For this assignment, you will implement a simple 3D scene that the user can navigate around. Your scene will be constructed from a 2D height field, along with 2D color data. The user can navigate around the scene using the familiar a, s, d and w keys. When it’s working, your program will look something like this:

![3D Scene Example](image)

### Program Organization

I am giving you a skeleton program to get you started. Most of the logic for the program is in a class called World. The world map will be stored in a class called WorldMap. You should keep your implementation in these two files, but you will need to add to both of them to get your program working. You can choose how much of the implementation goes in the World component vs the WorldMap component. I suggest you try to make the WorldMap component reusable and put application-specific logic into the World component.

I am also giving you code for a Geometry component. Geometry.h and Geometry.cpp contain a simple struct that can be used as a point/vector representation. It also includes member functions and overloaded operators to make it easy to perform common operations like dot product, cross product, vector addition and drawing via OpenGL.

### Scene Description and Startup

The geometry and appearance of the scene is maintained by a class called WorldMap (WorldMap.cpp and WorldMap.h). I’m giving you a starting implementation for this class. At startup, your program will use an instance of WorldMap to read height data from a greyscale image in PGM format. The basename for this file (i.e., without the .pgm extension) must be given on the command line at startup. The program will also read vertex color data from a PPM file with matching dimensions and a matching name basename.

The r by c greyscale height image will be interpreted as a height field r vertices in the Z dimension and c vertices in the X dimension. At image load time, intensity values in this image are converted to height values in a range chosen at compile time. Darker values are lower elevations, and lighter values are considered
higher elevations. The color image data from the PPM file is interpreted as corresponding color data for the vertices in the height field.

Scene Display

We interpret height and color data in the WorldMap as a 2D array of vertices spaced at unit intervals in the \( X \) and \( Z \) directions. The height of each vertex is determined by its height value, and the color of each is determined by the red, green and blue values. You should connect neighboring vertices by a quadrilateral.

You should display the scene with a perspective view from the user’s current location. To limit the computational cost, you will display only up to a \( 50 \times 50 \) quadrilateral region of the world map around the user’s location. To be able to see what’s around you, you will want a near clipping distance of less than one and a far clipping distance of about 25.

Fog

To hide the fact that we’re only drawing landscape around the user, we are going to use a little bit of linear fog to make the scene fade away in the distance. The following code fragment will turn on fog that completely obscures drawing that’s 25 units from the viewer. To get fog to work properly, you will need to put your camera positioning transformation in the MODELVIEW matrix and your projection in the PROJECTION matrix.

```c
// Turn on fog.
glEnable( GL_FOG );

// Use fog that varies linearly with distance.
glFogi( GL_FOG_MODE, GL_LINEAR );

// Use white fog (feel free to use any color you want)
GLfloat fcolor[] = { 1, 1, 1, 1 };
glFogfv( GL_FOG_COLOR, fcolor );

// Fog starts at a distance of 15 and saturates by a distance of 25.
glFogf( GL_FOG_START, 15 );
glFogf( GL_FOG_END, 25 );
```

Fog should be an optional feature in your program. Pressing the \( f \) key should toggle it on and off.

Quadrilateral Shading

The base color for each vertex will be based on the color data for that vertex in the WorldMap. To help the user in understanding the geometry of the scene, we will shade quadrilaterals based on a simple, diffuse reflection model. To do this, you will need to compute an approximate surface normal for each polygon. Since our quadrilaterals may not be geometrically flat, we will need to compute an approximate surface normal. The Newell method will do this nicely for us. If \( [x_0, y_0, z_0]^T \ldots [x_{N-1}, y_{N-1}, z_{N-1}]^T \) are the vertex locations for a polygon, an average surface normal, \( [n_x, n_y, n_z]^T \), can be computed as follows (understanding that \( i + 1 \) is computed modulo \( N \)):

\[
n_x = \sum_{i=0}^{N-1} (y_i - y_{i+1})(z_i + z_{i+1})
\]
\[
\begin{align*}
\mathbf{n}_y &= \sum_{i=0}^{N-1} (z_i - z_{i+1})(x_i + x_{i+1}) \\
\mathbf{n}_z &= \sum_{i=0}^{N-1} (x_i - x_{i+1})(y_i + y_{i+1})
\end{align*}
\]

Note that this computation will typically not yield a unit vector. Once you have computed the surface normal for a quad, compute the cosine of the angle between that normal and a fixed light direction of \([0.267261, 0.534522, 0.801784]\). Clamp this value to the \([0, 1]\) range and then use it to scale the base colors at each vertex of the quad. This gives the surface a diffuse shaded appearance.

**User Interaction**

While the program is running, the user can move the virtual camera around the landscape. I’m providing you with data members in the World class and glut callbacks to help support this. The vLoc member keeps up with the user’s current location in the X and Z direction, and the vDir field keeps up with the direction the user is facing. When the user presses the a, s, d and w keys, these fields are updated to move and rotate the user. I am also providing code to keep the user inside the map area. As you can see from my code, the user will always be on one of the polygons in the map.

As the user moves around, you will need to compute the proper height value to keep the camera one unit above the height field. The user moves slow enough that several steps are required to move across the surface of a single polygon. Thus, you will need to compute intermediate height values as the user moves over a polygon. This is a good job for bi-linear interpolation.

We just have to compute a weighted combination of the height values at the four corners of the quadrilateral. Vertices are spaced regularly, at integer coordinates in the X and Z directions. We can use the fractional part of the user’s current location in X and Z to compute a weighting that varies gradually as the user moves across the surface of the quad. Let \(wx\) be the fractional part of the user’s X location, and let \(wz\) be the fractional part of the user’s Z location.

\[
\begin{align*}
&\text{If } h_1, \ldots h_4 \text{ are the heights of four vertices around the quad that the user should be standing on, we can} \\
&\text{compute an interpolated height as:}
\end{align*}
\]

\[
h = h_1 (1 - wx) (1 - wz) + h_2 (1 - wx) wz + h_3 wx wz + h_4 wx (1 - wz)
\]

This computation gives us a position right in the quadrilateral surface, effectively, a location for the user’s feet. To make it look like the user is walking with their head above the surface, we’ll put the camera one unit above the height computed here.
**View Direction**

We want the user to look up a little bit when heading uphill and down a little bit when pointing downhill. This will be a little bit tricky, but it will give us a good chance to practice some of the geometric operations we’ve been learning about. It’s easy to compute the user’s location and the direction he is pointing in the XZ plane. We need to compute these for the surface he’s standing on.

As illustrated below, we can compute a look-at point in the XZ plane by simply taking the user’s location in the XZ dimensions and adding a vector corresponding to the current move direction. From the previous section, we know how to compute a location on the plane, where the user should be standing. We just need to compute a look-at point that’s in the plane. To do that, we’ll follow a vertical line from the user’s look-at point in the XZ plane until that line intersects the plane the user is on. We can then use this point as the user’s look-at point (well, almost). We can describe the plane the user is on via the user’s current location and our approximate normal. We can then use our standard technique to compute line/plane intersection to find the look-at point.

We need to do two more things to fix the look-at point. First, we need to move it one unit above the plane, just like we did for camera placement. Second, I’ve found that looking in a direction that’s perfectly parallel to the surface doesn’t actually look that good. It looks better to just look a little bit up when the surface is leaning up and a little bit down when it’s leaning down. To do this, let’s average together a look-at
point that’s computed as described above, and a look at point that has the same $Y$ coordinate as the user’s current location. Effectively, we will be cutting the slope of the view angle in half.

For extra credit, you can make the view direction vary gradually as the user moves over a polygon, rather than jumping to a new angle as soon as the user moves from one polygon to the next. You will compute an average surface normal for each vertex in the world (probably at parse time). You will do this by averaging together the (unit) surface normals for each face incident on a vertex. Then, while the user is moving across a face, you will linearly interpolate the normals to compute a view direction, the same way you average together heights.

**Running and Jumping**

If you want some more extra credit, give the user a little bit of momentum and friction. As the user presses $w$ or $s$, have them gradually speed up to some compile-time speed limit. When the user is not pressing the $s$ or $w$ key, have them gradually slow down as if they are experiencing friction. Maintain the user’s velocity in three dimensions so that, if they hit the top of a hill while going quickly, they will leave the ground and fly through the air for a little while. If you implement this part of the assignment, you should treat the interpolated height of the landscape as a lower bound on the user’s height. The user can fly above the ground, but the camera must always stay at least one unit above the surface.

**Guidelines**

Please observe the following when completing the assignment and submitting your solution:

- Be sure to draw your quadrilateral with the front face pointing up. Remember, if the vertices go in counter-clockwise order, you are looking at the front face. You should be able to turn on backface culling and not see any change in the appearance of your program.

- Students enrolled in the class for graduate credit are required to do the extra credit portion of the assignment.

- As always, be sure to comment and properly indent your code using spaces for indentation rather than hard tab characters.

**Program Submission**

You will need to turn in a printout of your solution along with an electronic copy. Now that I have a working submission system, let’s use it for everything. Go to the course homepage and run the submission client to submit your source code. I’m expecting three files, `World.cpp`, `WorldMap.h` and `WorldMap.cpp`. 