Assignment 2

CS4533 Fall 2020

Due: Wed. 11/4 11:55pm. Total Score: 145 points

Note: This assignment has 3 pages.

This assignment is to implement a simple animation by rolling a sphere which is modeled by a *wireframe* representation, with some user-interface capabilities. You will practice transformations and viewings, as well as programming with shader-based OpenGL using vertex buffer objects. You should study the sample code "Handout: rotate-cube-new.cpp" available at `http://cse.poly.edu/cs653/Rotate-Cube-New.tar.gz` and discussed in class, to see how you can spin a cube and support various keyboard functions. Use the code as a starting point for this assignment; in particular, the helper files "*.h" and "InitShader.cpp" are ready to be used as they are. Also consult "Handout: GLUT Mouse and Menus" available at `http://cse.poly.edu/cs653/assg2/Handout-GLUT-Mouse-Menus.pdf` for the information on how to set up mouse functions and pop-up menus.

Note:

1. The old approach of immediate-mode rendering using `glBegin()` and `glEnd()` to draw objects is **no longer allowed**. Instead, you should use **vertex buffer objects** (see "Handout: rotate-cube-new.cpp") to draw all objects. Also, object colors should **not** be set using `glColor()`; set them as vertex attributes via vertex buffer objects or as uniform variables in the vertex shader (when all vertices have the same color).
2. Sphere-drawing library functions such as gluSphere(), glutWireSphere(), glutSolidSphere(), etc., are **not allowed** — they use deprecated OpenGL functions `glVertex()` and `glNormal()` inside `glBegin()` ... `glEnd()`, i.e., they use the old immediate-mode rendering, rather than the desirable vertex buffer objects.
3. Assignments 3 and 4 will build on this assignment to eventually complete a course project.

General Instructions: same as Assignment 1.

(a) There are two sphere files in `http://cse.poly.edu/cs653/assg2/`. These files contain tessellated **unit**-sphere (i.e., the radius is 1) representations with 8 and 128 triangles, respectively. The tessellated spheres are **centered at the origin**. The input file format represents a sequence of n-vertex polygons where each polygon is represented as:

$$\begin{array}{ccc} n & & \\ x_1 & y_1 & z_1 \\ & \vdots & \\ x_n & y_n & z_n \end{array}$$

Here each polygon is a triangle, so n is always 3. At the very beginning of the file, there is also an integer indicating the total number of triangles in the file (8 and 128, respectively).

Write a function that prompts the user for the name of the input file, and reads in that file (of the above format). Store the data into some data structure where each triangle is stored once. **(5 points)**

(b) Set up the viewing such that
(1) $VRP = (7, 3, -10, 1)$, $VPN = (-7, -3, 10, 0)$, and $VUP = (0, 1, 0, 0)$,
(2) the VRP maps to the center of the (graphics) window,
(3) the window is square, and
(4) use **perspective** projection.

Under this viewing setting, draw a quadrilateral in color green (i.e., $(0, 1, 0)$) with vertices at $(5, 0, 8, 1)$, $(5, 0, -4, 1)$, $(-5, 0, -4, 1)$, and $(-5, 0, 8, 1)$ to indicate the x-z plane, and also draw the x-, y-, and z-axes in colors red (i.e., $(1, 0, 0)$), magenta (i.e., $(1, 0, 1)$) and blue (i.e, $(0, 0, 1)$), respectively. Translate the center of the sphere to the point $A = (-4, 1, 4, 1)$ and draw the *wire-frame* sphere (by setting a suitable mode in `glPolygonMode()`). Draw the sphere (lines) in golden yellow $(1.0, 0.84, 0)$ and set the background color to sky blue $(0.529, 0.807, 0.92, 0.0)$. Use your data structure in **Part (a)** to draw each wireframe triangle once. **(20 points)**

(c) This part is to *roll* the sphere on the x-z plane by using translations and rotations. First, roll the sphere so that its center goes from the point $A = (-4, 1, 4, 1)$ to the point $B = (3, 1, -4, 1)$ along a straight line. Note that the rotation and the translation should match to produce the effect of rolling the sphere on the x-z plane. As soon as the center passes through B, roll the sphere so that its center goes from the point B to the point $C = (-3, 1, -3, 1)$ along a straight line. Finally, roll the sphere so that its center goes from C back to A along a straight line, and then repeat rolling the sphere so that its center loops through the points A, B and C forever, until the user exits the program (e.g., by selecting "Quit" in the menu; see **Part (e)** below). Note that the sphere rolling should **always keep the same speed**, even among different rolling directions (A to B, B to C, and C to A). Also, you should adjust your viewing setting to make the sphere always entirely visible and also reasonably large.
(Note that here you can just use the initial solution as described in **Part (d)** below, and address the *rotational discontinuity* issue in **Part (d)**; see **Part (d)**.) **(70 points)**

Useful Tips:
1. This "sphere-rolling" part is a good example of *instance transformation* or *modeling transformation*. Note that the input sphere is always centered at the origin; in each frame you transform the sphere from "centered at the origin" to some final position and orientation of the sphere in that frame. The difference between two consecutive frames gives the effect of animation.
2. Use an idle function to do the animation; see the sample program "Handout: rotate-cube-new.cpp".
3. To specify the direction of the axis about which the sphere is rotating, try to find two directions (vectors) that are both perpendicular to this axis, and then use an appropriate operation on these two vectors to find the axis vector.
4. Rotating an angle of θ should translate the sphere by a distance of $r\theta$, where r is the radius of the sphere, and θ is in the unit of **radian**. Use a **unit-length** vector for the translation direction so that the distance of the translation is correct.

(d) This part is to enhance the sphere rolling in **Part (c)**. In **Part (c)**, an initial solution is to treat each "rolling segment" (i.e., (A, B), (B, C), or (C, A)) *independently*, i.e., when rolling along (B, C), treat it as a *fresh start* and let the rotation amount be 0 when the sphere center starts at B, and similarly, when rolling along (C, A), let the rotation amount be 0 when the center starts

at C, and so on. However, this solution results in a *rotational discontinuity* (i.e., a *sudden change of orientation*) when the sphere changes the rolling direction, which is *undesirable*. Observe that when starting at B along (B, C), the sphere orientation should continue from *the same orientation* as at the time when *the sphere just finished rolling along the previous segment* (A, B)—where such orientation is the result of the *accumulated rotations from the beginning up to that point* (note that the sphere center may have already gone through A, B, C for many rounds); similarly for other rolling segments. Enhance your sphere rolling to produce the correct effect when the rolling direction changes.
(**Hint:** (1) Maintain a matrix M that represents the *accumulated rotation* from the beginning up to *the end of the previous rolling segment*. Initially set M to be the identity matrix. (2) Look at the comments in the beginning of the file "mat-yjc-new.h" (as part of "Rotate-Cube-New.tar.gz") to see how to represent the matrix M using the "mat4" type (a 4x4 matrix) functions/operations supported there.) **(30 points)**

(e) This part is to add some user-interface capabilities, as follows.

(1) The initial viewer position is put at $(7, 3, -10)$ as the VRP given in **Part (b)**. Initially, show the sphere *standing still* with center at the point A, together with the other parts of the scene (the quadrilateral "ground" and the three axes, etc.) as specified in **Part (b)**.
Implement a keyboard callback function such that the 'b' or the 'B' key begins rolling the sphere, and the 'x' and the 'X' keys respectively decrease and increase the viewer x-coordinate by 1.0, and similarly for the viewer y- and z-coordinates (with 'y', 'Y', 'z' and 'Z' keys). These position-changing keys should work both before and after the rolling begins, and both when the rolling is enabled or disabled. (See (2) below.) The viewer will always look at the origin $(0, 0, 0)$.

(2) The right mouse button is used to enable/disable the rolling. Before the "begin" key ('b' or 'B') is hit, the right mouse button has no effect. After the "begin" key is hit and the rolling begins, pressing the right mouse button once will stop rolling, and pressing the right mouse button again will resume rolling, right from where it stopped.

(3) The left mouse button is associated with a menu, with the following entries (and the corresponding actions):
(i) "Default View Point" (put the viewer back to the initial viewer position),
(ii) "Quit" (exit the program).
(Additional menu entries and their corresponding actions will be added in Assignments 3 and 4.)

You should implement the menu functions such that after a menu/submenu entry is selected, the rolling is resumed (even if previously the rolling was halted by clicking the right mouse button), right from where it stopped.
Hints: You can enable or disable your idle callback function `idle()` *anywhere* in the program by calling
`glutIdleFunc(idle)` or `glutIdleFunc(NULL)`, where `NULL` is defined to be 0. This is used to enable/disable the animation (for enabling/disabling the sphere rolling). Consult the sample code "Handout: rotate-cube-new.cpp" and "Handout: GLUT Mouse and Menus" for information about user-interface callback functions. **(20 points)**