y-smoothing: $\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$

   The negatives of such matrices are also ok: we'll assume we take the absolute value of the result of the convolution.

   ***Rubric**: An acceptable matrix should have:*
   (a) a positive-to-negative or negative-to-positive change between any two components in at least the central row, AND
   (b) no such change in any column, AND
   (c) the entire matrix summing up to zero.

   Of course not all such matrices will work equally well in practice, but the idea is important here. 2 points for satisfying (a), <u>plus</u> 1 point for satisfying (b), <u>plus</u> 1 point for satisfying (c).

   If the answer confuses vertical edges with horizontal edges (i.e. swap rows and columns above), -2 points in addition to scheme above.

   b. (3 points) Write a 3x3 convolution kernel to brighten all pixels in an image by a factor of 2.

   $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

   ***Rubric:** 3 points for the above answer. 1 point for additional 2's elsewhere in the matrix (the central 2 must be present for any points at all).*

   c. (3 points) Extending (b), can convolution be used to implement *any* color-space filter? Briefly justify your answer.

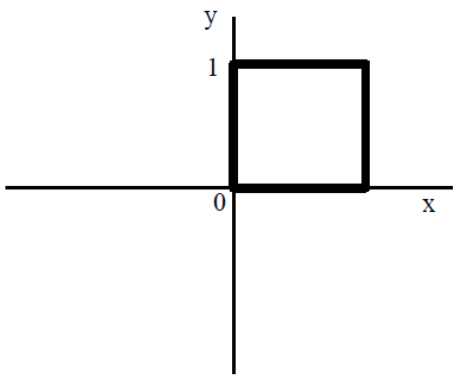
   No, non-linear operations can't be implemented by just multiplying by scalars.

   ***Rubric**: 1 point for writing "No", <u>plus</u> 2 points for including a correct explanation (citing a particular counter-example of a non-linear function, e.g. log(intensity), is fine).*

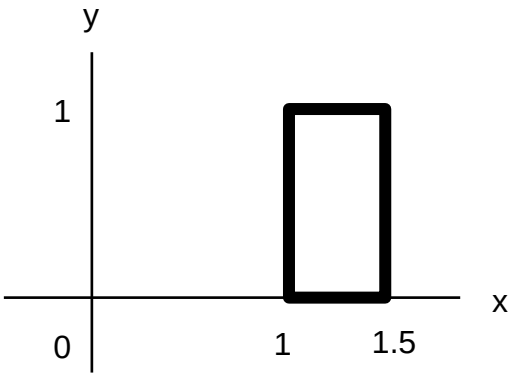
4. **(25 points)** We'll use the following notation for 2D transformation matrices:

   T(x, y): Translates a point by (x, y)
   R(a): Rotates anti-clockwise around the origin by a degrees
   S(sx, sy): Scales the coordinates of a point, relative to the origin, by (sx, sy)

   You are given a unit square with minimum corner (0, 0), as shown below:

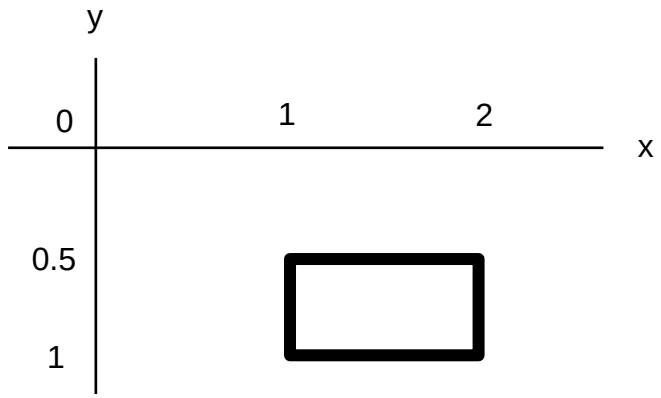


   a. (5 points) Sketch the final result of applying T(1, 1) × S(0.5, 1) × R(–90) to the square. Remember: we apply transforms right-to-left. Don't forget to draw the coordinate axes, with necessary markings!



   ***Rubric**: 2 points for drawing the box correctly along the X axis (1 to 1.5), plus 2 points for drawing it correctly along the Y axis (0 to 1), plus 1 point for adding enough labels (the 1, 1 and 1.5 in the figure, or variations thereof) to make out the coordinates of the corners. In other words, "visual" correctness of the box coordinates gets you 4/5.*

b. (5 points) Sketch the final result of applying R(–90) × S(0.5, 1) × T(1, 1) to the square. Again, don't forget to draw the coordinate axes with necessary markings!



***Rubric**: Same as (a).*

c. (15 = 5 x 3 points) The above sequences demonstrate that multiplication of 2D transformations is not, in general, commutative. However, what if we restricted the transformations to be only of a certain type? In particular, do the following commute under multiplication? In each case, briefly justify your answer (with a counter-example if the answer is no). You may assume the matrices are w.r.t. homogeneous coordinates, i.e. they are 3x3.

i. 2D translations, i.e. all possible T matrices

Commutative. Can be argued by explicitly multiplying two translation matrices and showing the order doesn't matter.

***Rubric**: 3 points for writing out two translation matrices and showing the result of the multiplication is order-independent. 2 points for writing T(a, b) × T(x, y) = T(a + x, b + y) = T(x, y) × T(a, b). 1 point for setting up the argument correctly but making a mistake in the multiplication (2 points if the mistake is minor), or omitting the multiplication entirely, or making a visual/hand-wavy argument. 0 points for claiming the product is non-commutative.*

ii. 2D rotations, i.e. all possible R matrices

Commutative. Can be argued by explicitly multiplying two rotation matrices and showing the order doesn't matter.

***Rubric**: 3 points for writing out two rotation matrices and showing the result of the multiplication is order-independent. 2 points for writing R(a) × R(b) = R(a + b) = R(b) × R(a). 1 point for setting up the*

argument correctly but making a mistake in the multiplication (2 points if the mistake is minor), or making a visual/hand-wavy argument. 0 points for claiming the product is non-commutative.

iii. 2D scalings, i.e. all possible S matrices

Commutative. Can be argued by explicitly multiplying two scaling matrices and showing the order doesn't matter.

***Rubric**: 3 points for writing out two scaling matrices and showing the result of the multiplication is order-independent. 2 points for writing $S(a) \times S(b) = S(ab) = S(b) \times S(a)$. 1 point for setting up the argument correctly but making a mistake in the multiplication (2 points if the mistake is minor), or making a visual/hand-wavy argument. 0 points for claiming the product is non-commutative.*

iv. 2D translations and scalings, i.e. all possible T and S matrices.

Non-commutative. Consider the sequences in parts (a) and (b) without the rotations.

***Rubric**: 3 points for any valid counter-example (visual examples are fine). 2 points for a valid counter-example without indicating the results (either visually or algebraically). 1 point for saying "non-commutative" without an argument. 0 points for claiming the product is commutative.*

v. 2D rotations and scalings, i.e. all possible R and S matrices.

Non-commutative. Consider the sequences in parts (a) and (b) without the translations.

***Rubric**: 3 points for any valid counter-example (visual examples are fine). 2 points for a valid counter-example without indicating the results (either visually or algebraically). 1 point for saying "non-commutative" without an argument. 0 points for claiming the product is commutative.*

5. **(10 points)** Recall that there are three types of shader-related variables: ***uniform***, ***attribute*** and ***varying***.
   a. **(6 = 3 x 2 points)** For each type of variable, indicate whether (1), (2) or (3) is the most appropriate location to <u>set</u> it.

```
// VertexShader.glsl
void main(){ (1) }

// main.cpp
(2)
glBegin(...);
  (3)
glEnd();
```

Uniform: (2)
Attribute: (3)
Varying: (1)

***Rubric**: 2 points per variable type, all or nothing.*

   b. **(1 point)** Assume a BRDF function can be specified with *K* parameters. If these parameters are constant across a primitive, what type of variable would you use for passing them from the main C++ program to the shader?

Uniform.

***Rubric**: 1 point, all or nothing. No need for an explanation.*

   c. **(3 points)** If the parameters varied across the surface of the primitive, briefly describe how you could handle this within the OpenGL framework (no code required, keep your answer short).

The best way to do this is to create a K-channel texture image containing the BRDF parameters across the surface (each pixel holds the BRDF parameters of the corresponding surface point), and then map it to the surface using texture coordinates.

A generally less good solution is to set the parameters as vertex attributes. They will then be interpolated across the primitive. This cannot model non-linear variations within the primitive (similar to Gouraud shading vs Phong shading).

***Rubric**: 3 points for indicating the texture idea in some form or the other. 2 points for the vertex attributes idea. Other answers evaluated at grader discretion, as long as it is consistent.*

6. **(EXTRA CREDIT, 15 points)** Chinmay wants to build a raytracer that can simulate lens effects. The lens is modeled as a primitive object with a known shape and refractive index. It supports intersection queries.

   Assume the scene already has a lens primitive, with the center of the lens placed at a distance of *D* units from the image plane, along the negative Z axis in camera coordinates. The lens covers a circle of radius *R* in the XY plane (again in camera coordinates). For each pixel on the image plane, Chinmay plans to trace several rays backwards into the scene, each ray aimed at a different part of the lens. His code for the raytracer looks like this (syntactic simplifications made for clarity):

```
vector<Ray> getRaysForPixel(Vec3 pixel_pos_on_image_plane)
{
  int NUM_RAYS_PER_PIXEL = 20;
  ...
}

void main()
{
  for (i = 0; i < HEIGHT; ++i)
    for (j = 0; j < WIDTH; ++j)
    {
      Vec3 p = getPixelPositionOnImagePlane(i, j);
      vector<Ray> rays = getRaysForPixel(p);
      RGB c(0, 0, 0);
      for (Ray r : rays)
        c += traceRay(cameraToWorld(r));

      output_image.setColor(i, j, c / rays.size());
    }
}
```

   a. (EC 10 points) Complete the function getRaysForPixel() so that Chinmay's code works as desired. Your code need not be 100% syntactically correct (i.e. it can be detailed pseudocode). You can assume:

   - the usual math/linear algebra functions are all available
   - the vector<Ray>::push_back(Ray) function adds a new element to the collection
   - *D* and *R* are known constants
   - pixel_pos_on_image_plane is in camera coordinates
   - getRaysForPixel() returns rays in camera coordinates (they are transformed to world coordinates by the cameraToWorld() function in main())

   Remember:

   - every generated ray must pass through the lens
   - the rays must sample the area of the lens (in the XY plane) uniformly
   - you must return a set of NUM_RAYS_PER_PIXEL rays

   (This is an open-ended question. The closer you can get to satisfying these conditions, with enough detail for direct implementation, the better. Just

writing “sample the area uniformly” is not enough. You need to provide more detailed (pseudo-)code.)

Tip: It’s possible to spend a looooong time on this question, so make sure you’ve given yourself enough time to answer the other questions.

Rubric:
+2 for some sort of “divide into cells and shoot rays through cells” idea
+2 for ensuring cells of roughly uniform area (naive use of polar coordinates isn’t great for this)
+2 for ensuring cells are roughly “round” (so overall distribution is uniform)
(Or, direct +6 for using some correct method that doesn’t use cells explicitly. Note that picking NUM_RAYS_PER_PIXEL i.i.d. random samples from the disk doesn’t work for small NUM_RAYS_PER_PIXEL such as 20 – the result isn’t very uniform over the area.)
+2 for ensuring correct number of rays (+1 for approximately correct)
+2 for jittering
–min(score – 2, 3) for not writing code
–1 for generating rays outside the lens
–1 for not fully covering the lens

Other issues like hardcoding constants, incomplete code etc will also incur small penalties.

b. (EC 5 points) Could you simplify your solution if NUM_RAYS_PER_PIXEL was 1 million, instead? Briefly describe how (no need to write code).

Rubric: *3 points for suggesting that picking 1 million independent samples from a uniform distribution over the cross-sectional area of the lens will also yield a sufficiently area-uniform distribution of samples. +2 points for describing* how *to sample from a uniform distribution over a circular disk (rejection sampling is a simple option). Other correct answers are also possible.*