Computer Graphics Spring 2021

TP 1

Submission date:

Problem. The main goal of this assignment is for you to become familiar with basic `OpenGL`. You will have to do the following things:

1. Learn how the basic `OpenGL, GDL2, GLM` and `Shaders` setup works. Download the starting `OpenGL` code I have given you, understand and compile it.
2. We will use the library `GLM` for points, vectors and matrices classes. See how `GLM` is used in the code I have given you.
3. Learn how to draw using old-fashioned way in `OpenGL`. You will write functions to read a `.obj` mesh file into a simple float data array, the indices into an index array. Then draw the mesh using `glBegin/glEnd`.
4. Then write a function to compute the normals of each vertex (as discussed in class), and using `glBegin/glEnd`, draw the normals of the model.
5. Write the code to implement the arrow keyboard keys.
6. Draw using vertex buffer and vertex array objects.

I have already created empty functions where you will have to write the code. All code will be written either in `main.cpp`, `myobject3d.h/cpp` or `myCamera.h/cpp` files. Places where you have to write code are indicated by text `ADD CODE`.

1. Running OpenGL

Download `startingcode.zip` from the course webpage. This contains the `Visual Studio 2017` project for a basic OpenGL example. Compile and run it. We covered all the code present here in class, but you should go over the entire code, and make sure you understand it. Here are the source files in the starting code:

- `main.cpp`. It contains the windowing code using the library `SDL2`, functions to handle mouse and keyboard events, as well as the main event loop that draws your scene.
- `myObject3D.h, myObject3D.cpp`. This contains the object class, and is where you will write most of your functions and OpenGL code.
- `myCamera.h, myCamera.cpp`. This contains the camera class. Part of the code you will write in this TP will be in this class.
- `helperFunctions.h`. This will contain useful stand-alone functions that you might need for various this. For the moment, it contains the function `rotate` that can rotate an input point in 3D around an axis through the origin at an input angle.
- `myShader.h, myShader.cpp`. Contains the code to read and load fragment and vertex shaders. <u>You will not need to modify this file in TP 1</u>.

2. Reading a Mesh

There is a mostly empty class `myObject3D` meant to store data and functions for 3D mesh objects. Currently the `myObject3D` class contains the following members:

```
class myObject3D
{
public:
    std::vector<glm::vec3> vertices;
    std::vector<glm::ivec3> indices;
    std::vector<glm::vec3> normals;

    glm::mat4 model_matrix;

};
```

You will have to add the following function to this class to be able to read a mesh:

- `bool readMesh(string filename)`. Read a mesh from a `.obj` file into vertices and indices array. Returns `true` if the mesh was read properly.

  The vertices go into `std::vector<glm::vec3> vertices`. So the variable `vertices[i]` contains the `i`-th vertex of the mesh.

  The face indices go into `std::vector<glm::ivec3> indices`. So the variable `indices[j]` contains the `j`-th triangle of the mesh.

  For simplicity, we will make each face of the mesh a triangle. If a face is not a triangle, then you should break that face into triangles in the `readMesh()` function before storing it in the array `indices`.

  Don't forget that the indices in a `.obj` file start from 1, and so you will have to subtract 1 from them before you add them to the array `indices`.

The class `myObject3D` **already contains** a drawing function, `displayObject()`, that can be called to draw the mesh you read:

```
void myObject3D::displayObject(myShader *shader)
{
        shader->setUniform("mymodel_matrix", model_matrix);
        shader->setUniform("input_color", glm::vec4(1, 1, 0, 0));

        glBegin(GL_TRIANGLES);
        for (unsigned int i = 0; i < indices.size(); ++i)
        {
                glVertex3fv(&vertices[indices[i][0]][0]);
                glVertex3fv(&vertices[indices[i][1]][0]);
                glVertex3fv(&vertices[indices[i][2]][0]);
        }
        glEnd();
}
```

If you press 'o' in the window, an open file dialog will appear, so you can try different meshes without re-starting your program.

3. Computing and Drawing Mesh Normals

Once you have read and stored the mesh into an object of class `myObject3D`, the next step to compute the normals and draw them.

You have to write the following functions:

- `void computeNormals()`. This function must compute the normals of each vertex in the mesh. Store these normals in the array `std::vector<glm::vec3> normals` in the class `myObject3D`, in the same way the vertex data is stored.

  Remember that to compute the normal of a vertex `vertices[i]`, you have to compute the normal of each face adjacent to `vertices[i]`, and then take the average of these normals. The following code computes the normal vector of a face `indices[j]`:

  ```
  glm::vec3 face_normal = glm::cross( vertices[indices[j][1]] - vertices[indices[j][0]],
                                      vertices[indices[j][2]] - vertices[indices[j][1]] );
  ```

- `displayNormals()`. Write a function in the `myObject3D` class to draw these normals on the object. Here is one way to draw them:

```
void myObject3D::displayNormals(myShader *shader)
{
        shader->updateUniform("mymodel_matrix", model_matrix);
        shader->updateUniform("input_color", glm::vec4(0, 0, 1, 0));

        glBegin(GL_LINES);
        for (unsigned int i = 0; i < vertices.size(); i++)
        {
                glm::vec3 v = vertices[i] + normals[i] / 10.0f;
                glVertex3fv(&vertices[i][0]);
                glVertex3fv(&v[0]);
        }
        glEnd();
}
```

3. Keyboard Interface

The code for mouse movement is already given to you. You have to add in the functions for keyboard movement:

- **Keyboard.** Write in functions that change the camera position so that the keys `forward, back, left, right` work. The `forward` key should move the viewer forward, `back` key should move the viewer backwards, `left` key rotates the viewer to the left and the `right` key rotates the viewer to the right. This code has to be written in `myCamera.cpp`, in the following place:

```
void moveForward(float size)
{
        //TODO: write code here to move forward.
}

void moveBack(float size)
{
        //TODO: write code here to move back.
}

void turnLeft(float size)
{
        //TODO: write code here to turn left.
}

void turnRight(float size)
{
        //TODO: write code here to turn right.
}
```

Remember that you only need to change the variables in the **myCamera** class. You should use the function `rotate` that is present in the file `helperFunctions.h`.

Drawing using Vertex Array and Vertex Buffer Objects

I have given you the four files—`vbo.h, vbo.cpp, vao.h, vao.cpp`—on the webpage. Download them, and add them to your project by 'dragging' them into the visual studio solution (the `.h` files go under the **Header Files** group, while the `.cpp` files go under the **Source Files** group).

1. Adding a VBO class to your project.

Remember that a `vertex buffer object`—or VBO for short—stores your data on the GPU. When you open a buffer, `OpenGL` gives you a `buffer id` that can then be used to access that buffer later.

Here is one example of a `VBO` class:

vbo.h

```
class VBO
{
public:
        VBO(GLenum);
        ~VBO();

        void bind() const;
        void unbind() const;
        void setData(GLvoid *data, int size);

        GLuint buffer_id;
        GLenum buffer_type;
};
```

Go carefully through the given `vbo.cpp` file to see what these functions are doing. The functions `bind()` and `unbind()` activate/deactivate the buffers. The key function is `setData`, which uploads the data to the vertex buffer object.

2. Adding a VAO class to your project.

Once you have a VBO set up, you can use a Vertex Array Object—VAO for short—for drawing. Here is how your VAO class can look like:

vao.h

```
class VAO
{
private:
        enum Attribute { POSITION, NORMAL, TANGENT, TEXTURECOORD };

public:
        GLuint id;

        std::map<Attribute, VBO *> attribute_buffers;
        VBO *indices_buffer;
        int num_triangles;

        VAO();
        ~VAO();

        void clear();

        void storeAttribute(Attribute c, int num_dimensions, GLvoid *data,
                            int size_in_bytes, GLuint shader_location);

        void storeIndices(std::vector<glm::ivec3>);
        void storePositions(std::vector<glm::vec3>, int shader_location);
        void storeNormals(std::vector<glm::vec3>, int shader_location);

        void draw();
        void draw(int start, int end);

        void bind();
        void unbind();
};
```

Go carefully through the given vao.cpp file to see what these functions are doing.

3. Drawing with VBOs and VAOs.

Now include, in your `myObject3D` class, a variable of type

`VAO *vao;`

Don't forget to add `#include "VAO.h"` in your `myObject3D.h` file!

In the `main.cpp` file, after you have read the mesh and computed normals, to initialize and create buffers for your `vertices` and `normals`, you should call the function

`obj1->createObjectBuffers()`

The above function should contain the following code:

```
vao = new VAO();

vao->storePositions(vertices, 0);

vao->storeIndices(indices);
```

In the above code, the function `vao->storePositions(vertices, 0)` stores the vertices in a `VBO` and links it with `location = 0` in the vertex shader.

Now the `display` function in the class `myObject3D` becomes:

```
void myObject3D::displayObject(myShader *shader)
{
        shader->setUniform("mymodel_matrix", model_matrix);
        shader->setUniform("input_color", glm::vec4(1, 1, 0, 0));

        vao->draw();
}
```

4. Sending normals to the shaders

Once your normals are computed correctly (displaying them will make it easier to verify this), you need to pass these to the shaders:

1. Create and store the normal data in object buffers. You can do this in a similar way to how we stored the vertex position data in the object buffers. The following function will upload the normals of your object:

   ```
   vao->storeNormals(normals, 1);
   ```

   This function uploads the normals to a `VBO` and links it with `location = 1` in the vertex shader.

2. Add a normal variable (of data type `glm::vec3`) in the vertex shader to receive these normals at the same location that was specified above in **Step 1.**

3. You will need to pass the normal matrix to the shaders to transform the normals (just like the view matrix transformed the vertices). The normal matrix can be computed with the formula `glm::transpose(glm::inverse(glm::mat3(view_matrix)))`.

   You will need to, in `main.cpp`, compute the normal matrix, and pass it to the shaders via uniform variables (in a similar way as was done for the view matrix).

4. Pass the normal vector and the vertex position from the vertex shader to the fragment shader using shared variables. Assigning them values in the `main()` function of the vertex shader, and then you can pass them to the fragment shader using:

   ```
   Vertex shader:         out vec3 mynormal;
                          out vec4 myvertex;
   ```

   You can receive these variables in the fragment shader using:

   ```
   Fragment shader:       in vec3 mynormal;
                          in vec4 myvertex;
   ```

5. There is no need to pass uniform variables from vertex shader to fragment shader. They can be accessed directly in the fragment shader in the same way they were accessed in the vertex shader:

   ```
   uniform mat4 myview_matrix;
   uniform mat3 mynormal_matrix;
   ```

5. Silhouette computation in the shaders

Write code to draw a silhouette of the mesh on the screen. Do not change the main or vertex-shader. You should only make changes to the fragment shader to implement this feature. The key 's' should toggle the silhouette.

1. Add `uniform` variables to the fragment shader for the view and normal matrices.
2. Pass the vertex and normal positions from the vertex shader to the fragment shader.
3. Apply the view matrix to the vertex position, and the normal matrix to the normal in the fragment shader to get the final correct value of the vertex and normal coordinates.
4. Remember that to get the vertex position in 3 dimensions, you have to divide by the homogenous coordinate.
5. Recall that `myvertex` lies on the silhouette if the absolute value of the `dot product` of the normal at `myvertex` with the direction from the camera to `myvertex` is small, say less than `0.2f`.