CSCI 111, Lab 11
3d height field visualizer

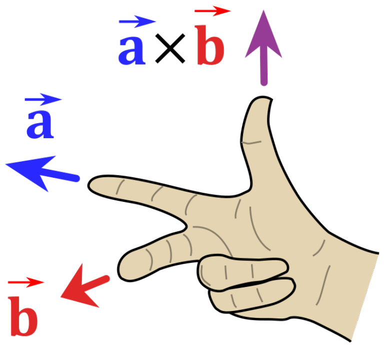
Due date: Midnight, Tuesday, December 6, on Canvas. No late work accepted.

File names: Names of files, functions, and variables, when specified, must be EXACTLY as specified. This includes simple mistakes such as capitalization.

Individual work: All work must be your own. Do not share code with anyone other than the instructor and teaching assistants. This includes looking over shoulders at screens with the code open. You may discuss ideas, algorithms, approaches, *etc.* with other students but NEVER actual code.

3d vector type: You will need your solution to Lab 10, the vector module, for this assignment. Add the following method to the class. It returns the cross product of two vectors, a vector perpendicular to both.

```
    def cross(self, other):
        a1, a2, a3 = self.x, self.y, self.z
        b1, b2, b3 = other.x, other.y, other.z
        s1 = a2*b3 - a3*b2
        s2 = a3*b1 - a1*b3
        s3 = a1*b2 - a2*b1
        return V(s1,s2,s3)
```



If you didn't finish this project you can use my implementation, available on the lab website.

The program In the folder `poly3d` on the lab website you can find a skeleton of this program. The main program, `poly3d.py` is complete, as well as the `vector.py`, `functions.py`, `gradient.py` and `polygon.py` modules.

You will only have to complete the `parametricsurface.py` module.

Step 1, heightfield: Start with implementing

```
    def makeHeightfield(self):
        """calculate nxn 3d points in xrange x yrange
           store V(x,y,z) in self.heightfield[i,j]"""
```

This should create an empty dictionary. Loop over i and j in range(n), and lerp i and j into the xrange and yrange, getting x and y in parameter space. Applying the function to x and y gives a point p in world space. Store p in self.heightfield[i,j]

Test this with a simple function, for example x+y, in the range 1..10 with n=10. Print out the heightfield so you can see it's working.

Step 2, project points: Next implement

```
    def projectPoints(self, eye):
        """project 3d points in heightfield to
              2d points (x,y) in plane of camera
              store in self.points[i,j]
           also calculate for these points
              self.minx, maxx, miny, maxy
           also calculate color based on gradient and height
              store in self.color[i,j]
           also calculate eye distance
              store in self.distance[i,j]
        """
```

Project the 3d points in `sefl.heightfield` using the eye point.

Create a frame of vectors around the eye, forward, up, and right:

```
    fwd = -eye
    up = V(0,0,1)
    rt = fwd.cross(up)
    fwd.gram_schmidt(up, rt)
```

Then use the Gram-Schmidt procedure to make them The method to make an orthonormal frame around the eye is part of the `Poly3d` object.

If pt is our 3d point from the heightfield, the x coordinate of our pt will be the projection of a vector to the point onto the right-pointing vector and the y coordinate will be the projection onto the up-pointing vector:

```
x = (pt - eye) * rt
y = (pt - eye) * up
```

For each [i,j] store the x,y tuple in `self.points[i,j]`.

Also calculate the distance from the point to the eye. This is just the length of the vector from the point to the eye, `(pt - eye).length()`. Store these in `self.distance[i,j]`

Also, since we're calculating every x,y value for every point, we can also remember the minima and maxima for both x and y. Save these, as well, in `self.xmin` etc.

Test this on some simple heightframes with some simple eyes. For example, what would you expect if the eye is V(1,0,0)? What about V(0,1,0)?

Step 3, scale points: Now implement the function

```
    def scale(self, size):
        '''lerp the polygons from xrange x yrange into 0..w x h..0
           can shrink the screen range 10% if you like
        '''
```

We start with a set of 2d points stored in `self.points`. These are the points in the camera frame, and their limits are stored in `self.xmin`, `self.xmax`, *etc.* We want to lerp these points into the ranges (0,width-1) and (height-1,0), where `width,height = size`

Loop over all i and j and for all points in `self.points[i,j]`, lerp the x value from `(self.xmin, self.xmax)` into the range `(0, width-1)`, and the y value into the range `(height-1, 0)`. You can store the result right back in `self.points[i,j]`.

We now have a set of points in screen coordinates, and we're ready to plot them.

Step 4, make polygons: Now finish the function

```
    def makePolygons(self, eye, size):
        """
           then build polygons from self.points, self.color, self.
  distance
           using [i,j], [i+1,j], [i+1,j+1], [i,j+1] indices
           then sort the polygons based on distance (farthest first)
        """
        self.projectPoints(eye)
        self.scale(size)
```

This function will be called from another object to create the polygons from a height-field. It first calls `projectPoints` and then `scale` to get points in screen space, ready to be drawn.

Now make polygons out of the projected points. For each point with i,j in range(n-1), use the points at [i,j], [i+1,j], [i+1,j+1], [i,j+1] to make a polygon. Use the average of these four distances as the distance to the polygon. You can use blue, `'#0000ff'` for the color.

Sort your polygons by distance, farthest first.

The best way to test now is to rerun the `poly3d` module, which will plot all your polygons. Not good? Debug!

Step 5, add color: Go back to `projectPoints`. Use the gradient object to find the color of the point, based on the z value in 3d. Store these in `self.color[i,j]`

Now go back to `makePolygons` and add this color to the polygon, instead of blue. You can average the four colors from the four corners, or just use one of the four colors.

Replot. Colorful?

Step 6, shading: Go back to `projectPoints`, where you found the color using the gradient and the z value of the 3d point.

Get three points from the heightfield, `p0` at [i,j], `p1` at [i+1,j], and `p2` at [i, j+1].

Warning! You can't use these when i or j are equal to n-1. It will be good enough if you just replace i with i-1 or j with j-1 in this case.

Now find two **tangent vectors** to the surface, `v1 = p1-p0` and `v2 = p2-p0`. Find the **normal** to the surface at [i,j] using the cross product of the vectors, `normal = v1.cross (v2)`. Normalize the length of this vector.

Now take the dot product of the normal with a normalized vector pointing toward the light. You can put the light anywhere you want. Mine is in this direction: `V(2,-1,4)` . Remember to normalize the light vector!

The dot product of the normal and the light gives you the shade value, but it if's less than 0.25 use 0.25 instead of the value. In other words,

```
shade = max(0.25, normal * light)
```

Use this shade value to darken the color. It's an optional parameter to the gradient object.

Optional, moving on: Colorize with a different gradient!

Find some interesting parametric surfaces!