In this assignment, you will write a program for drawing 3D objects with hidden lines removed—that is, you will draw objects as if they were solid, and not transparent. The programming and written assignments are due at 11:59 PM on the due date. As always, your work must be your own.

1 Written Exercises

These proofs will be graded along the same lines as the proofs for assignment 5 (i.e. pass/fail). As before, we will randomly select a few of your proofs to discuss (anonymously) in the recitations.

1. A simple path between vertices $u$ and $v$ in a graph means that no vertex is encountered twice. Use induction to prove that if there exists a non-simple path between a pair of vertices, then there must exist a simple path between those two vertices. Assume the graph is a directed graph. (Hint: Don’t do the induction by making the graph bigger and bigger. Instead, take an arbitrary graph, $G$, and induct on the lengths of paths within $G$. What is the base case?)

2. Let $G = (V, E)$ be a directed, acyclic graph and let $f$ be a function that maps each vertex $u \in V$ to some integer $f(u)$. We say that the function $f$ imposes a topological ordering on $G$ if for every edge $(u, v) \in E$, we have $f(u) < f(v)$.

   **Statement:** Let $f$ be a function that imposes a topological ordering on a graph $G$. If $x$ and $y$ are distinct vertices in $G$, and there is a path from $x$ to $y$, then $f(x) < f(y)$.

   Give an inductive proof of the above statement.

2 Programming: Drawing Solid 3-D Objects

Consider figures 1 and 2. In these figures, we have drawn a torus (a mathematical surface reminiscent of a doughnut) in two different fashions. In one figure, we have drawn the torus as though it were transparent. In the other, we have drawn the torus as though it were opaque. The problem of removing the appropriate lines in this sort of figure is called the “hidden line removal problem.”

In this assignment, you will write a program that removes hidden lines, using a simple technique called the “painter’s algorithm.” The idea is to first draw things which are farther away, then draw closer things on top of them. The objects you draw later may partially or completely obscure the objects you drew earlier. This is the method that some painters use: first, a full background is drawn;
Figure 1: Transparent wire-frame torus

Figure 2: Wire-frame torus with hidden lines removed
then, objects in the foreground are painted over the background, partially obscuring it. On a computer, the problem is figuring out in what order to draw everything.

The objects that you will draw in this assignment will have a very simple form. Objects are made up of a number of polygons, situated in three-dimensional space, denoted $\mathbb{R}^3$. A polygon with $n$ points consists of a number of vertices $p_i = (x_i, y_i, z_i)$ with $1 \leq i \leq n$. You will view polygons from a hypothetical vantage point $V = (V_x, V_y, V_z)$ in $\mathbb{R}^3$. That is, you will pretend that you are situated at the point $V$, and that you are looking directly at the point $(0, 0, 0)$. A piece of paper is placed in front of you, and you mark down where every point in space appears to be on that piece of paper.

For example, consider figure 3. The object we want to draw consists of a cube, whose center lies at $(0, 0, 0)$. The cube consists of six polygons, with each polygon having four vertices. (We have drawn the cube slightly “exploded” to make this clearer, and have only drawn three of the six polygons.) The viewer is located at point $V$. The gray shaded plane is called the viewing plane. We project vertices onto this plane by drawing a line from a vertex to $V$, and marking where this line hits the viewing plane. We label points on the viewing plane by giving them an $x$ and $y$ coordinate, with $x$ running horizontally along the viewing plane, and $y$ running vertically.

Thus, drawing a 3-D polygon on a 2-D piece of paper is fairly simple. For each vertex $p_i \in \mathbb{R}^3$, we find the projection of that point in the two-dimensional image plane ($\mathbb{R}^2$), and draw straight lines between successive vertices of a polygon. We will provide you with a routine that takes a point in $\mathbb{R}^3$ and returns the projection of that point into $\mathbb{R}^2$ i.e. the coordinates of the projection of the point in the image plane.

We will also provide you with routines that draw the edges of a polygon in the plane, and “white out” anything inside. By making sure you draw polygons that are farther away first, you will correctly
One simple idea (which unfortunately doesn’t always work), is to compute the distance between the viewer, and each polygon, and then sort the polygons based on their distance from the user. Unfortunately, it is not clear exactly how to define the distance between the viewer and a given polygon. For instance, we might measure the distance of the viewer to the center of each polygon, or perhaps the distance between the viewer and the first vertex of each polygon, and then sort polygons based on these distances. However, this sort of ordering scheme does not always work—in figure 4, for example, part of polygon $B$ is farther from the viewer than polygon $A$, but other parts of this polygon are nearer the viewer than polygon $A$.

![Figure 4: Two polygons that are hard to draw.](image)

Instead of computing distances and forming a total ordering by sorting, we will establish a partial ordering by examining the relation between all pairs of polygons. For every pair of polygons $P_i$ and $P_j$, you will establish that either

1. $P_i$ is obscured by $P_j$, so $P_i$ must be drawn first
2. $P_j$ is obscured by $P_i$, so $P_j$ must be drawn first
3. neither polygon obscures the other, so either of $P_i$ and $P_j$ may be drawn before the other

You will represent this information as a directed graph. Each node of the graph represents a polygon, and there is an edge directed from node $i$ to node $j$ if $P_i$ should be drawn before $P_j$ (meaning that $P_j$ obscures $P_i$). It is up to you to choose a means of representing the graph. (Either a dense or sparse representation is fine.) You will build the graph by comparing pairs of polygons and inserting edges as appropriate. Once you have formed the graph, you will perform a topological sort to find an order in which to draw the polygons so that polygon $P_i$ is drawn prior to polygon $P_j$ if there is an edge from $P_i$ to $P_j$. 
3  Input/Output and Datastructures

We will provide you with several objects on which to test your program. The objects are stored in text files and have the following simple format:

```
number-of-polygons
polygon
polygon
polygon
...
```

`number-of-polygons` is an integer which indicates how many polygons are in the object. Each `polygon` has the following format:

```
number-of-vertices
vertex
vertex
vertex
...
```

`number-of-vertices` is an integer which indicates how many vertices are in this polygon. Each `vertex` has the following format:

```
x-coordinate  y-coordinate  z-coordinate
```

Each of these coordinates is a double.

The output of your program will be PostScript code. You’ll be able to either view it on the screen or print it out. Don’t worry, you don’t need to know anything about PostScript in order to do this assignment—the routines we provide you will take care of generating the PostScript code.

We have also defined the datatype `polygon`, in the file `graphics.h`. You will use this datastructure to store your polygons.

4  Outline of Program

The following is a broad outline of what your program needs to do. The functions mentioned below are described in the next section.

- Set up the viewpoint by reading its coordinates from the command line and calling `setup_view()`.
• Read the whole object into memory from the input.

• For each vertex of each polygon in the object, use the routine `view_transform()` to translate and rotate that vertex into a new position based on the user’s choice of viewpoint. Store the transformed point in the `pts` array of the polygon. (For a fuller explanation of why we do this, read the comments in `graphics.C`).

• Take each transformed vertex (obtained from `view_transform`), and project it into the viewing plane, using the routine `project()`. Save the value returned by `project()` in the appropriate spot of the `proj` array of the polygon.

• Compute the “bounding box” of the projected image of each polygon. The bounding box of a polygon is the smallest possible rectangle whose sides are parallel with the x and y axes, and that encloses all the projected points of the polygon. The box is stored in terms of two points \((x_{\text{min}}, y_{\text{min}})\) and \((x_{\text{max}}, y_{\text{max}})\) which denote the lower-left and upper-right corners of the box. The value of \(x_{\text{min}}\) for example, is simply the smallest x coordinate of all the projected points of the polygon. Make sure you store the bounding box information for the polygon in the `b` element of the `polygon` datastructure: the routine `compare_poly()` will not function correctly without this information.

• Compute a bounding box `bscene` for the entire scene: that is, find the smallest rectangle that encloses all the bounding boxes computed in the previous step. Call the routine `init GRAPHICS` with `bscene`.

• For all pairs of polygons, call `compare_poly()` to figure out the relationship between the two polygons, and build a directed graph which describes the dependencies between polygons as previously described. (Hint: you should only make about \(n^2/2\) calls to `compare_poly()`, not \(n^2\) calls.)

• Perform a topological sort on your directed graph (see the next section for more information).

• Use the results of the topological sort and draw the polygons in order. To actually draw a polygon, you will use the routines `begin_polygon()`, `polygon_vertex()`, and `end_polygon()`.

• Call `end GRAPHICS`.

5 Topological Sorting

**Warning!** In some cases, the graph you build may have cycles (we tried to supply datafiles that would yield acyclic graphs, but on some rare occasions, cycles may legitimately appear in your graph). You should be careful to code your routines in such a way that a cycle in the graph does not result in an infinite loop.

There are a variety of ways to do topological sorting. The simplest is as follows: beginning at each node of the graph that has not yet been visited, perform a recursive depth-first search. When the
Depth-first search call for a given node returns, add that node to the beginning of an output list \( L \). After the entire graph has been searched, output the list \( L \) from beginning to end. In this manner, nodes that finish early are output later, and vice versa.

While this works well, it has the problem that for very large graph structures, it may result in too many recursive calls (leading to a “stack overflow” because there is often a limit as to how deeply you can recurse). You can try the above if you want, but depending on your machine, it might not work.

An alternative is to avoid recursion, by using an explicit stack. A nonrecursive topological sort will use a \( \text{dfs} \) procedure of the following form:

```c
defs(source) {
    if source is visited
        return;

    stack traverse_stack;
    traverse_stack.push(source);

    while traverse_stack is non-empty {
        let v = node at top of traverse_stack
        if v is unvisited { // v has been recently added
            mark v as visited
            for all neighbors u of v
                push u onto traverse_stack
        }
        else { // v’s neighbors are done being processed
            pop traverse_stack (to remove v)
            if v has not yet been added to L
                add v to beginning of L
        }
    }
}
```

You will have to supply code that implements a stack of pointers to nodes. Your interaction with the stack, in coding the above algorithm, should be solely through the stack’s member functions, in order to have a clean and simple \( \text{dfs} \) function. (Hint: In the above pseudocode, looking through all of \( L \) to see if \( v \) has been added is expensive (in terms of running time). Find a better way.)

### 6 Support Routines

Run `hw6setup` to obtain the supplied files/testfiles. You will need to compile with the following source and header files: `graphics.C`, `compare.C`, `pointmath.C` and `graphics.h`. The file `graphics.h` describes several datastructures used by the routines below; you should read
through graphics.h and make sure you understand the data structures described in that file. You will probably also want to include the standard C++ header files iostream.h and stdlib.h.

Important: You will also need to compile with the math library. On the line in your Makefile where you compile all your .o files together, add -lm. (See the assignment 3 Makefile for an example.) All of the routines listed below are found in graphics.C, except for compare_poly, which is found in compare.C.

- void setup_view(point3 v) sets the viewer as having position v.
- point3 view_transform(point3 p) takes the point p, transforms it according to the viewpoint, and returns the transformed point.
- point2 project(point3 transformed_p) takes a point returned by view_transform, projects it into the viewing plane, and returns the result (which is a 2D point).
- void init_graphics(bounding_box bscene, bool transparent) prepares the graphics routines to begin drawing. The box bscene represents the bounding box that you computed for the entire scene. If transparent is true, then polygons will be drawn in the style of figure 1, otherwise, polygons will be drawn so as to white-out their interior (such as in figure 2). This routine must be called before you begin to draw any polygons. For debugging purposes, it may be useful to call this routine with transparent being true (see below).
- void begin_polygon() signals that you are about to draw a new polygon.
- void polygon_vertex(point2 p) draws the next vertex of the polygon. You will call this once for each vertex of a polygon, passing the projected vertex (stored in the proj array).
- void end_polygon() signals that you are done drawing this polygon.
- void end_graphics() should be called when you are done drawing all the polygons.
- poly_relation compare_poly(polygon *poly1, polygon *poly2) is used to compare two polygons. If the routine returns POLYGON1_FURTHER_AWAY, then you need to add an edge in your graph that indicates that poly1 must be drawn before poly2. If POLYGON2_FURTHER_AWAY is returned, you must add an edge so that poly2 is drawn before poly1. If POLYGON_NO_RELATION is returned, then it doesn’t matter which polygon is drawn first—thus, no edge is added into the graph, on account of these two polygons.

7 Coding hints

7.1 How to read input arguments from the command line

Unlike the programs from your previous assignments, this program will take part of its input—the viewpoint—from the command line. Thus, if you run your program by typing
then this means “read the object stored in the file torus1, draw it from the viewpoint (100,150,200), and store the PostScript output in the file torus1.ps”. From within ddd, this would be run 100 150 200 < torus1 > torus1.ps

A more detailed description of accessing command line can be found in the notes on the assignment Web page—here we give you a quick summary. You can use the following code-fragment to access the command line parameters:

#include <stdlib.h>
int main(int argc, char *argv[]) {
    
    double xval = atof(argv[1]), yval = atof(argv[2]), ...  // e.g. 100 and 150 in the above example
    
}

### 7.2 How to read numbers from the input file

The I/O operators in C++ make reading input easy. Here is an example that reads in an integer, then two doubles:

#include <iostream.h>

int k;
double a, b;
cin >> k >> a >> b;

### 7.3 Breaking up the assignment

You will find it easier to code up the assignment by initially omitting the graph-building, and topological sorting step. That is, simply read the polygons in, transform the data, and then output the polygons in the order you read them. If you do this, you should call init.graphics() with the flag transparent set to true. This will give you pictures like figure 1, as opposed to figure 2. After you have all this working correctly, then you should go back and do the graph-building, and topological sorting step. (Don’t forget to set the flag transparent to be false, though—otherwise, even if you order everything perfectly, you’ll still get pictures that look like figure 1.)
8 Diagnostic Aids

We have supplied a sample solution program `sdrawobject`. To run it, type

```
    sdrawobject vx vy vz < inputfile > outputfile.ps
```

where \((vx, vy, vz)\) are the coordinates of the viewpoint. To help you in debugging, `sdrawobject` will produce a file `draw.log` which will contain information about the bounding boxes, and which polygons are obscured by which other polygons. (The `draw.log` file might become quite large depending on the dataset you test your file on.) Your program should not produce such a file.

To view the output on your screen, type:

```
    gs outputfile.ps
```

To print the output, type:

```
    lpr -P printer-name outputfile.ps
```

Do NOT use `enscript` or any other printing utility to print out these files—if you do, you will only generate many pages of useless output, and waste a lot of paper.

9 Test Cases

We have provided the following input files for you to use in testing your program. Some of the larger files take a significant amount of time to process; you should debug your code using the smaller files.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
<th># Polygons</th>
<th>Interesting Viewpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>torus1</td>
<td>a torus</td>
<td>64</td>
<td>((-50, 40, -30))</td>
</tr>
<tr>
<td>torus2</td>
<td>a more detailed torus</td>
<td>225</td>
<td>((-50, 40, -30))</td>
</tr>
<tr>
<td>torus3</td>
<td>even more detailed</td>
<td>1225</td>
<td>((-50, 40, -30))</td>
</tr>
<tr>
<td>surf1</td>
<td>an interesting mathematical function</td>
<td>400</td>
<td>((-100, 50, -75))</td>
</tr>
<tr>
<td>surf2</td>
<td>more detail of above</td>
<td>1600</td>
<td>((-100, 50, -75))</td>
</tr>
<tr>
<td>surf3</td>
<td>another interesting function</td>
<td>1600</td>
<td>((200, 60, -75))</td>
</tr>
<tr>
<td>hard_to_draw</td>
<td>see for yourself</td>
<td>642</td>
<td>((-140, 70, -100))</td>
</tr>
</tbody>
</table>

10 What You Should Hand In

Create your `Makefile` so that the final executable is named `drawobject`. Structure the code among however many source and header files seems logical. Please do not modify any of the header or source files we have supplied. Hand in all source, header, and `Makefiles` necessary to compile your programs (and yes, that includes the ones we supplied you with). Do not NOT hand in ANY output, dataset, or executable files. We will deduct points if you do so! Hand in the answers to the written exercises in a file named “written.”