Project 2

1 Goal

Practice OpenGL, in particular setting up a view and applying the right transformations to the input triangle mesh model, computing its normal vectors and setting the rendering parameters. Implement interactive interface allowing the user to rotate the model, zoom in or out and change some of the rendering parameters.

This is an **Individual** project. You can discuss high level or technical issues with others, but not the actual code details. All the code you submit must be yours.

2 Things to be implemented

2.1 Input file

You can assume the input file name is '`input.t`'. The first two entries of the input file will be integers, the triangle count t and the vertex count v. Then, there will follow a triangle table: t lines, each listing three integers (ID's of vertices defining a triangle). Finally, the input file will contain the vertex table: coordinates of all v vertices. For example, here is an input file that represents a tetrahedron:

```
4 4
0 1 2
2 1 3
2 3 0
0 3 1
0.000000 0.000000 1.000000
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 0.000000
```

The first two numbers say that you have 4 triangles and 4 vertices (mind the order!), the coordinates of the vertices are listed in the final 4 lines and the other lines tell you which triples of vertices bound a triangle. To read the file, don't try to do anything sophisticated. Here is what will do:

```
class point {
public:
double x,y,z;
};

class triangle {
public:
int a,b,c;
};

.........

ifstream ifs("input.t");

int number_of_triangles, number_of_vertices;
ifs >> number_of_triangles >> number_of_vertices;


triangle *tri_table = new triangle[number_of_triangles];
vertex *vert_table = new vertex[number_of_vertices];


int i;
```

```
for ( i=0; i<number_of_triangles; i++ )
ifs >> tri_table[i].a >> tri_table[i].b >> tri_table[i].c;

for ( i=0; i<number_of_vertices; i++ )
ifs >> vert_table[i].x >> vert_table[i].y >> vert.table[i].z;

.........
```

2.2 Rendering the model

You will need to draw the model you have read in. We are not setting very strict requirements as to how to set the lighting conditions or viewing parameters (see below though); do whatever pleases your eyes, just make sure the image looks clean and you don't do anything crazy like putting the light source inside the model. Also, make sure that in the initial image you draw the model shows up and it fits inside the window, and is large enough to allow one to see what it is. The center of the window should roughly overlap with the center of the object. One way to ensure this is to apply a transformation (combination of translations and scales) to the model so that the center of the *bounding box* of the model projects to the center of the screen and the entire bounding box is visible.

After the program is started, a still image should be displayed. The light should be placed somewhere a little bit above the viewpoint. Also, the back or front face culling should be turned on and culling parameters should be set so that the model appears correctly. One culling parameter setting should work for all models if all triangles' vertices are specified in consistent orders (e.g. same as specified in the input file).

You don't need to build triangle strips or fans, just draw the triangles as 'soup', (`GL_TRIANGLES` mode).

2.3 Interface

2.3.1 Rotating the model: Virtual trackball

The main part of the user interface will be a virtual trackball. Basically, the idea is to give the user the ability to grab (by pressing the left mouse button) an imaginary (invisible) ball centered at the middle of the displayed model and let him/her rotate it by moving the mouse with the left button down. Of course, the displayed model should rotate together with the ball. This requires proper handling of mouse events (see `glutMouseFunc`, `glutMotionFunc`) and, of course, including the user-specified rotation in the modelview matrix.

Make sure that the light source stays in the same place when trackball is used.

2.3.2 Menu

You should be able to select the following items from a menu (just as in the enclosed code, the menu should be attached to the right mouse button):

Flat Switch to flat shading mode

Gouraud Switch to Gouraud shading mode

Phong Switch to Phong shading mode

Specular Make the surface very specular (try to make it so that Phong looks different than Gouraud for most models)

Diffuse Make the surface diffuse

Zoom in Zoom in a little (we want to be able to iterate this until we see just this little triangle in the middle!)

Zoom out Zoom out a little (also here, we need to be able to iterate this)

For zooming in and out, change the field of view in the projection transformation matrix (that's how you zoom in when you take a photograph!). Don't move the object closer or farther.

3 Normals for the triangle mesh

Compute the vertex normals by averaging the normals of all incident faces. More precisely, first compute the triangle's normal as the cross product of vectors running along the edges. Then, add all of the normals of incident faces to get the normal at a vertex. This will yield the area-weighted average normal. Please implement this procedure efficiently though: make sure that your implementation is linear time (i.e. it does not have nested loops with bounds growing with the size of the mesh).

An example of a BAD (quadratic running time) implementation is as follows: for each vertex v, search for incident triangles by looping over all triangles and finding ones that have v as a vertex. Add the normals of these triangles to get the averaged normal at v. It's quadratic since it has two nested loops (one over all vertices, one over all triangles).

An example of a GOOD (linear time) implementation is: initialize a variable $n[v]$ to 0 for each vertex v. Loop over all triangles. For each triangle with vertices v_0, v_1 and v_2 add $\vec{v_0v_1} \times \vec{v_0v_2}$ to $n[v_0]$, $n[v_1]$ and $n[v_2]$. Note there is just one loop here (over all triangles), with constant amount of work done per iteration. Thus it's a linear time implementation. This will make it MUCH faster for large meshes.

4 OpenGL related commands

A natural way to implement the project is to create all OpenGL objects (here: programs, buffers and vertex arrays) at startup and then use them in the drawing callback.

4.1 Types

OpenGL provides type names that make it easier to port code to different platforms. It is best to use these types, especially for variables used for CPU-GPU communication. For the projects, please limit yourself to using `GLfloat` or `float` for floating point data - even though OpenGL supports doubles, there are some restrictions on what functions can take them as arguments. Besides, single precision floats typically provide more than enough accuracy for computer graphics.

If you need integer data, use `GLint` or `GLuint`. **Index buffers, used for indexing vertex arrays, have to use unsigned entries**, such as `GLuint` (which would be `unsigned int` on any system you will likely encounter these days).

The `glm` library is based on templates, so to control the return types of functions you have to make sure that the arguments are of the right type (i.e: `float`). Use `f` suffix for all floating point constants you use (to make sure they are treated as single precision) of explicitly cast any arguments to `float` or `GLfloat` when using glm functions (see the sample code).

4.2 Setting up/dealing with shader programs

Place your shaders in separate files. Then, use the provided function

```
Program * createProgram ( const char *vshFile, const char *fshFile ).
```

The arguments are names of files containing the vertex and fragment shader (in this order). **Use relative paths for the shader file names - otherwise we'll run into problems running your code.** The function compiles and links the shaders and prints out any error messages into the terminal (be sure to look at them to see if everything went fine!).

It is not that important to know precisely what Program class looks like (but feel free to look at it!). Think about it as a representation for a compiled program that resides in the GPU memory. To make things simpler, the Program class has no typical copy semantics: try to call a copy constructor or assignment operator for Program object and you'll get an error. For best results, **use only pointers to Program objects in your code. The same applies to other classes that represent OpenGL objects: Buffer, VertexArray and IndexArray.** Apart from simplicity, the reason for setting things up this way is that typically copying OpenGL objects is useless anyway, since there is only a limited access to them.

Here are methods of the `Program` class that should suffice for completing the project:

- The `on` method turns the program on (i.e. tells the system to use that program to process any vertex stream sent to the GPU) and turns all other programs off.
- The `off` methods sets the current program to an invalid value; note that an invalid program will not generate any errors, but may lead to odd looking results.
- To set uniform variable values, use the method `setUniform`. There are a lot of overloaded implementations of this method provided. The arguments are the name of the uniform variable (`const char *` type) followed by the values to send to it or a pointer to an array containing these values. The function should be able to figure out how many values need to be sent. It also works as a way to send matrices to uniform variables (see sample code). **If the variable name is not found, an error message is printed into terminal but the code will not terminate.** The reason for setting it up this way is to make debugging easier. Sometimes, you want to debug shaders by simplifying the calculations (e.g. using a single color instead of illumination formula to see if things show up in the right places). This may cause some uniforms to disappear from the compiled code (since the compiler removes all whose values do not affect the output). Even if this happens, you'll be able to run your code with no changes.

4.3 Buffers

Buffers are basically arrays residing in the GPU memory. Construct any buffer you need using a constructor of the `Buffer` class. For reasons explained earlier on, use only pointers to objects of the *Buffer* class in your code. The three arguments to use with the constructor are

- Number of *components*, or scalars per entry; typically the entry is a vertex of some 3D model (e.g. 3 for 3D points or normals, 4 for points in homogenous coordinates, 2 for 2D vertex locations, 1 for a scalar value per vertex)
- Number of entries (vertices)
- Pointer to data to be transferred to the buffer; the functions we provide support `GLfloat*`, `GLuint*`, `GLint*`, `GLshort*`, `GLushort*`, `GLbyte*` and `GLubyte*`. If the number of components is c and the number of entries is e, then that pointer should point to an array of size at least $c * n$ of one of these types.

4.4 Index buffers

Index buffers are used to store indices into 'regular' buffers. Under the hood, there is in fact no difference between index buffers and buffers, but they are used in different ways. **Only unsigned types `GLuint`, `GLushort` and `GLubyte` can be used to store index information.** In the project, use `GLuint` since some of the models have more than 65, 536 vertices.

To create index buffer pointer, do

`IndexBuffer *myIndexBuffer = new IndexBuffer ( size, ptr ); ,`

where size is the number of indices in the buffer and ptr is a pointer to the data you want to send into the index buffer. The pointer should point to an array of dimension at least `size`.

4.5 Vertex arrays

Vertex arrays (VAs) are collections of buffers used to draw vertex properties from. For them we provided the `VertexArray` class. To build vertex array, first create a pointer to a 'blank' VA using the argument-less constructor, like this:

```
VertexArray *myVA = new VertexArray;
```

Then, attach attribute buffers to the VA using the method `attachAttribute` of the `VertexArray` class.

```
void VertexArray::attachAttribute ( const GLuint aix, const Buffer * b );
```

This associates attributes stored in a buffer b with an attribute index `aix`. Of course, in order for everything to work as expected, you need to make sure attribute indices are unique (different for different attributes). In most cases, you want the buffers attached to the VA to contain data for the same number of vertices. You also need to provide the matching `location=aix` identifiers for the vertex shader's input variables.

To send vertices defined by the array into the pipeline you have two options. Make sure a program is turned on before sending vertices into the pipeline.

The method

```
void VertexArray::sendToPipeline ( GLenum ptype, int first, int num );
```

sends `num` vertices starting vertex of index `first`. In other words, consecutive indices `first`...`first+num-1` are used to look up attributes from buffers attached to the VA and create vertices. Make sure you don't exceed the bounds for any of the buffers attached to the VA. The first argument instructs the primitive setup stage how to form primitives out of the vertices. Use `GL_TRIANGLES` for triangle soup.

The alternative is to use the method

```
void VertexArray::sendToPipelineIndexed ( GLenum ptype, IndexBuffer *b, int first, int num );
```

Here, you provide an index buffer as the second argument. Indices into the buffers attached to the VAs are looked up from the index buffer, using indices `first`...`first+num-1`. In other words, data for the `i`-th vertex (starting with `i=0`) is extracted using index `b[first+i]`, where `b[]` accesses entries of buffer `b` as if it were an array (you can't really do it in code since b lives in GPU memory).

The sample code provides an example for both ways to send vertices out.

4.6 `glm` library

Since all OpenGL transformation calls are deprecated, we'll be using `glm` as a simple replacement. `glm` can also be used for vector operations (you can even use it to do 3D vector calculations if you'd like to). Everything in the glm library lives in the `glm` namespace, so either insert `using namespace glm` (this is what the sample code does) or precede the names with `glm::`.

In the project, `glm` will be particularly useful when you deal with transformation matrices. For example, you can do this to compute the superposition of two rotations and a translation:

```
mat4 M = translate(mat4(),vec3(0.0f,0.0f,-20.0f))*
glm::rotate(mat4(),float(angle1),vec3(1.0f,2.0f,3.0f))*
rotate(mat4(),float(angle2),vec3(-2.0f,-1.0f,0.0f));
```

A nice feature of the matrix types provided by glm is that they are GLSL-friendly. To send a matrix to a uniform variable `Msh` in a program `p`, you can use pointer to the top-left entry (here: `&M[0][0]`) as the pointer to the vector containing the matrix entries. This is the code:

```
p->setUniform("Msh",&M[0][0]);
```

A simple way to build a perspective projection matrix is to use `glm::perspective` function. In the sample code, we have

```
mat4 PMat = perspective(8.0f,1.0f,15.0f,25.0f);
```

which constructs the perspective projection matrix with field of view 8 **degrees**, aspect ratio 1 (suitable for a square window) and with front and back clipping planes 15 and 25 units away from the viewpoint, the origin. The camera is directed along the *negative* z-axis.

Here is a quick `glm` cheat sheet that should suffice for this project.

`mat4` is a type for 4×4 matrices. There are several versions of constructors available for it, but all you'll need is the default one that initializes the matrix to the identity.

To create the 4×4 matrix of translation by a vector `[Tx,Ty,Tz]` use

```
translate(mat4(),vec3(float(Tx),float(Ty),float(Tz))).
```

To create the 4×4 matrix of the rotation by angle a around an axis going out of the origin and in the direction `A=[Ax,Ay,Az]`, use

```
rotate(mat4(),float(a),vec3(float(Ax),float(Ay),float(Az))).
```

The rotation angle a is expressed in degrees. Trigonometric functions in `cmath/math.h` use radians, so you will likely need to convert between radians and degrees in your code. Also, don't use `rotate` with `A=[0,0,0]`.

To create the 4×4 matrix of the uniform scale by `c`, use

```
scale(mat4(),vec3(float(c)).
```

The constructor for `vec3` with one argument uses the same value as all coordinates of the vector, i.e. here makes the vector `[c,c,c]`.

Note that the first argument of all the transformation calls is a matrix. This is for historical reasons that are not worth going into. Just send identity (in the examples above, the default constructor is used to create it) as the first argument and you'll get the expected transformation back.

To convert a 4×4 matrix `M` into a 3×3 matrix (by dropping the last row and column), use `mat3(M)`. You may find this handy is you decide to use 3×3 matrix to as the normal transformation.

5 Useful GLSL functions

GLSL code is very similar to the C code (see the sample code). However, it provides a number of graphics-oriented functions that you will find useful for the project. Here is a list.

`float dot ( vec3 a, vec3 b )` computes the dot product of the arguments.

`vec3 cross ( vec3 a, vec3 b )` computes cross product.

Scalar by vector multiplication, vector addition, subtraction and negation are available and work like overloaded operators. For example, if you have two variables `w` and `v` of type `vec3`, you can do `-w+0.8*v`.

`vec3 normalize ( vec3 a )` normalizes vector a.

To convert a 3D point `p` of type `vec3` into homogenous coordinates, use `vec4` constructor: `vec4(p,1.0)`. You can use constructors to create vectors of different dimensions too. For example, `vec3(0.0,1.0,2.0)`.

Matrix by matrix multiplication and matrix by vector multiplication can be done by using overloaded `*` operators. For example, if you have `M` of type `mat4`, i.e. 4×4 matrix, and a variable `v` of type `vec4`, you can do `M*v` to multiply them. There are types for matrices of other dimensions (up to 4) available too – you may find `mat3` useful for storing normal transformation (equivalently, the rotational part of the modelview matrix).

To access components of a vector you can use xyzw, rgba or stpq as names of coordinates. For example, if `p` is of type `vec4`, `p.xyz` or `p.rgb` is a 3D vector of type `vec3` obtained from `p` by dropping the 4th coordinate. If only one coordinate name is used, the result is a `float` (e.g. `p.x` is the x-coordinate of `p`).

6 Grading

We'll run your code for some of the posted examples. Breakdown of the grade:

1. flat/Gouroud/Phong modes: 35 (includes normal vector calculations; penalty of 20 for quadratic implementation)

2. zooming in/out: 10

3. Trackball: 35

4. Specular/diffuse materials: 20

The grading will be done by just running your code and trying things out. Make sure that the time between the code is started and the model shows up in your window is only about a few seconds and there are no delays when switching between modes. As always, we need to be able to *see* that something works to give credit for it.

When grading the trackball, we'll check if it gives reasonable immediate feedback (meaning reasonable refreshing of the displayed image in response to the mouse-moving-with-button-down events) and if it is reasonably intuitive to use. Most importantly, the rotation direction/axis/angle should be naturally linked to the mouse cursor movement. To make things simpler, we promise we will not resize the window when testing your code.