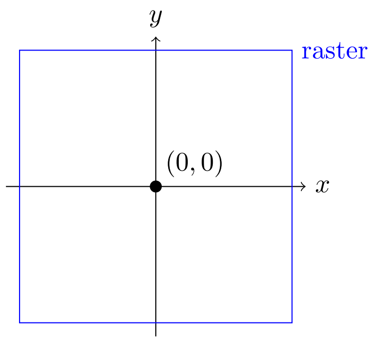
COMP 3490 Assignment 1
Fall 2024

Due date and submission instructions. Submit **one zip file** containing **all** of the pde files required to run your submission. Name your zip file in the style of `VaughanJenniferA1.zip`, but with your name and not my name. All submissions should be uploaded to the Assignment 1 folder on UM Learn by 11:59pm on **Wednesday October 2**.

Getting started. Read these instructions all the way through. Then read the supplied template code, carefully. Several operations are already implemented for you. To start with, make sure you know how to change the current color, and how to set a pixel to the current color using the pixel's (x, y) coordinates.

You must **not** use any of Processing's built-in drawing commands such as `point()`, `line()`, `triangle()`, etc. All images must be created pixel by pixel using **only** the `setPixel()` function given in the template. In Question 1, you will implement Bresenham's line-drawing algorithm using `setPixel()`; thereafter, you can draw lines by calling your implementation.

Conventions. The supplied `setPixel()` function expects to receive (x, y) coordinates using the convention that x and y are measured in pixels, and the origin $(0, 0)$ is at the center of the raster.



For most of the assignment, you will be creating objects in 3D space and using the supplied `projectVertex()` function to transform them into the above coordinate system. We haven't covered the details of that transformation, so you won't know the exact relationship between 3D space and the 2D raster. Here are some guidelines:

- Positive z values are closer to the viewer, and negative z values are further away.
- Objects with x, y and z coordinates ranging from -200 to 200 should show up well on the raster.
- Rotations are performed about the point $(0, 0, 0)$. Put this point at the center of your 3D shape.

Coding standards. Your code will be read, and you should write it with that in mind. Use reasonable variable and function names, and avoid magic numbers. For specific criteria, see Question 6.

Code will be tested with Processing 4.3. Code that does not compile, that does not run, that runs but produces no image, or that crashes often enough that it is difficult to test will receive a score of 0.

You must complete the function stubs provided in the template. You can and should add new functions as needed. You can and should add new data or methods to the Triangle class or edit the existing methods as needed. I strongly recommend adding a class for your 3D shape (e.g., `Hyperboloid`, `Sphere`, or `Cone`). You can add other classes as well, but this is not required.

Do not change any of the hotkeys that are implemented in the template. These will be used for testing. Pay attention to the global flags that they set, and make sure that your code responds appropriately. Do not change the structure of the template code: there are some inelegances, but it works.

Vectors and math. A goal of this assignment is to introduce you to Processing's `PVector` class. Here is the reference entry. You may use all of the `PVector` methods. Caution: some methods return a new vector, while others act in place on the current vector. Make sure you know which one you are using!

Additionally, `max()`, `min()`, `abs()`, `sqrt()`, `sin()`, `cos()` and `Math.pow()` are all allowed.

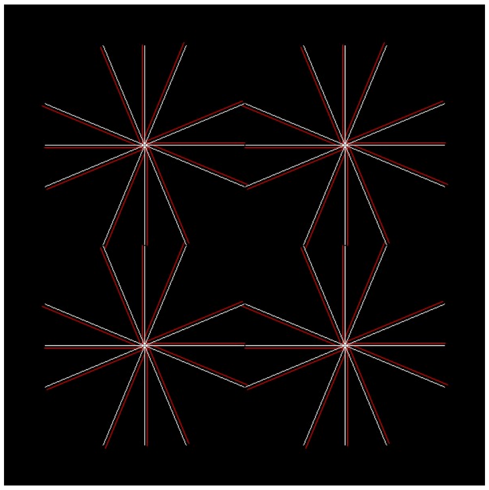
Questions

1. **Implement Bresenham's line algorithm (8 marks)**

   The function `bresenhamLine()` is located in the file named `BresenhamLine`. It receives two sets of 2D raster coordinates: `(fromX, fromY)` for the starting point, and `(toX, toY)` for the ending point. Complete this function so that it draws a line segment on the raster with Bresenham's algorithm, using the specified starting and ending points.

   You can easily go online and find code for this algorithm that is optimized for integer arithmetic. Do **not** submit that version: it will receive no marks. Instead, implement the algorithm precisely as described in class. Your implementation must work for lines in all eight octants.

   The template has a display mode that draws several lines using your implementation, and draws comparison lines using Processing's `line()` function, as shown below. The offset between the red and white lines is done on purpose so that both sets of lines can be seen at the same time.



   Marks: 3 for one case of the algorithm, 5 for extending it to the remaining octants

Interlude: concerning triangles. A triangle in 3D space is defined by the locations of its three vertices. Taken on its own, a triangle is a planar figure. It has a normal vector that can be determined by a 3D cross product of its edge vectors, and that normal vector is the same at every point on the triangle.

However, triangles can be used to approximate a curved surface like a sphere. In that case, each vertex is a point on the surface, and is associated with a normal vector to the *surface* at that point. If the surface is curved, then the normal vectors may be different at each of the three vertices of the triangle.

The `Triangle` class was written with the latter usage in mind. The constructor for this class expects two arrays of vectors: one containing the vertices, and the other containing the corresponding normal vectors.

You can add a constructor to the `Triangle` class that only receives an array of vertices, and treats such a triangle as planar. This is convenient but not required.

The `Triangle` class was also written to facilitate the process of creating rotated copies. Note the method `updateAll()`, which is called at the end of the constructor. If you add more instance variables to the `Triangle` class—and you should definitely do so—put the code that calculates their values into `updateAll()` so that all of your variables are updated as the triangle rotates.

Some instance variables to consider adding to `Triangle`: the edge vectors, the center point, a normal vector at the center point, the projected vertices, the projected edge vectors, ….

2. **Test mode: a single triangle (9 marks)**

   It is very common in graphics to have to work with several different coordinate systems simultaneously, and that's exactly what you will be doing for the rest of the assignment. At each step, it is critical that you know which coordinate system you are currently working in.

   Before we build a more complicated shape, let's start with a single triangle. Create one triangle in 3D space. In `setup()`, initialize `singleTriangle[]` to be a `Triangle` array of length 1, and save your triangle as its only entry. Rotation is set up to run automatically using this array.

   Recommendation: put your triangle in a plane parallel to the xy-plane, but not at $z = 0$. The rotation effect will show up better if the triangle does not cross the origin. Try to make the triangle large, but not so large that the rotation sends it off the raster.

   The order of the vertices matters. Set up the original location of the triangle so that it has right-handed (counterclockwise) winding when viewed from a point far out on the $+z$-axis.

   Treat the triangle as planar: that is, all of its normal vectors are identical.

   Now complete the function `drawTriangles()`, and start to fill in the function `draw2DTriangle()`. Add to the `Triangle` class as needed. By the end of this question, your code should do the following.

   - Use the supplied `projectVertex()` function to project each 3D vertex onto the 2D raster. Then draw the edges of the triangle on the raster using your implementation of Bresenham's algorithm. Switch outlines on or off in response to the hotkey (at this point, turning outlines off leaves a blank canvas).

The function `projectVertex()` uses perspective projection. You don't need to worry about the mathematical details right now: we will cover this topic later. However, you should be aware of what this function does to your data, including the conditions that cause it to return `null`. If the projection culls one of the vertices, omit the triangle.

- Implement a **back-face culling** test that culls degenerate and back-facing triangles using a cross product test. Coordinate system check: this is a 2D calculation and needs to be done with projected vertices and edges.

  Note that degenerate triangles probably will not have an area of precisely 0. Instead, cull any triangle whose area is less than some small value, e.g., about 1 in pixel units.

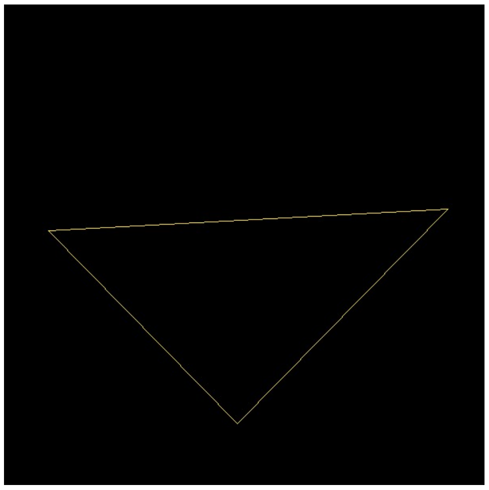
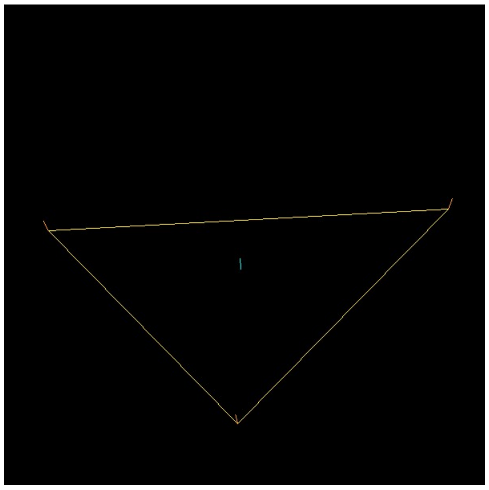
- Complete the function `drawNormals()`. In response to the hotkey, draw short lines in the direction of the normal vectors at each of the triangle's vertices, and at its center point (i.e., at the average of its three vertices).

  For your single planar triangle, the normal vector at the center point is the same as the normal vector everywhere else. However, for compatibility with triangles that represent curved surfaces, calculate the center normal by averaging the three vertex normals.

  Coordinate system check: determining the direction normal to the triangle is a 3D calculation. Use the normal vectors in 3D space to determine the ending points of these lines, then project the starting and ending points into 2D space, and use Bresenham's algorithm to draw them.

Do not let the function `draw2DTriangle()` grow into a hundred-line monstrosity. Break up its many tasks into separate functions. Add data and methods to the `Triangle` class. If there are mathematical operations that you find yourself using multiple times, implement them as separate functions and put them in `MathLib.pde`.

If all goes well, you should now be able to draw images like the ones below.

A triangle in space, with (right) and without (left) normal lines drawn.

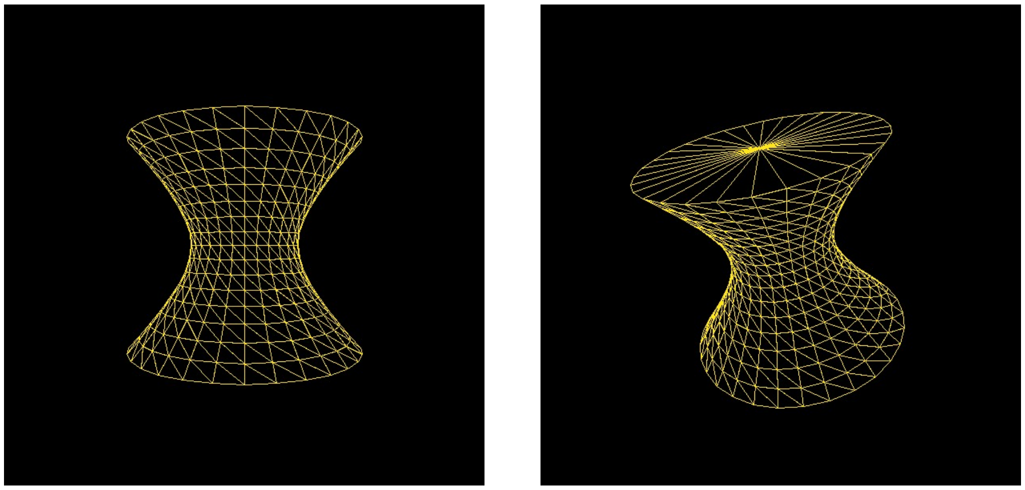
Marks: 4 for triangle edges and normal vectors, 2 for back-face culling, 3 for normal lines. Note that rotation **must work** for testing purposes.

3. **Tessellate a surface in 3D space (8 marks)**

It is also very common in graphics to build more complicated shapes out of triangles. In this question, you will construct a **tessellation** of the shape you wish to draw, using triangles.

Choose one of the following options.

- **Option 1**. A hyperboloid with total height h and minimum radius c.



The equation of the hyperboloid is

$$x^2 + z^2 = y^2 + c^2, \quad -\frac{h}{2} \leq y \leq \frac{h}{2}.$$

Recommendation for tessellation. There are three different surfaces that you need to cover with triangles: the barrel (the curved part), and the two endcaps.

To generate points on the barrel, proceed in steps in the y-direction (vertically from $-\frac{h}{2}$ to $\frac{h}{2}$) and in the θ-direction (horizontally around the rim from 0 to 2π). Given (y, θ), the corresponding point on the hyperboloid is

$$(\sqrt{y^2 + c^2}\sin(\theta), y, \sqrt{y^2 + c^2}\cos(\theta)),$$

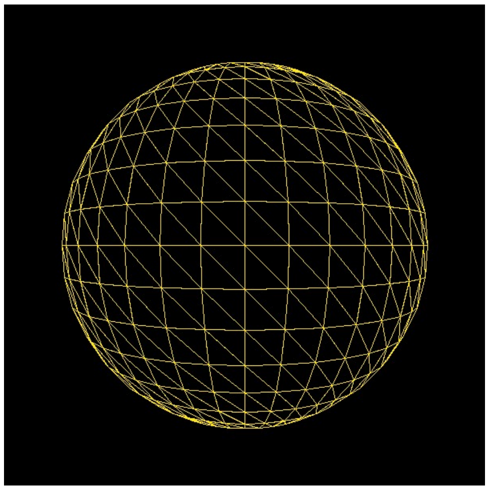
and a normal vector at this point is

$$(\sqrt{y^2 + c^2}\sin(\theta), -y, \sqrt{y^2 + c^2}\cos(\theta)).$$

For my endcaps, I created triangles that have one vertex at the center of the circle and the other two on the rim. You can make further divisions in the radial direction if you wish, but this is not required. Since each endcap is a planar surface, all points on an endcap have identical normal vectors.

Your tessellation should depend on two parameters: n_y, the number of divisions in the y direction; and n_θ, the number of divisions in the θ direction. Your construction should work for any $n_y \geq 2$ and $n_\theta \geq 3$. Create constants that the graders can easily change when they are evaluating your submission.

- **Option 2**. A sphere of radius r.



The equation of the sphere is

$$x^2 + y^2 + z^2 = r^2.$$

Recommendation for tessellation. To generate points on the sphere, proceed in steps in the ϕ-direction (vertically along a line of longitude from 0 to π) and in the θ-direction (horizontally around the equator from 0 to 2π). Given (ϕ, θ), the corresponding point on the sphere is

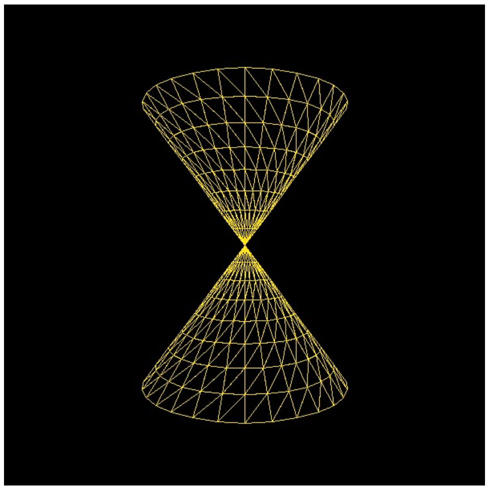
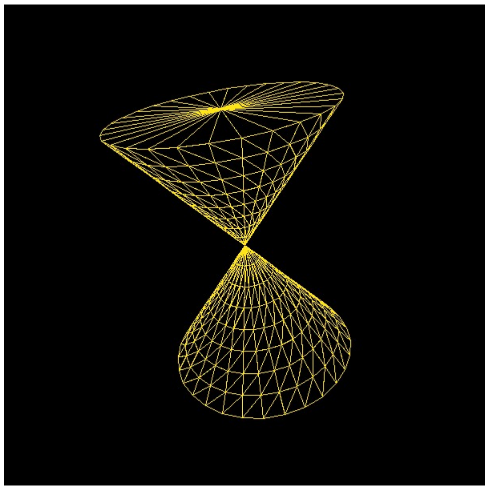
$$(r\sin(\phi)\cos(\theta), r\cos(\phi), r\sin(\phi)\sin(\theta)),$$

and a normal vector at this point is

$$(\sin(\phi)\cos(\theta), \cos(\phi), \sin(\phi)\sin(\theta)).$$

Your tessellation should depend on two parameters: n_ϕ, the number of divisions in the ϕ direction; and n_θ, the number of divisions in the θ direction. Your construction should work for any $n_\phi \geq 2$ and $n_\theta \geq 3$. Create constants that the graders can easily change when they are evaluating your submission.

- **Option 3**. A cone with height h and radius-height ratio c.

The equation of the cone is

$$x^2 + z^2 = c^2 y^2, \quad -\frac{h}{2} \leq y \leq \frac{h}{2}.$$

Recommendation for tessellation. As with the hyperboloid, you will need to tessellate the barrel and the two endcaps.

To generate points on the barrel, proceed in steps in the y-direction (vertically from $-\frac{h}{2}$ to $\frac{h}{2}$), and in the θ-direction (horizontally around the rim from 0 to 2π). Given (y, θ) where $y \neq 0$, the corresponding point on the cone is

$$(|y| c \sin(\theta), y, |y| c \cos(\theta)),$$

and a normal vector at this point is

$$(|y| \sin(\theta), -cy, |y| \cos(\theta)).$$

The vertex of the cone is located at $(0, 0, 0)$. You will have to think about how you want to handle normal vectors at that point.

The endcaps can be constructed in the same manner as for the hyperboloid.

Your tessellation should depend on two parameters: n_y, the number of divisions in the y direction; and n_θ, the number of divisions in the θ direction. Your construction should work for any $n_y \geq 2$ and $n_\theta \geq 3$. Create constants that the graders can easily change when they are evaluating your submission.

This option is the trickiest one, in my opinion, because the vertex requires special handling.

Whichever option you choose, split up the construction of the tessellation into two parts.

Part 1: Vertices and normal vectors. Generate the points that will be the vertices of the triangles, and save them in some convenient data structure (my solution uses a multi-dimensional `PVector` array). Make a plan on paper first, then implement your plan systematically. Incomprehensible code will lose marks.

Also generate and store the normal vectors that you will need.

Part 2: Triangles. Loop over your vertices and normal vectors to construct the triangles that will cover the surface. Remember that the order of the vertices matters. Every triangle should have CCW (right-handed) winding as viewed from outside the surface.

For the most part, you can think of breaking up the curved portion of the surface into rectangles, then further dividing each rectangle in half along a diagonal to form triangles (see if you can identify this pattern in the images above). However, there are regions on the sphere and on the cone that are exceptions to that rule.

In `setup()`, initialize `surfaceTessellation` to hold your completed tessellation. Since you have already implemented `drawTriangles()` and `draw2DTriangle()`, switching to the `SURFACE` drawing mode should draw and rotate your surface automatically.

Comment. Since your Bresenham function uses floating-point arithmetic, there may be some imperfections in the result (e.g., 1-pixel gaps, doubled lines). Don't worry about tracking all of these down: the proper solution would be to convert the line-drawing algorithm to integer arithmetic, and we're not doing that here. As long as the flaws are minor, no marks will be deducted.

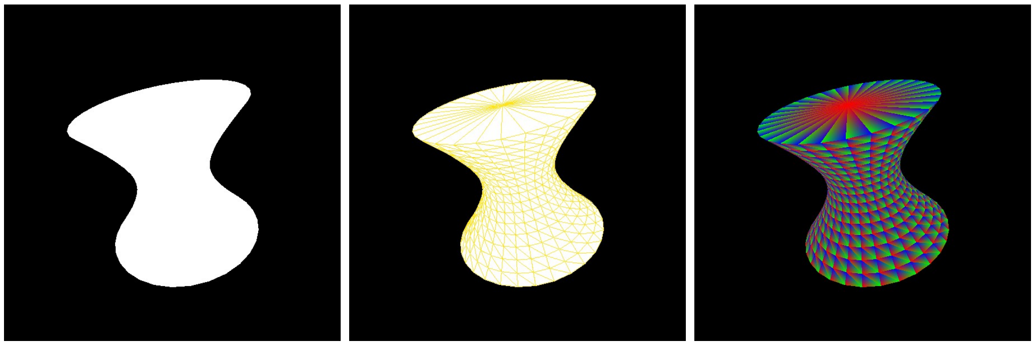
Marks: 3 for vertices, 5 for triangles.

4. **Scan-line fill of triangles (10 marks)**

Complete the function `fillTriangle()` that fills a triangle using the scan-line fill algorithm and the point-in-triangle test, and call this function when you are drawing your triangles. Read the `DrawingModes` file carefully and make sure your code is compatible with all of the different drawing and shading modes. You can test your implementation on the single triangle or the 3D shape, but note that there are some details you might miss if you are only drawing one triangle.

For the `FLAT` shading mode, you will simply give each triangle a uniform color (given in the template – see `GraphicsLib`). In `SURFACE` drawing mode, if the outline is removed, the result should be a silhouette with no gaps.

For the `BARYCENTRIC` shading mode, visualize the barycentric coordinates (u, v, w) at each point by using them as (r, g, b) values.



Flat fill and no outline, flat fill with outline, barycentric fill with no outline.

Comment. As before, minor flaws such as occasional 1-pixel gaps between outline and fill can be ignored. However, pervasive gaps or flickering will result in deductions.

Marks: 5 for scan-line algorithm, 2 for flat mode, 3 for barycentric mode.

5. **Phong lighting, Gouraud shading, Phong shading (9 marks)**

Now you will implement some of the more advanced lighting and shading models.

General remarks about Phong lighting. Recall from class that the Phong lighting model has the form

$$\text{light}_j = m_A A_j + m_D D_j + m_S S_j,$$

where j ranges over (r, g, b); where A_j, D_j and S_j are the ambient, diffuse, and specular terms, respectively, for color j; and where m_A, m_D and m_S represent the fraction of light of that type that is reflected by the object.

In the file `GraphicsLib`, you are given the arrays `MATERIAL` and `PHONG_COLORS`.

- The coefficients m_A, m_D and m_S are stored in `MATERIAL[A]`, `MATERIAL[D]` and `MATERIAL[S]`, respectively.
- The array `PHONG_COLORS[A]` contains the (r, g, b) values of the ambient color of the light incident on the object. Similarly, the diffuse and specular (r, g, b) values of the incident light are contained in the remaining entries in `PHONG_COLORS`.

Look through the global constants defined in `GraphicsLib` for the other quantities you need for your Phong lighting calculations. In particular, to speed up your program, only calculate the specular term if $\widehat{R} \cdot \widehat{V} >$ `SPECULAR_FLOOR`.

In the template, complete the function `phong()` so that it returns the result of the Phong lighting calculation at the given point, using the given parameter values.

Coordinate system check: lighting is a 3D problem. In each of the following shading models, the vectors you pass to `phong()` should come from the 3D rotated triangles.

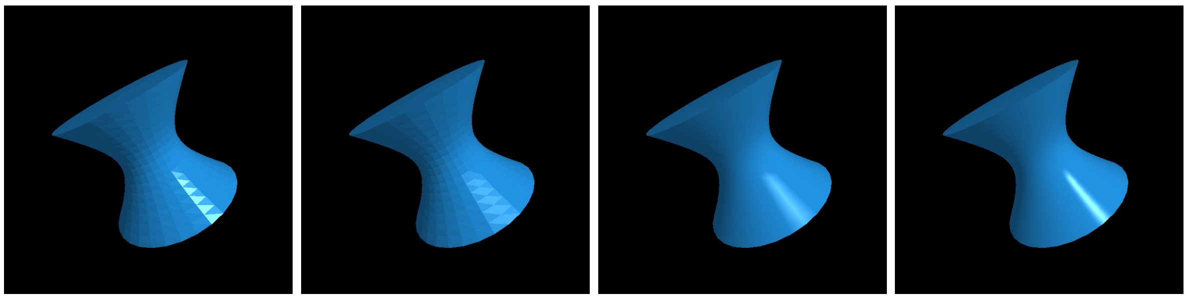
Part 1: Face-level Phong lighting. In this context, “face” refers to the face of a triangle. Calculate Phong lighting at the center of each triangle, and assign this color to the entire triangle.

The location of the center of the triangle is found by averaging the three vertices, and the normal vector at this point is found by averaging the three vertex normal vectors.

Part 2: Vertex-level Phong lighting. Calculate Phong lighting at each vertex of the triangle, average the results, and assign the averaged color to the entire triangle.

Part 3: Gourand shading. This should be a relatively minor modification to the code you already wrote for barycentric shading in Question 4. Now, instead of using the barycentric coordinates u, v and w as (r, g, b) values, use them as weights for the three Phong lighting colors you calculated at the vertices of the triangle. Assign the weighted average of the vertex colors to each pixel, using that pixel’s barycentric coordinates.

Part 4: Phong shading. Finally, implement Phong shading: for each triangle, use barycentric coordinates to interpolate between the three vertex normals and obtain a normal vector at each point inside the triangle. Use the position of each pixel and the corresponding interpolated normal vector in the calculation of Phong lighting, one pixel at a time.



Left to right: face-level Phong, vertex-level Phong, Gouraud shading, and Phong shading.

Same number of triangles in all cases!

Marks: 3 for face-level, 2 for vertex-level, 2 for Gouraud, 2 for Phong.

6. **Tidy Up (4 marks)**. Reread your code. Imagine the experience of a person who is not you, reading your code. Make that experience better. In particular, the graders will be looking for the following.
   - Are there mathematical operations that you need that are not `PVector` methods? Write functions for them and put them in `MathLib`.
   - Does a literal like 3 appear when you are referring to a number of components? Create a constant (or check to see if one already exists) and use that instead.
   - Do you have the same line of code repeated three times because you need to do some calculation for each edge or each vertex of a triangle? Rewrite using loops and arrays.
   - Do you have functions that are unreasonably long? Break them up into individual tasks where possible.