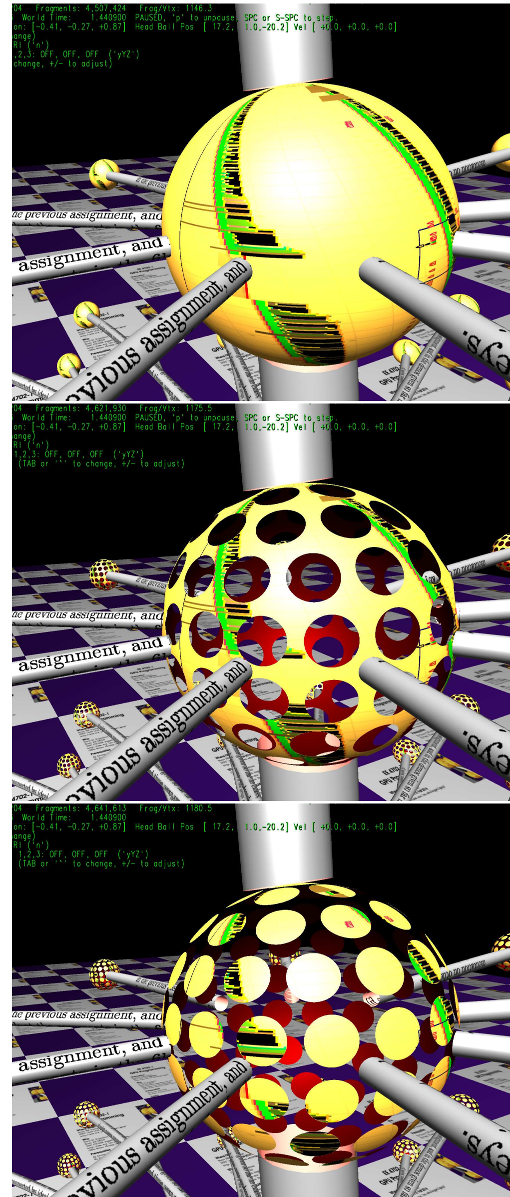
LSU EE 4702-1 **Homework 4** **Due: 16 November 2018**



Problem 0: Follow the instructions on the `https://www.ece.lsu.edu/koppel/gpup/proc.html` page for account setup and programming homework work flow. Compile and run the homework code unmodified. It should initially show a scene from the links code, the one showing a vaguely tree-like form, the *silly tree*, constructed from flexible links and balls. See the screenshot to the upper right. The screenshots at the middle and lower right were taken from correctly solved code.

Non-Assignment-Specific User Interface

Pressing h (head) will grab or release one end (to be precise, the ball at one end) and pressing t (tail) will grab or release the other end. (Actually, those keys toggle the fixed-in-space status of their respective balls.)

Press digits 1 through 4 to initialize different scenes, the program starts with scene 1. Scene 1 starts with the balls arranged in the tree-like form.

Press Ctrl = to increase the size of the green text and Ctrl - to decrease the size. Initially the arrow keys, PageUp, and PageDown can be used to move around the scene. Press (lower-case) b and then use the arrow and page keys to move the first ball around. Press l to move the light around and e to move the eye (which is what the arrow keys do when the program starts).

The + and - keys can be used to change the value of certain variables. These variables control things like light intensity and options needed for this assignment. The variable currently affected by the + and - keys is shown in the bottom line of green text next to VAR. Pressing Tab cycles through the different variables.

Look at the comments in the file `hw04.cc` for documentation on other keys.

Code Generation and Debug Support

The compiler generates two versions of the code, `hw04` and `hw04-debug`. Use `hw04` to measure performance, but use `hw04-debug` for debugging. The `hw04-debug` version was compiled with optimization turned off and with OpenGL error checking turned on.

Keys y, Y, and Z toggle the value of host Boolean variables `opt_tryout1`, `opt_tryout2`, and `opt_tryout3` and corresponding shader variables `tryout.x`, `tryout.y`, and `tryout.z`. The

user interface can also be used to modify host floating-point variable `opt_tryoutf` and corresponding shader variable `tryoutf` using the Tab, +, and - keys, see the previous section. These variables are intended for debugging and trying things out.

Sphere Rendering

The sphere can be rendered using two methods, the *tessellated sphere*, and the *true sphere* methods. The option in use is toggled by pressing (lower case) z and the current setting is shown next to Sphere in the green text. Both TRUE and HW04 indicate the true sphere method, TESS indicates the tessellated sphere method.

The tessellated sphere method is the one initially covered in class in which the sphere is approximated by triangles. A buffer object holds the triangles' vertex coordinates arranged for a triangle strip rendering pass. Those coordinates are in a local coordinate space in which the sphere center is at the origin and the sphere has a radius of 1. The number of triangles is based on variable `opt_slices`, larger values result in a closer approximation to a sphere.

The rendering pass for both tessellated and true spheres is done in routine `render_bunch_render` in file `shapes.h`. For both cases three additional buffer objects are prepared, one holding the sphere orientation, `sphere_rot`, one holding the sphere coordinates and radius, `sphere_pos_rad`, and one holding the sphere color `sphere_color`. There is one element in each buffer object for each sphere. An element of `sphere_rot` is a 4×4 matrix, which rotates a coordinate from local to global space (but does not perform translation or scaling). Because of differences in matrix layout, the corresponding matrix in shader code rotates from global to local space. Each element of `sphere_pos_rad` is a 4-element vector. The first three components form the coordinate of the sphere center in global space. The 4th component is the sphere's radius. Each element of `sphere_color` is a 4-element vector, which holds the red, green, blue, and alpha color components.

The tessellated spheres are rendered in an instanced pass. The pipeline inputs are taken from the vertex buffer object, and the number of instances is set to the number of spheres. The shaders in `hw04-shdr-sphere.cc` render the sphere (and its shadow volumes). The input to the vertex shader, `vs_main_instances_sphere()`, is a triangle (sphere surface) coordinate in local space. It converts it into global (object) space using the instance ID to get the location, orientation, and radius from the buffer objects described above. The color is retrieved by the fragment shader.

The rendering pass for the true spheres is also started in `render_bunch_render`. In this rendering pass there is no vertex shader input other than the vertex ID. The input primitive is a point and the number of vertices is set equal to the number of spheres. This pass uses code in `hw04-shdr-hw04.cc`. The geometry shader, `gs_main`, computes the coordinates of a square that will frame the sphere as seen from the user, and emits two triangles that form this square. The input to the fragment shader, `fs_main`, is the coordinate on this square corresponding to the fragment location. It computes the point on the sphere surface intersected by a line from the eye to the fragment location. If there is no intersection the fragment is discarded, otherwise a normal and texture coordinates are computed. To compute the texture coordinates the sphere rotation must be taken into account.

The code in `fs_main` has some variables to be used in this assignment, they are described in the problems.

Graphics and Performance Investigation Options

The user interface can be used to toggle various rendering options and for generating a screenshot.

Pressing F12 will write a screenshot to file `hw04.png` (`FOO.png`, where `FOO` is the name of the executable, such as `hw04-debug`). Any existing screenshot will be silently overwritten so be sure to rename files that you want to keep.

The rendering of shadows is toggled by o and the rendering of reflections it toggled by r. Their state is shown in the green text next to Effect:. Pressing n will toggle how surface normals are computed for tessellated spheres, the possibilities are to use the triangle normal or the sphere normal. The use of the triangle normals makes it easier to see the triangles from which the sphere was tessellated.

The scenes differ in the number of objects, which include spheres, links, and the platform (which for this assignment we'll consider one object). The rendering of objects by type can be toggled on and off by pressing !, @, #, for spheres, links, and the platform. See the green text line starting with Hide.

Display of Performance-Related Data

The top green text line shows performance in various ways. XF shows the number of display frames per frame buffer update. An ideal number is 1. A 2 means that two display frame update were done in the time needed to update the frame buffer once, presumably because the code could not update the frame buffer fast enough. GPU.GL shows how long the GPU spends updating the frame buffer (per frame), GPU.CU shows how long the computational accelerator takes per frame. The computational accelerator computes physics in this assignment. On the lab computers the computational accelerator GPU is different than the one performing graphics. CPU GR is the amount of time that the CPU spends on graphics, and CPU PH is the amount of time that the CPU spends on physics.

The second line, the one starting with Vertices, shows the number of items being sent down the rendering pipeline per frame. Clip Prim shows the number of primitives before clipping (in) and after clipping (out).

Problem 1: The code in `fs_main` (in file `hw04-shdr-hw04.cc` so that when `holes` is true (see the code) the sphere is not visible in the locations described below and when `lenses` is true the sphere is only visible in those same locations, in both cases show the inside of the sphere when appropriate. See the screen shots at the top of the assignment.

Variable `opt_n_holes_eqt` specifies roughly how many holes there should be along a circumference (such as the equator on a globe). Distribute the holes roughly evenly and so that 80% of a line along the sphere connecting the centers of two adjacent holes would be over holes (40% over one hole, 20% on the rendered surface, 40% over the other hole).

The hole locations should rotate with the sphere, just as texture coordinates do. That is, the part of a texture that touches a hole should not change as the sphere rotates.

Note that the fragment shader computes the same kind of `eta` and `theta` angle values used to construct the sphere, and from those computes the texture coordinates. Those same `eta` and `theta` values can be used to find hole locations. For example, if n is the number of holes along the equator, the η_h value for a hole might be $\eta_h = \lfloor \eta \frac{n}{\pi} + \frac{1}{2} \rfloor \frac{\pi}{n}$, where η is the `eta` value for the fragment. The η_h and θ_h value can be used to find the surface coordinates of a hole, and from that one can find the distance from the hole center to the fragment. The fragment is in the hole if this distance is below some threshold.

(*a*) Modify `fs_main` so that the sphere surface is not rendered where there are holes when `holes` is true, or so that the sphere surface is rendered only where there are holes when `lenses` is true.

(*b*) If there is a hole in the front surface of a sphere render the back surface of the sphere. When doing so compute the correct normal and check for the presence of a hole. If the back surface is visible set `front` to false.

Pay attention to:

- Shader compilation and link errors. These are sent to stdout when the program starts to run.
- Coordinate spaces. The code uses suffix `_l` for sphere local space, `_e` for eye space and `_o` for object (global) space.