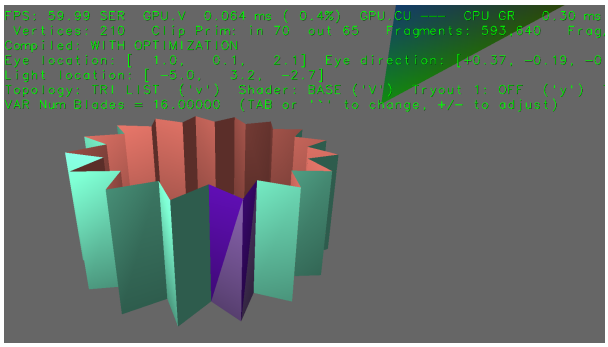
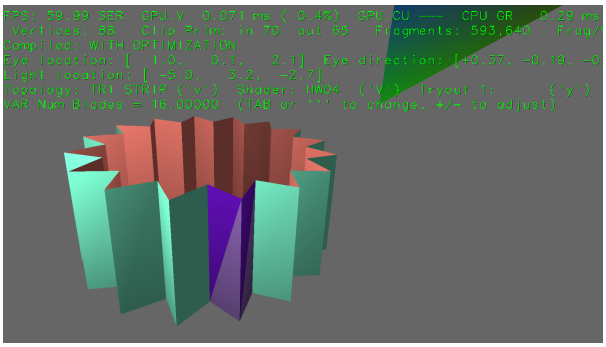
LSU EE 4702-1 **Homework 4 Due: 6 November 2022, 23:59**

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 look like the screenshot below on left. This is the gear from the midterm exam, which itself is similar to the paddle wheel from Homework 2 (though done in Vulkan). The screenshot on the right is taken from a correctly assignment. They look the same, but there is an important difference: the right image was rendered using fewer vertices.





User Interface

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size. Press `F12` to generate a screenshot. The screenshot will be written to file `hw04.png` or `hw04-debug.png`.

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).

Assignment-Specific User Interface

The scene can be reset by pressing `1`, `2`, and `3`. When `1` is pressed the number of turbines and their positions is randomly chosen. When `2` is pressed the number and positions are the same each time. When `3` is pressed the view will switch to the view used in taking the screenshots in this assignment. Pressing `p` will stop and start the turbines from spinning.

The gear can be rendered using two different shaders, *Base*, and *HW04*. Pressing `V` (upper-case) toggles between the two. The shaders' pipelines can be set to group vertices based on two topologies, *triangle lists*, and *triangle strips*. The `v` (lower-case) toggles between the two. The currently selected shader and topology are shown in the green text on the line starting with `Topology`.

The number of blades in the gear (okay, I should have called them teeth, or maybe cogs) can be changed by changing `VAR Num Blades`.

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 is compiled with optimization turned off. You are strongly encouraged to run `hw04-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page.

To help you debug your code and experiment in one way or another, the user interface lets you change variables. In this assignment these include the variables `opt_tryout1` and `opt_tryout2`, which are available both in CPU code and in shader code. You can use these variables in your

code (for example, `if ( opt_tryout1 ) { x += 5; }`) to help debug, to help familiarize yourself with how the code works, or to experiment with new ideas. Keys `y` and `Y` toggle the value of these variables. Their values are shown in the green text.

Display of Performance-Related Data

The top green text line shows performance in various ways. The number to the right of FPS shows the frame rate, the number of frames completed per second. On some displays 60 is the target frame rate and anything significantly lower than that indicates mediocre performance. Next, the green text shows whether frames are being prepared one at a time (serially), SER, or the steps in preparing a frame are being overlapped, OVR. In overlap mode commands for one frame are recorded while commands for a prior frame may be executing. For assignments early in the semester the mode will be kept at SER.

GPU.V shows how long the GPU spends updating the frame buffer (per frame), GPU.CU shows the execution of CUDA code per frame. CUDA code is physics in some assignments, but not this one and so the time should be shown as `---`. On some of 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 recording Vulkan graphics commands (or whatever it does in the callback installed by `vh.cbs_cmd_record.push_back`). CPU PH is the amount of time that the CPU spends on physics or whatever it does in the callback installed by the call to `vh.display_cb_set`.

For this assignment rendering is done by rasterization. (Other assignments will use ray tracing.) For rasterization 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). The next line indicates whether the code was compiled with optimization. Use the version without optimization for debugging and the version with optimization for performance measurements.

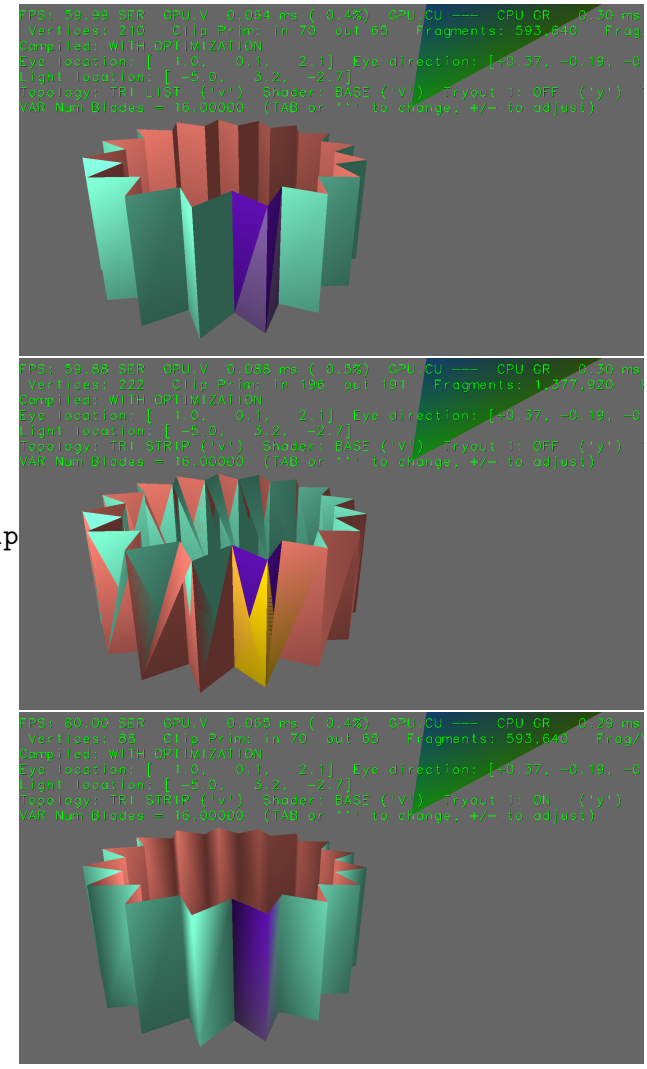
The `+` and `-` keys can be used to change the value of certain variables. These variables specify things such as the light intensity, sphere radius, and variables that will be needed for this assignment. The variable currently affected by the `+` and `-` keys is shown in the bottom line of green text. Pressing `Tab` cycles through the different variables. To locate variables which can be set this way, and to see how they were set, search for `variable_control.insert` in the assignment file.

Code Overview

Routine `World::render` is called each time the frame buffer is to be prepared. It is similar to other render routines used in class, the parts relevant to this assignment will be discussed here.

Some notes on how to work with the `VPipeline`, `VVertex_Buffer_Set`, and the buffer types can be found in this under-construction Vulkan course library note set.

Problem 1: The code in `render` prepares and uses two buffer sets, `bset_gear_list` and `bset_gear_strip`. In the unmodified code both of these contain vertices that correctly describe the gear for a triangle list topology. Buffer set `bset_gear_list` is used when the rendering is set to use the triangle list topology, which is correct. (The topologies are toggled by pressing `v` [lower-case].) Buffer set `bset_gear_strip` is used when the topology is set to a triangle strip. Since in the unmodified code the vertices in `bset_gear_strip` are arranged for a triangle list using it in this way is wrong, the middle screenshot shows the result.



Modify the code preparing `bset_gear_strip` so that the vertices are in triangle-strip order. Search for "Problem 1" in `hw04.cc` to find the code to modify. The bottom screenshot shows the gear with this problem correctly solved, but when using the base shader. Notice that the number of vertices in the bottom image is about one third the number in the other two. (In the next problem the blurriness will be fixed.)

Problem 2: When Problem 1 is correctly solved (and before this one is solved) the gear will appear blurry. This is because the base shader computes the lighted color at a vertex using the color and normal of the vertex. That does not work for a triangle strip because one vertex can be used for three triangles, not all of which have the same normal and color.

Modify the shaders in file `hw04-shdr.cc` to fix this problem plus the code preparing pipeline `pipe_gear_hw04` in `hw04.cc`. These are the shaders labeled `HW04` in the green text.

To fix the problem try the following: First, don't even bother sending normals into the rendering pipeline. Instead, have the rendering pipeline compute the normals. This should be done in the geometry shader. To fix the color problem, use a color associated with a single vertex when computing the lighted color. This too should be done in the geometry shader. And, if you've done those an still feeling like coding, try removing the integer pipeline input used for the color index, and instead pack the color index into the w component of the vertex coordinate. That will reduce the amount of data per vertex to just four floats.

When this problem is correctly solved the result should look like the right-hand screenshot on the first page of this assignment. (The number of vertices should be only slightly larger than the number of primitives.)

See the Open GL Shading Language 4.60 documentation, linked to the course References page, for a list of available functions, such as for computing a cross product, and other details on the language.