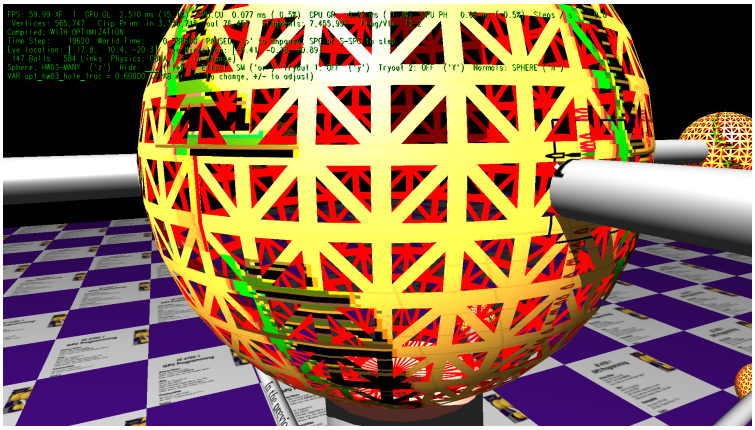
LSU EE 4702-1 **Homework 3** **Due: 6 November 2020**

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 (in `projbase/links`), the one showing a vaguely tree-like form, the *silly tree*, constructed from flexible links and balls. In the original links code the balls are ordinary spheres. In this assignment the spheres are to have triangular holes, as shown in screenshot below, which shows a closeup after Problem 1 or 2 is solved correctly. (Problems 1 and 2 do the same thing in different ways.)



Non-Assignment-Specific User Interface

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 `p` to pause the simulation.

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. Using the `Shift` modifier increases the amount of motion, using the `Ctrl` modifier reduces the amount of motion. Use `Home` and `End` to rotate the eye up and down, use `Insert` and `Delete` to rotate the eye to the sides. 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 forward through the different variables.

Look at the comments in the file `hw03.cc` for documentation on other keys.

Assignment-Specific User Interface

The sphere can be rendered by three different shaders, `HW03-ONE`, `HW03-MANY`, and `HW03-TRI`. The shader being used is shown to the right of `Sphere` in the green text. To cycle through the shaders press `z`.

The size of the hole is specified by variable `opt_hw03_hole_frac`. This can be modified using the UI, look for that variable name to the right of VAR.

Pressing `n` toggles between computing sphere lighting based on triangle normals, `TRI`, and sphere normals, `SPHERE`. When solving the problems it might help to rendering using triangle normals so that you can see where your hole is positioned within the triangle. But then switch to sphere normals to make sure the lighting is done correctly.

Graphics and Performance Investigation Options

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

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.

Pressing `F12` will write a screenshot to file `hw03.png`. 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.

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).

Code Generation and Debug Support

The compiler generates two versions of the code, `hw03` and `hw03-debug`. Use `hw03` to measure performance, but use `hw03-debug` for debugging. The `hw03-debug` version is compiled with optimization turned off and with OpenGL error checking turned on. You are strongly encouraged to run `hw03-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page.

When OpenGL error checking is on (as it is in the debug version) helpful error and warning messages will be printed about misuse or abuse of the OpenGL API. These will appear on the terminal window (which might be a gdb session) from which `hw03-debug` was started.

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.

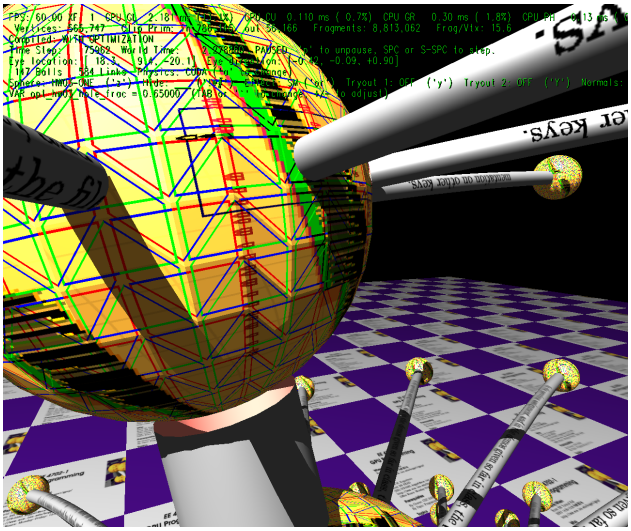
Problem 1: Modify the code in `gs_main_many_triangles` and perhaps `fs_main_many_triangles` in file `hw03-shdr.cc` so that each triangle used to tessellate the sphere has a triangular hole in it. (See the screenshot on the first page.) The relative size of the triangle should be based on the value of `opt_hw03_hole_frac`. Variable `opt_hw03_hole_frac` ranges from 0 to 1 (and can be controlled using the UI). When `opt_hw03_hole_frac` is 0.05 there should be a tiny hole, when `opt_hw03_hole_frac` is 0.95 there should be a large hole. At 0, there is no hole, at 1 the sphere itself should be invisible.

In this problem the hole effect should be achieved by having the geometry shader emit primitives for the area surrounding the hole.

- It should be possible to see the inside sphere surface and beyond through the hole.
- The texture should be applied in the same position as it would be if there was no hole. That is, when switching between the `HW03` and `TRI` shaders the position of the texture should not move.
- Don't forget to set the number of output vertices in the geometry shader. Don't set them to more than you need.
- Be sure to set normals correctly. To check that they are correct cycle through the different shaders using the `z` key. The coloring of the sphere should be the same for the `HW03` and `TRI` shaders.
- Do not expect shadows to work correctly. In particular, light won't go through the holes interrupting the shadow cast by the part of the sphere that is present.

Problem 2: Modify the code in shader routines `fs_main_one_triangle` and perhaps routine `gs_main_one_triangle` so that each triangle used to tessellate the sphere has a triangular hole in it, the same kind of hole as in the previous problem. In this problem do so by discarding a fragment if it would be within the hole.

To assist with this solution the geometry and fragment shaders have been modified to draw lines, see the screen shot to the right. Figure out how those lines are drawn, and modify the shaders so that there is a hole. Note that calling built-in function `discard` in the fragment shader discards the fragment. For your convenience there is a variable `hole_here` that can be set for those fragment at a hole.



- The size of the holes in the one-triangle shader must match the size in the many-triangle shader.

Problem 3: Which appears to be the better method? That is, which uses more computational resources? See if you can determine any difference in performance between the two.