Assignment 4

CS4533 Fall 2021

Due: Monday 12/20 11:55pm; Total score: 165 points

Note: This assignment has 5 pages.

This assignment extends Assignment 3, by implementing additional functions for the fog effects, blending, texture mapping, lattice effects, and a particle system for a firework effect. As before, you need to use shader-based OpenGL and any deprecated OpenGL commands are **not allowed**.

General Instructions: same as Assignment 1.
Note: You must turn in **all** your assignments **by 12/20 11:55pm** (note that this deadline is **after** the Final Exam (on Thur. 12/16)). **No assignments will be accepted after 12/20 11:55pm.**

(a) Recall from Assignments 2 and 3 that there is a menu associated with the left mouse button. Add a new menu entry " Fog Options " and implement it as a submenu with the following 4 entries:
(1) " No Fog ": disable the fog effect,
(2) " Linear ": enable the fog effect, using the linear fog equation, with the fog starting and ending values 0.0 and 18.0, respectively,
(3) " Exponential ": enable the fog effect, using the exponential fog equation, with the fog density value 0.09, and
(4) " Exponential Square ": enable the fog effect, using the exponential square fog equation, with the fog density value the same as (3).

Your program should allow the user to switch among all these fog options.
Define the fog color to be gray and *semi-transparent*: $(0.7, 0.7, 0.7, 0.5)$, and use the fog equations as discussed in class. You should implement the fog equations in the **fragment shader**, and blend the **final fragment color** of the object (as the **final result of shading, texture mapping, etc. (see parts (c) and (d)), as appropriate**) with the fog color according to the fog equations.
Tips: In the (fragment) shader, you can use GLSL functions `exp(x)` for e^x, `clamp(x, a, b)` to clamp the value of x to the range $[a, b]$, and `mix(x, y, a)` (where $a \in [0, 1]$) to obtain $x + a \cdot (y - x) = (1 - a)x + ay$.
Note: The final opacity after the fog effect should **preserve** the **fragment opacity before the fog effect**. This is particularly important for the shadow when we perform **shadow blending** (see **part (b)**), because blending is done by **blending the fragments** that are **final outputs of the fragment shader** using their opacities. **(30 points)**

(b) The shadow you produced in Assignment 3 is *opaque*; this part is to provide an option to produce a *semi-transparent* shadow, blended with the ground. (Recall from Assignment 3 that the shadow color is $(0.25, 0.25, 0.25, 0.65)$.) To do so, enable blending (by calling `glEnable(GL_BLEND)`) with the blending function specified as the *over* operation (by calling `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`) before drawing the shadow, and disable blending (by calling `glDisable(GL_BLEND)`) after drawing the shadow.

If the texture mapping is activated for the ground (see **part (c)** below), then the semi-transparent shadow should be automatically blended with the texture-mapped ground.

Recall from Assignment 3 that you used a multiple-pass process to make the shadow a decal of the ground. If the shadow is just a single polygon, then directly combining blending with this multiple-pass process should work correctly. However, the shadow actually consists of many overlapped sphere triangles after the shadow projection and hence the same problem related to the z-buffer occurs again. Modify your multiple-pass process to produce the correct effect.

In the same menu, add a new menu entry " Blending Shadow ", and implement it as a submenu with two submenu entries: " No " — do not enable shadow blending (and hence produce an opaque shadow), and " Yes " — perform shadow blending as described above. (Again, you can enable and disable blending by calling `glEnable(GL_BLEND)` and `glDisable(GL_BLEND)`.) **(15 points)**

(c) This part is to implement a simple texture mapping to your quadrilateral ground that indicates the x-z plane. A function `image_set_up()` and necessary global declarations are given in the file `https://cse.engineering.nyu.edu/cs653/assg4/texmap.c`. Put this function and the declarations into appropriate places of your program, as indicated in the comments of the file. The function generates an 8×8 checkerboard; your task is to texture map this checkerboard image to the quadrilateral ground to create a texture of a 12×10 checkerboard. Notice that the aspect ratio of the quadrilateral ground is 6/5, so the resulting checkerboard image preserves the aspect ratio and thus there is no distortion. You should assign the texture coordinates to the vertices of the quadrilateral appropriately (by setting up the **vertex buffer object** correctly), and make the texture map to repeat in both the s and t dimensions to "tile up" the quadrilateral. **You should NOT change the dimensions of the checkerboard image** — assign appropriate texture coordinates instead. Also, "modulate" the fragment and texture colors by performing a component-wise multiplication of these two colors; the resulting checkerboard should have light green and dark green colors. You can choose any filtering function for magnification and minification. (See the sample-code handout "Checker-New.tar.gz" for an example.)

In the same menu, add a new menu entry " Texture Mapped Ground ". Implement it as a submenu with two submenu entries: " No " — do not perform texture mapping on the ground, and " Yes " — perform texture mapping as described above. **(20 points)**

(d) This part is to implement a texture mapping on the sphere with texture coordinates **generated through computation in the vertex shader** (instead of being passed on to the vertex shader as an attribute), for both 1D and 2D texture mappings.

(1) **1D texture mapping**. The function `image_set_up()` in the file `texmap.c` as mentioned in **part (c)** also generates a 1D stripe image; your task here is to perform a *1D texture mapping* to map this stripe image onto the sphere to create contour lines (equally-spaced parallel stripes) on the sphere. The texture coordinates of the vertices of the sphere triangles should be generated through computation in the vertex shader, with the following options implemented via a keyboard function: hitting a key of 'v' or 'V' (meaning **"vertical"**) switches the texture coordinate to be $s = 2.5x$, and hitting a key of 's' or 'S' (meaning **"slanted"**) switches the texture coordinate to be $s = 1.5(x + y + z)$, for each vertex (x, y, z), where (x, y, z) is in the **object space** (i.e., **object frame** — this is the coordinate system of the **input sphere** where the sphere center is at the origin)

if a key of 'o' or 'O' is hit, and this is switched to the **eye space** (i.e., the **right-handed eye frame** where the eye is at the origin looking toward the $-z$ direction — this is the coordinate system after applying the `LookAt()` transformation on the world frame) if a key of 'e' or 'E' is hit. (Initially, when texture mapping on the sphere is first activated, the options should be "vertical" and object space.)

(2) **2D texture mapping**. Same as the 1D texture mapping in (1) except that now the 8×8 checkerboard image (as stored in **part (c)**) is used and that the texture coordinates are computed differently: for **"vertical"** (key 'v' or 'V'), they are $s = 0.5(x + 1)$ and $t = 0.5(y + 1)$, and for **"slanted"** (key 's' or 'S'), they are $s = 0.3(x + y + z)$ and $t = 0.3(x - y + z)$, for each vertex (x, y, z) which can again be in the object space or eye space as in (1). Also, change the green color of the (green-white) checkerboard image to a reddish color $(0.9, 0.1, 0.1, 1.0)$: in the fragment shader, if the texture color read is green (e.g., by checking its red-component value), then change the texture color to $(0.9, 0.1, 0.1, 1.0)$. (Note that only **one** checkerboard image is stored, which is green-white as given/stored in **part (c)**.)

In both (1) and (2), when applying the texture, choose to "modulate" the fragment color and the texture color by a component-wise multiplication of them. Also, between the wrapping modes (clamp, repeat) use "repeat", so that the result of (1) is a yellow sphere with multiple red contour lines, and the result of (2) is a sphere tiled up with a yellow and reddish checkerboard.

Notice that "vertical" and "slanted" give vertical and slanted contour lines/checkerboard respectively. Also, in the object-space option, the texture is "glued to the sphere" and rolling together with the sphere; in the eye-space option, the texture is "fixed to the eye" and gives the feeling that the sphere is "swimming through" the texture.

In the same menu, add a new menu entry " Texture Mapped Sphere ". Implement it as a submenu with 3 submenu entries: " No " — no texture mapping on the sphere, " Yes - Contour Lines " — perform the texture mapping of (1) above, and " Yes - Checkerboard " — perform the texture mapping of (2) above. You should **not** perform texture mapping when you draw the wire-frame sphere or when the sphere is drawn for producing the shadow.

Tips: To use multiple textures, you need to bind each texture to a **different texture unit** and set the sampler uniform variables in the fragment shader accordingly. See various places of "Note" in "Handout: checker-new.cpp". **(40 points)**

(e) This part is to implement a lattice effect on the sphere. In the vertex shader, compute a **new set** of texture coordinates (i.e., different from those in **part (d)**) for each vertex (x, y, z) of the sphere triangles, with the following options implemented via a keyboard function: hitting a key of 'u' or 'U' (meaning **"upright"**) switches the texture coordinates to be $s = 0.5(x + 1)$ and $t = 0.5(y + 1)$, and hitting a key of 't' or 'T' (meaning **"tilted"**) switches the texture coordinates to be $s = 0.3(x + y + z)$ and $t = 0.3(x - y + z)$, where (x, y, z) is the vertex coordinate in the **object space (i.e., object frame)**. This set of texture coordinates is passed along to the fragment shader as a varying variable (in the same way as in texture mapping), but no texture image is used. Instead, the corresponding texture coordinates in the **fragment shader** are used to decide whether the current fragment should be discarded (using the GLSL command "discard;" in the fragment shader) or not: if the (positive) fractional part of $4s$ (using the GLSL function `fract(4 * s)`) and the (positive) fractional part of $4t$ (using `fract(4 * t)`) are **both** < 0.35, then discard the current fragment (so that it is **not** rendered to the frame buffer (and **not** to the z-buffer either));

otherwise proceed as usual. In this way, a lattice effect on the sphere is produced.

Implement a keyboard function to associate with this lattice effect: hitting a key of 'l' or 'L' turns on this lattice effect, and hitting 'l' or 'L' again will turn off the lattice effect, and so on (i.e., 'l' or 'L' toggles the lattice effect on and off). Initially, the lattice effect is **"off"** and the texture-coordinate computing mode is **"upright"**.

Note:

1. If the shadow is "on", then the lattice effect on the sphere should also result in the corresponding lattice effect on the shadow, to be rolling together with the sphere, no matter whether " Blending Shadow " is " Yes " or " No ".
2. Suppose the **object-space** option is used in the texture mapping of **part (d)**. Then the **"upright"** lattice (key 'u' or 'U') will align with the **"vertical"** 1D/2D texture on the sphere (key 'v' or 'V'); similarly, the **"tilted"** lattice (key 't' or 'T') will align with the **"slanted"** 1D/2D texture on the sphere (key 's' or 'S'). **(20 points)**

(f) This part is to implement a particle system to produce a firework effect using shaders.

Your task is to produce a particle system consisting of N particles ($N = 300$ is suggested). Set up a vertex buffer object for these N particles as N vertices, rendered as points (`GL_POINTS`) with point size 3.0 (i.e., `glPointSize(3.0)`). In your OpenGL program, assign each particle a random velocity and a random color (but once assigned, the velocity and the color of each particle stay **fixed** throughout the process) — use arrays to store the velocities and the colors of all particles as their vertex attributes. For each particle, use the command `2.0*((rand()%256)/256.0-0.5)` once to assign the velocity in the x-direction, use the same command another time to assign the velocity in z, and use the command `1.2*2.0*((rand()%256)/256.0)` to assign the velocity in y. Note that the x- and z-velocities are each in the range $[-1.0, 1.0]$ whereas the y-velocity is in the range $[0.0, 2.4]$. Similarly, for each particle, use the same command `(rand()%256)/256.0` but call it once per color component to assign each of the r, g, b values; note that each color component has a value range of $[0.0, 1.0]$.

By Newton's law, for a particle with initial position (x_0, y_0, z_0) and a velocity of (v_x, v_y, v_z), its position $(x(t), y(t), z(t))$ at time t is $x(t) = x_0 + v_x t$, $y(t) = y_0 + v_y t - \frac{1}{2} g t^2$, and $z(t) = z_0 + v_z t$. Let each particle have the **same** initial position (x_0, y_0, z_0) that is close to the origin (of the world frame) but slightly higher than the ground, say at $(0.0, 0.1, 0.0)$ (i.e., $(x_0, y_0, z_0) = (0.0, 0.1, 0.0)$ for **every** particle). For the time t, use `(float) glutGet(GLUT_ELAPSED_TIME)` to get the elapsed time as t, which is in the unit of milli-second; use $0.001t$ so that it is in the unit of second. That is, suppose (X, Y, Z) is the particle location for the current frame at time t milli-second (where t is obtained by `(float) glutGet(GLUT_ELAPSED_TIME)`). The final formula for X is $X = x_0 + 0.001 * v_x * t$, and similarly for Z. The gravity acceleration g is 9.8 meters/(sec^2); in the world frame we let the length unit to be 10m, so g is 0.98 (10m)/(sec^2), and we have $\frac{1}{2} * (0.98) * (0.001t)^2 = 0.5 * (9.8) * 10^{-7} * t^2$. Moreover, the firework particles are light materials subject to the air friction; we model this effect by scaling the "downward y term" $-\frac{1}{2} g t^2$ with a factor of 1/2. Therefore the final formula for Y is $Y = y_0 + 0.001 * v_y * t + 0.5 * a * t * t$ where $a = (-4.9) * 10^{-7} = -0.00000049$. Note that at each frame **all** particles have the same t.

For each firework "animation cycle", each particle starts with the same initial location $(x_0, y_0, z_0) = (0.0, 0.1, 0.0)$, and its location (X, Y, Z) at the current frame is given by the final formulas for X, Y, Z above. When Y is smaller than 0.1, this particle **ceases to exist** and should be **discarded**

from the current frame. At some point, when almost all particles are discarded, the current animation cycle ends and the next animation cycle starts (where all particles start at the same initial location $(x_0, y_0, z_0) = (0.0, 0.1, 0.0)$ again), and the process repeats forever until the user stops the firework (see below).

In the same menu, add a new menu entry " Firework ". Implement it as a submenu with two submenu entries: " No " — disable the rendering of the firework, and " Yes " — enable the firework animation: if it is already enabled, continue; otherwise, start a **new** firework animation cycle **from beginning** (i.e., with all particles starting at $(0.0, 0.1, 0.0)$) and repeat the cycle forever as described above until " Firework $\rightarrow$ No " is selected. Note that when the firework is being animated, other parts of the program (e.g., sphere rolling, shadow, lighting, texture mapping, lattice, fog, etc.) should still perform **the same way as before**.

Note that you only need to use the **assigned particle colors** as stored in the vertex attribute array to render the particles; do **not** worry about lighting or fog effects, etc., on the particles.

Hint:

1. It is advised that you use a **separate** shader program (i.e., a **separate pair** of vertex and fragment shaders) for the firework particles. (Namely, overall you use **two pairs** of vertex and fragment shaders, one for the firework particles and the other for all the remaining objects.) For the firework particles, use the vertex shader to compute the location (X, Y, Z) for each particle/vertex.
2. In the fragment shader, you can call "`discard;`" to discard the current fragment without modifying (i.e., rendering it to) either the frame buffer or the z-buffer, as mentioned in **part (e)**. Use this approach to discard the particles/vertices with $Y < 0.1$ (in the **world frame**). The final Y value in the world frame is obtained in the vertex shader; set up/pass along suitable information to the fragment shader so that the fragment shader can perform a corresponding conditional discard correctly.
3. When the firework status is switched from "disabled" to "enabled", ideally we would like to "start the animation clock" from 0. However, the elapsed time t obtained by `(float) glutGet(GLUT_ELAPSED_TIME)` can only keep increasing as the program runs. To resolve it, at the moment of the status switch, call `(float) glutGet(GLUT_ELAPSED_TIME)` and store it into another variable t_{sub}, and use $t = 0$ initially and $t = t - t_{sub}$ subsequently to achieve the effect of "starting the animation clock from 0". Another related issue is about the time span of the animation cycle. Let this time span be T_{max} (this is a constant value decided by test-running the program), and suppose we use the modified t as discussed above. If both t and T_{max} were integers, then we would like to use the integer *mod* (%) operation $t = t \ \% \ T_{max}$, but now both t and T_{max} are floating-points. You should perform an equivalent *mod* operation for floats on t and T_{max}. **(40 points)**