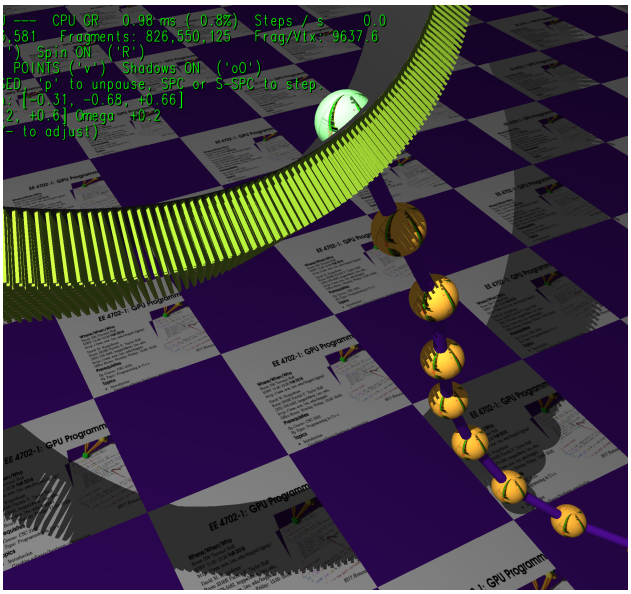
LSU EE 4702-1 **Homework 4** **Due: 15 November 2019**

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 string of beads hanging from a ring. The ring is shown as a cylinder surrounded by green-colored protrusions with tan-colored tops, based on the solution to Homework 3. You may notice that in the screenshot to the right that the protrusions are casting shadows. Efficiently casting those shadows is the goal of Problem 2 in this assignment.



Assignment-Specific User Interface

The setting for the rendering of shadows is shown to the right of Shadows on the HW04 line. There are three possible states, OFF, in which no shadows are rendered, ON, in which shadows are to be rendered normally and VOL, in which the protrusions' shadow volumes are visible. The rendering of shadows is toggled using `o` (lower-case oh) and the rendering of shadow volumes is toggled using `O` (upper-case oh). When shadow volume rendering is turned on the ring is not shown (because it gets in the way). Also, the outside (light-facing) face of shadow volumes is colored blue, the inside (shadow-facing) face is colored red. The colors are based on frequency shift of moving objects. Imagine that the front side is coming towards you and so it is blue shifted, and the back face is moving away, so it is red shifted. If the eye (actually the near plane of the view volume) is not itself in a shadow, all shadow volumes should appear blue.

In all of the shaders the balls, links, and ring should cast shadows. In the unmodified code the protrusions cast shadows only with the STRIP-PLUS shader, when Problem 2 is correctly solved protrusions will cast shadows with the POINTS shader, and will do so with much fewer fragments than with STRIP-PLUS.

The code in this assignment uses one of three shaders, shown to the right of Shader on the HW04 line of the green text. The shader in use can be changed by pressing `v`. Shader FIXED uses the OpenGL default shaders, as was used in previous assignments. Shader STRIP-PLUS is for use in Problem 1. When STRIP-PLUS is in use the protrusions are rendered using a single triangle strip but with the strip-restart feature developed in Homework 3.

When shader POINTS is used the protrusions are rendered using primitive type `GL_POINTS` in a `glDrawArrays` rendering pass. The shaders are in file `hw03-shdr.cc`. STRIP-PLUS uses vertex shader `vs_strip_plus` and geometry shader `gs_strip_plus`. When POINTS uses vertex shader `vs_points` and geometry shader `gs_points_common` with argument `shadow_volumes` set to `false`.

When the STRIP-PLUS shader option is selected shadow volumes are rendered by code in `vs_strip_plus_sv` and `gs_strip_plus_sv`. This code correctly renders the shadow volumes, the only changes to be made are for the improved triangle re-start to be implemented for the Strip Plus+ shader in Problem 1. When the POINTS shader option is selected shadow volumes are rendered by the code in `vs_points` and `gs_points_common` with argument `shadow_volumes` set to

`true`. In the unmodified `gs_points_common` there is an `if` statement guarding an area where the shadow volume code is to be placed. To find it search for "Put Problem 2 Solution".

In non-shadow rendering passes `fs_main` is used for both STRIP-PLUS and POINTS. In shadow rendering passes `fs_main_sv` is used for both of the shaders.

To change the protrusion size modify variable Bump Size as shown to the right of VAR.

Previous Assignment User Interface

When the simulation starts the head (first, zero) ball is attached to the ring and can slide freely along the ring. Pressing `r` (lower-case) will toggle between the head ball sliding freely and the head ball rotating along the ring at a fixed rate. The green text line starting with HW02: shows the state of the ball's attachment to the ring and of the ring itself. On this line the text to the right of Head Contr (head ball constraint) is RING-F if the ball is sliding freely, RING-A (animated) if the ball is rotating with the ring, FREE if the ball is not attached to anything, and LOCKED if the ball is fixed in space (does not change its position). The last two states can be obtained by pressing `h` (head). (Press `h` and `r` to familiarize yourself with the behavior, and be sure to read the next Common User Interface section which describes other available UI features, such as moving the viewer and pausing simulation.)

When in mode RING-A the rotation rate is determined by variable VAR Rotation rate (variable `hw01.omega` in the code). See the Common User Interface section below to see how to change that and other variables.

Pressing `f` toggles friction on and off, the state is shown to the right of Friction and by the color of the ring. When friction is turned on the ring color is yellowish, like sandpaper. Pressing `R` spins the ring.

Common User Interface

Press `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 between the `OC_Locked` and `OC_Free` constraint 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 almost vertically. Scene 2 starts with the balls arranged horizontally, and they will start swinging. Each time scene 1 and 2 are initialized the ring is positioned to a new position. In scenes 3 and 4 the ring is always positioned the same way.

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 tail 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 to change things like the light intensity, spring constant, and variables 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.

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.

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.

There are problems on the next page.

Problem 1: Recall that in Homework 3 Problem 1 a Strip Plus [tm] shader was developed in which a triangle strip could be restarted in the middle of a rendering pass by setting the normal in the first two vertices to zero. With Strip Plus it is possible to render several separated triangle strips in a single rendering pass, avoiding the overhead of multiple render passes that would have to be borne using ordinary triangle strips. Strip Plus can only be used, though, when the normal of a provoking vertex is to be applied to the entire triangle. That's not the case when triangles are approximating a curved surface, but lets just consider situations in which we are rendering flat surfaces and so using the provoking vertex normal is fine. Further, lets consider rendering situations in which we always intend to provide the true triangle normal. In that case we don't need to provide a normal at all, the geometry shader can easily provide one. With The Strip Plus + shader the beginning of a triangle strip is identified by setting the w component of the first two vertices to zero. The geometry shader computes geometric normals for the triangles.

(a) Modify the code in `hw04.cc` and the strip plus shaders in `hw04-shdr.cc` so that the w component of the vertex coordinate is used to mark the first two vertices of a triangle strip and so that the geometry shader computes the normals.

(b) Compare the amount of data sent by the Strip Plus (Homework 3) and the Strip Plus + shaders.

Problem 2: Routine `gs_points_common(bool shadow_volumes)` is similar to the shader in the solution to Homework 3 Problem 2, except that it has a Boolean argument. When that argument is true the geometry shader is supposed to emit shadow volumes rather than triangles forming the protrusions. In the unmodified routine it just returns when `shadow_volumes` is true.

Modify the routine so that it renders efficient shadow volumes. Efficient means that the shadow volume has six faces. That's in contrast to a shadow volume with 30 faces that would result in constructing a 3-face shadow volume for each of the 10 triangles making up the protrusion. For an example of how to construct a shadow volume for a triangle see routine `gs_strip_plus_sv`.

If the problem is solved correctly shadows will match those cast using STRIP-PLUS and the number of fragments will be about $\frac{1}{5}$ the number used by STRIP-PLUS.