CSci 490, Spring 2005, Assignment 5

This assignment, worth 50 points, is due at 3pm, Friday, March 4*. Submit it by attaching your modified files to an e-mail to* `cburch@cburch.com`*.*

In Assignment 3, you modified a Java graphics system that drew wire-frame drawings in orthographic projections so that it would support projective projections too. In this assignment, you'll modify the program so that it draws polygons instead of wire-frame drawings. Besides implementing a technique for scan-converting polygons, this will also require the addition of depth-testing.

You can download the overall code via the Web page. (It is important that you use this rather than your Assignment 3 solution, because I have modified the library to including support for lighting, as well as using a different technique for rapidly drawing pixels on the screen.)

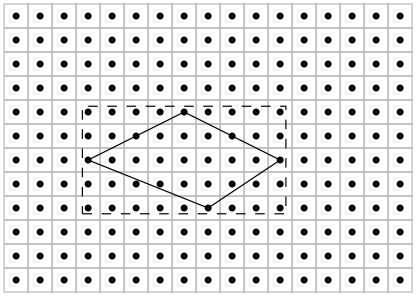
As it stands, the Graphics3D class's `drawPolygon` method draws nothing; as a result, nothing will appear on the screen. Your job is to modify the Graphics3D class so that

From the Web page you can download a set of Java classes that implement wire-frame 3D graphics.[1] Currently, it performs an orthographic projection; but in this assignment, you will modify it to use a perspective projection instead.

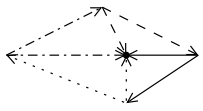
Part I (35 pts)

You should first modify Graphics3D to scan-convert polygons. Note that Graphics3D has an integer-array instance variable named `pixels`. This array is in the same format as we used in Assignment 1: It is a one-dimensional array, with one integer per pixel, with the entries listed in row-order. Each integer encodes the corresponding pixel's opacity, red, green, and blue components.

In your `drawPolygon` method, you should place the appropriate values into the `pixels` array so that the polygons show up accurately in the window. I recommend that you use the brute-force technique for doing this. The brute-force technique first computes the pixel coordinates of the bounding box within the window; then it iterates through each pixel within the bounding box, testing whether it lies within the polygon, and drawing those pixels that do lie within.



One way to test whether a point lies within a two-dimensional polygon (as you have on the screen), you can go around the polygon in order. For each edge, you can take a vector representing that edge, cross it with the vector from an endpoint to the query point, and check the sign of the z-coordinate to determine whether the point lies to the left or to the right of the edge. If all the cross products have z-coordinates with the same sign (all positive, or all negative), then the point lies on the same side of all the edges — that is, the point is within the polygon. If two cross-products have different signs, then the point is outside the polygon.



This technique works only for convex polygons, but your program, like OpenGL, can assume that each polygon is convex.

[1] As I write this, I have yet to proofread the code and accompanying documentation. It will, however, be available by Friday afternoon, February 4.

The distributed code includes a line computing the proper color for the polygon. However, for Part I, I definitely recommend that you color all pixels falling within the polygon *white* instead. As a result, you should see the silhouette of a rotatable cube. The reason for the recommendation of showing the silouette only is that without depth-testing (which you will implement in Part II), the polygons will be the one that were last drawn, which may not be the ones occurring in the front.

For this part, and for the next, you should pay some attention to efficiency. In particular, you should be careful to lift any computation that is executed repeatedly out of the loops. As an example, testing whether a point lies within the polygon involves computing a vector going along each edge. This vector is the same for every point in the bounding box, so your code should really compute this vector before ever entering the loop. While I won't scrutinize your program too carefully on this point, I will deduct credit for computations like this that are glaringly redundant.

Part II **(15 pts)**

Now implement depth-testing. For depth-testing, you will want to add a depth buffer to the Graphics3D object.

With each pixel to be added, you will need to compute the pseudodepth of that pixel. Computing this is easy if you know the equation of the plane through the polygon (as it exists through the three-dimensional coordinates coming out of the end of the pipeline). This plane's equation will be of the form

$$Ax + By + Cz + D = 0\,.$$

Using $\mathbf{n}$ to represent the plane's normal, this can be written instead as

$$\mathbf{n} \cdot \mathbf{P} + D = 0\,.$$

All we need to determine is $\mathbf{n}$ and D so that the above is true for the vertices of the polygon.

Computing the normal is easy: Take two vectors going along adjacent edges of the polygon, and compute their cross-product. Once you have this, then you can compute D by computing $-\mathbf{N} \cdot \mathbf{V}$ for some vertex $\mathbf{V}$ on the polygon.

As always, I will be very happy to explain any concepts relating to the assignment or other topics from class.