Project 2 Part II

1.1 Overview

You should design basic data structures and classes for computer graphics.

- Part I: linear algebra (quaternions and transformations)
- Part II: integration of part I into an OpenGL framework

1.2 Required Functionality: Part II

Look at the OpenGL framework, compile, and run. For Windows, the project should work out of the box. For Mac, you may have to adjust the header paths and you will have to add the relevant libraries to the linker separately. See the TA if you encounter problems on a Mac.

1.3 Basic OpenGL

- Familiarize yourself with the framework. In particular, try to understand GLUT's callback mechanism. A full documentation of GLUT can be found here: https://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf

- Use **`g_State.sys_timer.Query()`** to get the time in seconds since the program started. Use this time to rotate the cube by 36 degrees per second around its Y axis. Use **`glRotatef`** for this purpose.

- Now tilt the cube by 30 degrees towards the camera. You should be able to see the top of the smoothly spinning cube.

1.4 Use own matrix classes

- Replace any existing calls to **`glRotatef`**, **`glTranslatef`**, **`gluPerspective`**, etc. by your matrix code as indicated in the framework. Use only **`glLoadMatrixf`** or **`glLoadMatrixd`** to upload your matrices to OpenGL and only where needed. The way you have implemented matrix functions in part I allows you to write something like
  `matf M = matf::Translate(…)*matf::Rotate(…) …`
  That is, you do not need an actual instance (matrix object) to access these functions. Be prepared to explain to the TA why.

1.5 Arcball

- Familiarize yourself with the concept of quaternions. In particular, be prepared to explain the relation between quaternions and 3D rotations as well as the relation between quaternions on the one hand and dotproduct and crossproduct on the other hand. An introduction to quaternions can be found here: https://en.wikipedia.org/wiki/Quaternion

- Use your implementation of Quaternions from part I to implement the **`arcball<T>`** class in **arcball.h** for navigating the object. Then, replace the matrices from 1.4 in this assignment to rotate the cube using the arcball implementation. Hook up your arcball implementation to GLUT's callback function, in particular:
    - Right mouse button down triggers **`arcball::click()`**
    - Moving the mouse while holding the right button calls **`arcball::drag()`**
    - Releasing the right mouse button calls **`arcball::release()`**

  Also make sure to call **`arcball<T>::update_window()`** in GLUT's **`reshape()`** function.

Hints: The arcball is a virtual trackball on the screen. It is an invisible sphere placed around either the object or the screen center. In this exercise, we will place the sphere behind the screen center. If the user clicks, the xy click position is converted to normalized device coodinates (Fig. 1). Mind that GLUT also reports positions *outside* the current view, these have to be clipped. If xy is inside the virtual unit sphere centered at the origin of the normalized device coordinates, an intersection with the unit sphere is computed. If it is outside, xy is projected to the closest point on the sphere. Note that the result of this intersection is a 3D coordinate xyz. Given a click, drag pair of positions, convert the two 3D intersections to quaternions p (click) and q (drag). Using only quaternion math, compute a rotation that goes from q to p. Return this orientation in **`arcball<T>::get_orientation()`**.

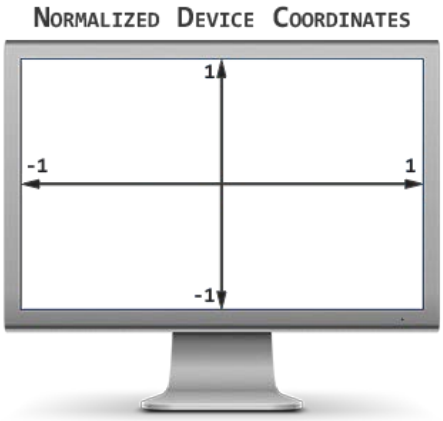

Figure 1. Normalized Device Coordinates.

1.6 Camera

- Implement the missing parts of the **`camera<T>`** class in **camera.h**. The camera will be an ego-camera centered around the user. It will use **`mat<T>::LookAt`** (see part I). Use **`vec<T>`** whenever appropriate.

- Hook up your camera class to GLUT's callbacks. In particular:
    - When the left mouse button is pressed, store the click position

- When the mouse is moved, update the camera's "lookat" part – use **`camera<T>::left_right_rotate()`** and **`camera<T>::up_down_rotate()`** for this.
- When directional keys are pressed (**`GLUT_KEY_{LEFT | RIGHT | UP | DOWN}`**, call **`camera<T>::left_right_move()`** and **`camera<T>::fwd_back_move()`** to update the eye position.

Finally, integrate your camera implementation into the rendering loop, this will replace any previous view matrix in **main.cpp**'s **`displayfunc()`**.

1.7 3D Models

- Familiarize yourself with the concept of OpenGL buffers. See **`generate_cube()`** in main.cpp as an example. Using the model class, load a different model by filling in missing code in **main.cpp**.

- Add a GLUT Popup menu and a submenu called "Model". In the "Model" submenu, add five items called "Cube", "Sphere", "Torus", "Cylinder", and "Boundary". In the callback of this submenu, load the corresponding obj file and upload the data to OpenGL for rendering.
  - Use **`void main_menu(int item)`** as the callback for the main menu (**main.cpp**)
  - Use void **`model_menu(int item)`** as the callback for the model menu (**main.cpp**)
  - Attach the menu to the middle mouse button (**`GLUT_MIDDLE_BUTTON`**)
  - Implement the callback **`model_menu`** in **main.cpp**

Hints: The model class will load an obj file and split up "nodes" into simple vertices. A "node" is a vertex with a unique position that has a different normal, color, or texture coordinate for some of the adjacent faces. OpenGL does not support such nodes. **`model::get_vertex_data()`** will then return a pointer to the vertex data as a simple **`float`** array, while **`model::nVertices()`** will return the number of vertices in the model. In the same fashion, **`model::get_face_data()`** will return a pointer to an **`int`** array, and **`model::nFaces()`** will return the number of triangles. This data is essentially in a shared index format, where faces store indices to vertices. OpenGL can render such data natively, but you need to "upload" the data to the GPU first. For this, inspect the code in main.cpp's **`generate_cube()`** and implement the missing part in main.cpp's **`load_model()`**. This will create one buffer per vertex attribute (position and color for now) and one additional buffer for face indices. The information about these buffers is then stored in a **`GL_VERTEX_ARRAY`** object (**`g_State.hVAO`**). Putting data in large coherent blocks on the GPU will result in good performance, as opposed to the now deprecated immediate mode using **`glBegin()`** / **`glEnd()`**. Therefore, do not use **`glBegin()`** / **`glEnd()`** in your code.

List of Files

Your project should have the following files:

main.cpp	main file, provided, **needs implementation** (part II)
model.cpp	obj file loader, complete
Program.cpp	wrapper for OpenGL shaders, complete
arcball.h	arcball file, **needs implementation**
camera.h	camera file, **needs implementation**
glu.h	GL utilities, complete
mat.h	your matrix file, **needs additional implementation** (part I)
model.h	obj file loader, header file, complete
Program.h	wrapper for OpenGL shaders, complete
quat.h	quaternions file, **needs implementation** (part I)
timing.h	timer class, complete
vec.h	your vector file, should be complete by now
Program.inl	wrapper for OpenGL shaders, complete
matrix_extensions.txt	prototypes for new matrix functions
fsColor.txt	shader file, complete
vsTransform.txt	shader file, complete

In addition, you need GL.h, glut.h, etc. For Windows, they are in the ./GL folder. For Mac, they are likely to be on your system, but you need to alter the paths to include them: https://developer.apple.com/library/mac/qa/qa1613/_index.html

Also, you need the GLEW (GL Extension Wrangler). The project contains a version for Windows, but it can also be downloaded here: http://glew.sourceforge.net/

On Windows, you will need the glut32.dll and the glew32.dll. For both, you need to link to a library. To do that, right click on the project, select Properties, and then Linker to add locations and libraries as necessary. If not done properly, the compiler will complain about missing symbols.

Finally, the exercise also comes with another copy of the obj files used in the previous exercise for your convenience.