

## Project #3 — See the World

### 1. Project specifications

**Target due date** : Friday, May 30.

**Points** : This project is worth 5 points.

**Teams** : You are welcome to work with a partner as a two-person team. One of you should be “project leader” and the other, “project staff”. As a comment in your program, please say who is which. It is the leader’s responsibility to submit the project. If you don’t have a partner but would like one, let me know and I’ll help to arrange something. Also, as confirmation, the “project staff” member should send me email saying whom he or she is working with as leader.

**Language** : For this project, you need a high-level language such as C, C++, or Java, to produce an output file in a simple format, which is then plotted by a program to be provided to you. This description is for C++ on the Windows NT network. Other options are possible.

**Submitting your program** : The project leader should submit the *source* file (the .cpp file for C++). Use the usual submit folder, under the name of `seeworld.cpp` .

**Grading** : As usual, grading will be pass/fail. However, if your program is well organized and well documented, with good variable names and good use of vectors and matrices, I’ll make a note of it for writing recommendations later.

**Description, in outline** : Imagine the earth as a unit sphere centered at the origin, with the north pole at  $(0, 0, 1)$  and the  $x$ -axis going through the point with latitude  $0^\circ$  and longitude  $0^\circ$ .

- The program should get a specified latitude and longitude in degrees, either from command-line arguments or by reading them from standard input.
- Input to the program consists of a file `continents.dat` that gives outlines of the continents, in terms of points and information about how they are connected.

- The program should find a rotation that moves the the view-point to the point at infinity on the  $z$ -axis (using the up-vector  $\mathbf{k} = (0, 0, 1)$ ). The program should apply this rotation to all data points and then project them on the  $x, y$ -plane. Hidden lines should be removed by simply not drawing a segment at all if either end is hidden.
- The output of the program should consist of the  $x, y$  pairs and connection information, in a simple format described below.
- As mentioned, the actual plotting will be done with a program to be provided to you.

**Input file** : This is the file `H:\class\m149.1\continents.dat`, which describes the outlines of the continents. Each line of this data file has three numbers for one data point. The first number is 0 or 1, where 1 means the point is connected to the previous point and 0 means it is not connected. The second number is the latitude of the point in degrees; the third is the longitude in degrees.

```
0 18.6 120.7
1 18.5 121.4
1 18.5 122.2
...
```

**Output file** : The output file should be in a very simple form, which we may call the “ez format” (ez = easy): Each nonblank line of the output file contains just two numbers representing  $x$  and  $y$  for one point. A blank line means not to connect the preceding point to the following point; otherwise, consecutive lines represent points that are intended to be drawn connected. It is suggested that your output file be named `earth.ez` .

**Arguments** (latitude and longitude of the viewpoint in degrees): The user should be able to enter these at the time the program is run. One good way is to use command-line arguments, so that the user would type the command line

```
seeworld.exe 30 -90 < continents.dat > earth.ez
```

(for example). Below is a discussion of just how to access such arguments in your program.

Another way is to have your program prompt the user for these arguments and read them in from standard input. The only disadvantage is that then you can’t use standard input and output for the data files in and out; instead, you will need to build the names of those files into the program and open them for reading and writing.

**Getting the relevant files** : From the folder `H:\class\m149.1\lab2` you will need the files `continents.dat` and `ez2gs.cmd`; it would be easiest first to make copies of these files in your own folder, the one in which you are working. (To make a copy, use Windows NT Explorer to go to the folder, use the right mouse button to make a copy, go to your working folder, and use the right mouse button to paste.)

**Getting a command line** : From the Start menu, get the Programs menu, and then click on **Command Prompt**. If your working folder is `m149\lab2` in your home folder, type `cd m149\lab2` . Then you can run commands as described.

**Viewing your output** : Assuming you have made shortcuts as recommended and have generated the output file `earth.ez`, you can generate the graphic output by the command `ez2gs.exe earth.ez` on the command line. To center the picture you may need to use the horizontal and vertical scroll bars.

**Summary of steps** : (Let's assume command-line arguments are used.)

**Step S-0.** Make the shortcuts described above.

**Step S-1.** Make your program source file, which on NT should be called `seeworld.cpp`.

**Step S-2.** Compile your source file to make a binary file called `seeworld.exe`.

**Step S-3.** To view the earth from the viewpoint above the point with latitude  $30^\circ$  and longitude  $-90^\circ$ , generate an output data file `earth.dat` by using the DOS command

```
seeworld.exe 30 -90 < continents.dat > earth.ez
```

**Step S-4.** View the file using the DOS command

```
ez2gs.cmd earth.ez
```

After a wait for processing the file, you will get a drawing window in which the picture will be shown.

**Avoiding confusion about angles** : There are two separate uses of latitude and longitude in this program: (1) to give the direction of viewing and (2) to give the location of each point in the data file. In the first case, you'll be using the angles to make a rotation matrix; in the second case, you'll be using them to find the Cartesian coordinates of each point in  $\mathbf{R}^3$ . In both cases, you will need to **change the angles to radians** before doing anything with them. Use descriptive variable names to distinguish uses.

Remember, the *latitude* of a point on the earth is the angle up from the equator, as seen from the center of the earth, and the *longitude* is

the angle east of a line that runs from pole to pole through Greenwich, England.

## 2. More details of your program

Let's assume you use C++ and call your program `seeworld.cpp`. Also, since in C++ the coordinates of a point `p` are `p[0]`, `p[1]`, `p[2]`, for clarity define `x` to be 0, `y` to be 1, and `z` to be 2, so you can write `p[x]`, `p[y]`, `p[z]` for the three coordinates. (But this means you can't use `x`, `y`, `z` later for other things!)

1. Output the points of a unit circle. Specifically, make a loop with a variable (say `i`) going from 0 to 60 (say), use it to have a variable `t` go from 0 to  $2\pi$  in 60 steps, and for each value of `t`, output `cos(t)` and `sin(t)` on one line. Don't omit the point where `i` is 60.
2. Get viewing latitude and longitude from command-line arguments, as described in Section 7 below; convert to radians.
3. Make a suitable rotation matrix `R` taking the viewpoint to infinity on the  $z$ -axis. Specifically,  $R = R_{\pi/2-lat}^{z \rightarrow y} R_{long+\pi/2}^{y \rightarrow x}$ .
4. For each data line:
  - (a) Read data, checking for end of file. (The discussion here assumes you have opened `H:\class\m149.1\continents.dat` as a named file.)
  - (b) Convert angle data to radians.
  - (c) Convert data points to Cartesian coordinates in three dimensions, treating the world as a sphere of radius 1. Specifically, for a point of latitude  $\theta$  and longitude  $\phi$ , Cartesian coordinates are  $(\cos \theta \cos \phi, \cos \theta \sin \phi, \sin \theta)$ .
  - (d) Transform by `R` to get coordinates (say) `p[x]`, `p[y]`, `p[z]`.
  - (e) If the point `p` is visible from above (i.e., if `p[z] > 0`) then
    - i. If the previous point was hidden *or* `p` is not connected to it, write a blank line.
    - ii. `cout << p[x] << " " << p[y] << endl;`(If `p` is hidden, do nothing. If you use nested `if...else{}`'s, remember that C and C++ associate the `else` with the most recent `if` unless you used `{ }` blocks.)

- (f) Save information about visibility of  $p$ , to use for the next point. (Before starting the read-data loop, you'll need to set this information to "hidden" for use by the first data point. Otherwise the first data point will be plotted connected to the circle!)

### 3. Recommended procedures and functions to use

- any needed from the page of suggested matrix procedures;
- one to multiply a matrix times a vector;
- one to turn latitude, longitude in radians into Cartesian coordinates;
- any others that are useful for clarity, e.g., one for degrees to radians.

Please use arrays for vectors and matrices, for example,  $p[]$ ,  $R[][]$ , instead of using a different letter for each vector or matrix entry. This is usually better programming procedure, as it permits vector and matrix calculations to be done with loops rather than many separate statements. Then if there is a bug in the loop, it will be dramatic and will be more apt to be discovered than a bug in one of several separate statements. As mentioned, for clarity you can define  $x,y,z$  to be the indices  $0,1,2$ .

### 4. Suggested phases

- Start with a program to output a circle. Display using the `ez2gs.exe` program, as above.
- Add commands to do 3, 4(a), 4(b) and to write out the Cartesian coordinates. Display results.
- Remove the writes and add 4(c), 4(d), 4(e), 4(f), using the identity matrix for  $R$ , so that you are looking from above the north pole. Display results.
- Add the calculation of arbitrary  $R$  as in 2, 3. Display results.

### 5. Header information

You'll probably need a header line `#include <cmath>` to get type definitions for `sin()` and `cos()`

## 6. Debugging advice

- If you get strange long lines across your picture, check your logic about connections of points. Have you used nested `if` statements properly?
- If your points are bunched up and not spread around the picture, check your transformations. Do your matrix and vector multiplication routines use the indices `i j k` correctly?
- If you get mirror images of the correct continents, check whether your hidden-line test is the right way around.

## 7. Command-line arguments

There are several ways to get specific numbers into your program, say for latitude and longitude: You can write them in explicitly, you can put them in as definitions, you can put them in as initial values of variables, you can compute them, you can read them from standard input using `scanf()`, you can read them from a named file, or you can get them as “arguments” from the command line used to run your program.

For the viewing latitude and longitude, this last way is best.

In the C++ language, the main program can look like

```
main(int argc, char** argv)
{
    view_lat_deg = (argc>1) ? atof(argv[1]) : 0 ;
    view_long_deg = (argc>2) ? atof(argv[2]) : 0 ;
    ...
}
```

*Explanation:* `argv` is a list of strings (`argc` of them), of which the first is the name of the calling program [not needed here] and the rest are the arguments as strings. Thus the first argument is `argv[1]`, as a string of characters even though for this program it represents a number. The `atof()` function converts this alphanumeric string to a floating number. (This function needs `#include <math.h>`, but you will already have included that line because of the sine and cosine functions.) The `?:` expression says to use the given argument if it is present or 0 otherwise.

## 8. More detailed suggestions on using C++

Depending on your computing background and that of your partner, you may wish to write your program on one of several levels. Let’s assume you’re using C++.

## 8.1. Too low-level

You might think of using named coordinate values  $x, y, z$  instead of an array  $p[0], p[1], p[2]$  or a vector class. But then how are you going to represent matrices? And how are you going to do matrix multiplications? If you end up with lots of long, messy expressions, that's bad programming. There are very likely to be mistakes that you don't notice.

## 8.2. Basic level (PIC 10A only, or C-style)

Do use an array such as  $p[0], p[1], p[2]$  for a point, but to keep things straight, follow the suggestion above by defining  $x, y, z$  to be coordinate indices 0,1,2, rather than the actual coordinate values, so that you can say  $p[x], p[y], p[z]$  when you wish but also do loops with  $p[i]$  other times. Since this is the basic level, you might not use functions. Your program might look something like the following. Examine carefully how the matrix multiplication goes; think it through to see how it does correspond to what you do when you multiply matrices by hand.

```
// headers, with <math.h> or <cmath> as appropriate

const int x = 0;
const int y = 1;
const int z = 2;
const double pi = ...;

void main(int argc, char** argv)
{

// ----- get args, convert to radians -----

    double view_lat_deg = (argc>1) ? atof(argv[1]) : 0.0;
    double view_long_deg = (argc>2) ? atof(argv[2]) : 0.0;

    double view_lat_rad = ...;
    double view_long_rad = ...;

// ----- find rotation R taking viewplane to north pole -----

    double rot_long[3][3];
    double rot_lat[3][3];

    rot_long[x][x] = rot_long[y][y] = cos(view_long_rad + pi/2);
    rot_long[x][y] = ...;
```

```

        //etc. to complete rot_long

rot_lat[x][x] = ...
        //etc. to complete rot_lat

double R[3][3];

for (int i=0; i<3; i++)
    for (int j=0; j<3; j++)
    {
        R[i][j] = 0.0;
        for (int k=0; k<3; k++)
            R[i][j] += rot_lat[k][j] * rot_long[i][k];
    }

// ----- draw circle -----

for (int i=0; i<=60; i++)
{
    double t = 2*pi*i/60.; // better use 60., not 60 (why?)
    cout << ... // write out cos(t), space, sin(t)
}
cout << endl;

// ----- main loop -----

int connect;
double data_lat_deg, data_long_deg;
double data_lat_rad, data_long_rad;
double data_pt[3];
double p[3];

// ----- read in data, convert to Euclidean -----

(open stream fin to read continents file)

if (!fin)
{
    cerr << "can't open continents file for reading" << endl;
    exit(-1);
}

```

```

while ( fin >> connect >> data_lat_deg >> data_long_deg )
{
    // convert to data_lat_rad, data_long_rad  in radians;
    // expressions on right will involve sines, cosines

    data_pt[x] = ...
    data_pt[y] = ...
    data_pt[z] = ...

    // rotate the data point: p = R * data_pt

    for (int j=0; j<3; j++)
    { // call outer index j since entry of row vector is column
        p[j] = 0.0;
        for (int i=0; i<3; i++)
            p[j] += R[i][j] * data_pt[i];
    }
    // ----- check visibility, etc. -----

    // (see earlier part of this handout)
}
}

```

Even if you intend to work at the 10A level, please read the next part.

### 8.3. Using more advanced ideas

**Functions** The program above is crying out for some functions. Too many pieces of it are repetitive—converting degrees to radians, making rotation matrices, etc. Repetition leads to small editing errors that are easy to miss.

[Classes] It isn't easy to have a function return an array as a value; you'd need to allocate the space for the array. It's easier to use classes.

Some class declarations are suggested below. To keep things easy, let's use a vector class `vec3` and a matrix class `mat33` for three dimensions only, which means that memory allocation issues can be avoided entirely.

With a class, you can still define `[]` so the result looks like an array: `R[i][j]` and so on. You can also define `*` for matrices times matrices and vectors times matrices so that you can write `R = A*B` for matrices.

Suggested classes, in outline:

```
class vec3
{
    double ent[3];          // entries
    friend ostream& operator<<(ostream&, const vec3&);
public:
    vec3();                 // (set entries to 0)
    vec3(double a, double b, double c){ ent[0]=a; ent[1]=b; ent[2]=c; }
    vec3( const vec3& v );  // copy constructor
    // no destructor necessary
    vec3& operator=( const vec3& v );    // assignment
    double& operator[](int i);          // so can say v[i] = 3, etc.
                                        // (be sure to check 0 <= i < 3)
};

ostream& operator<<(ostream& o, const vec3& v);

class mat33
{
    vec3 rows[3];
    friend ostream& operator<<(ostream&, mat33&);
public:
    mat33(){ }              // (entries will already be 0)
    mat33( const mat33& m ); // copy constructor
    // no destructor necessary
    mat33& operator=(const mat33& m); // assignment
    vec3& operator[](int i);         // so m[i] means a row,
                                        //      m[i][j] an entry
    mat33 operator*(const mat33& m); // returns this mat times m
    mat33 operator*(const vec3& v);  // returns this mat times v
    mat33 transpose();               // returns transpose of this mat
};

mat33 identity33();

mat33 rot3(double theta, int i_from, int i_to);

vec3 cartesian(double latitude, double longitude); // (can use prev fns)

ostream& operator<<(ostream& o, const mat33& m);
```

Of course, you will need to supply expansions for functions, either in-line or

separately.

Be sure to include any headers needed.

Technically, in the classes above, whenever a method does not change any class member it is best to tag it with `const`, as in `mat33 transpose() const`;

Here are expansions of the `identity()` and `rot3` functions—but let's use the name `identity33()` instead.

```
mat33 identity33()
{
    mat33 M;                // entries start out 0
    for (int i=0; i<3; i++)
        M[i][i] = 1.0;
    return M;
}

mat33 rot3(double theta, int i_from, int i_to)
{
    mat33 M = identity33();
    M[i_from][i_from] = cos(theta);
    M[i_from][i_to]   = -sin(theta);
    M[i_to][i_to]     = cos(theta);
    M[i_to][i_from]   = sin(theta);
    return M;          // don't forget this line!
}
```

In writing the `rot3` function, to keep things straight just focus on the case where `i_from` is 0 (x) and `i_to` is 1 (y). Notice that the `rot3` function works correctly no matter whether you want `rot3(theta,x,y)` or `rot3(theta,y,x)`.

Assuming you have also included the matrix multiplication routine as a member function of `mat33`, namely

```
mat33 operator*(mat33 m)    or better,
mat33 operator*(const mat33& m), you can now say simply
mat33 R = rot3(pi/2 - view_lat_rad,z,y)*rot3(view_long_rad+pi/2,y,x);
```

#### 8.4. Doing it really right

If you want to be current, you can use the Standard Type Library (STL), which has now been officially adopted as part of C++. There is a vector class `vector` that is handy. To access it, use

```
#include <vector.h>    or
```

`#include <vector>` depending on the compiler.

You can still define `vec3` and `mat33` for convenience, as derived classes:

```
class vec3: public vector<double>
{
public:
    vec3():vector<double>(3){}
};
```

```
class mat33: public vector<vec3>
{
public:
    mat33():vector<vec3>(3){}
};
```

One advantage is that `vector` is already debugged. Another is that you don't need to write copy constructors and assignment operators, since the default ones already work properly on vectors. Also, `[]` is already defined for vectors.

You do still need to write the `*` operator for matrices and for a vector times a matrix. Also, if you want to be able to have debugging output you still need to write an ostream operator for vectors and another one for matrices. In addition, you'll need to write the functions `identity33()`, `rot3`, and `cartesian`