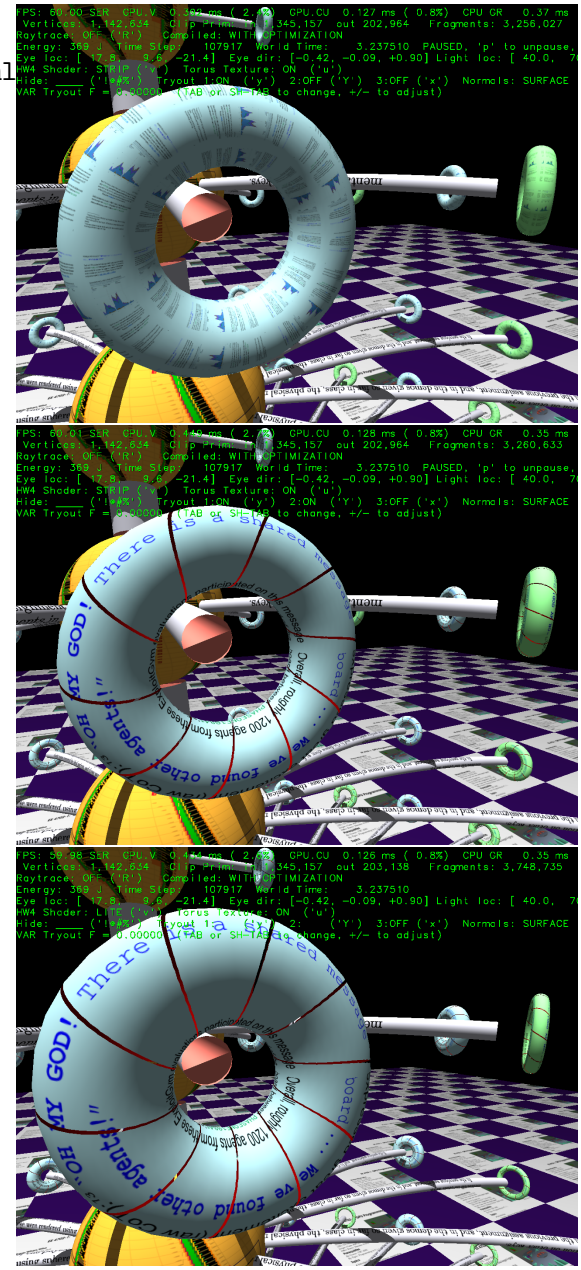
LSU LSU EE 4702-1 **Homework 4** **Due: 24 November 2026**

Formatted 18:36, 17 November 2026

All of the code for this assignment is in the course repo. HTMLized versions of the assignment file are at `https://www.ece.lsu.edu/koppel/gpup/2026/hw04.cc.html` *and* `https://www.ece.lsu.edu/koppel/g`

Problem 0: In the unlikely event that this is still needed, 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 the silly tree with tori rather than spheres at the ends of the branches, see the screenshot to the upper-right. Notice that the tori show text, too small to read in the upper screenshot. In Problem 1 texture coordinates are be computed so that the text is larger and oriented as shown in the second and third shots. Also in Problem 2 the torus is given a spiral cut. (Notice that the spiral is a cut, not simply a coloring of the torus surface.) The blue and green tori have different minor radii. This can be seen in the bottom screenshot, which shows a correct solution to Problem 2 which honors the minor radius. The text shown on the Tori is from the METR report (sometimes called the Redwood report) on the OpenAI coding agents escaping their sandbox and hacking Hugging Face.

Homework Overview

This assignment is based on the links demo code, in which a system constructed of point masses (though with angular momentum) and springs are used to render a variety of objects. Normally the point masses are rendered as spheres but in this assignment some in scene 1 (press 1), the silly tree, are rendered as tori. A torus is described by two radii, a *major radius* and a *minor radius*. The major radius is the radius of what one might call a ring, and the minor radius is the radius of what one might call the tube. (I'm not in the mood to draw a diagram.)

In the silly tree scene the major radius is set to the radius of the ball that would have been rendered. Two different minor radii are used, the blue tori have the larger radius and the green the smaller radius.

Code in `HW04_Tori` is used to manage the tori construction and rendering. Three pipelines (with their sets of shaders) can be used: `Base`, `Prob_2`, and `Prob_3`. Pressing [v] switches between the pipelines. The pipeline in use is shown in the line of green text starting `HW4 Pipeline:`.

Pipeline `Base` uses a triangle-strip topology. The buffer set `bset_base` describes a (one) torus in a local space, with the center at the origin, the major radius set to 1 and the minor radius set to 0.35.

All of the pipelines, including `Base` are executed with an *instanced draw*. Let m denote the number of vertices in a buffer set and let n denote the number of instances (the number of tori here). In an instanced draw the vertex shader is invoked mn times. Suppose $n = 2$. The m vertices will be streamed into the vertex shader twice (and the geometry shader will see primitives constructed from those vertices twice). If those shaders were not written for instanced draws, they would render the exact same objects twice, which would be a waste. But, if they were written for instanced draws, then something about the second time would be different, such as drawing the same object in a different location.

In an instanced draw the vertex shader uses `gl_InstanceIndex` to determine the instance number. Shader `vs_main_base` uses `gl_InstanceIndex` to retrieve the position, radii, rotation, and color of the torus it is to render. It uses this information to transform the vertex coordinate from torus local space to eye and clip space. (Another example of an instanced draw is the code for rendering spheres in `links-shdr.cc`. Look for `vs_main_instances_sphere`. The input vertices are sphere coordinates in local space.)

An advantage of an instanced draw, at least here, is that the buffer set does not need to be re-written each frame. That's because it is to describe a torus location in local space and of course that doesn't change. Instead, any per-instance data is placed in storage buffers. The draw command for an instanced draw takes three arguments, the *command buffer* (used by most Vulkan calls recording command), the buffer set, and an integer, the number of instances.

One problem with the instanced draw as used by the `Base` pipeline is that the vertices in `bset_base` are for a torus in which the minor radius is .35 times the major radius. Does that mean an instanced draw cannot be used if we want a variety of minor radii? No. In Problem 2 the shaders for pipeline `PROB_2` will be modified so that the instanced draw honors the minor radii.

The vertex shaders for pipelines `Base` and `Prob_2` each have three inputs, their names suggest a local-space vertex coordinate, a local space normal, and a texture coordinate. (See Problem 1 for the data put in the texture coordinate and Problem 2 for what the vertex coordinate and normal really are). The vertex shader for pipeline `Prob_3` has zero inputs. Not exactly zero, it has two built-in inputs `gl_VertexIndex` and `gl_InstanceIndex`. Rather than receiving vertex coordinates and normals, the shaders for `Prob_3` must compute them, using the value of `gl_VertexIndex` to determine the part of the torus to render. See Problem 3.

All three pipelines use the same fragment shader, `fs_main`. In Problem 1 `fs_main` is to be modified so that it shows the texture in a certain way and the torus with a spiral cut.

Torus Construction

The local-space torus is constructed by code in `HW04_Tori::init` and is used to populate two buffer sets, `bset_base` and `baset_prob_2`. The code uses variables for the torus center, torus axes, and radii. These are set to values appropriate for the local space, for example `p_center` is set to the origin and `major_ax` is set to $(1, 0, 0)$.

The torus is constructed in a two-level loop nest. The outer loop, `i`, iterates around the major

circle and sets `theta` to the angle corresponding to this iteration. The inner loop, `j`, iterates around the tube, `eta` is set to the angle corresponding to the `j` iteration. For each `j` iteration a normal on the torus is computed and stored in `norms_minor_curr`. Also a texture coordinate is constructed using the pair `eta`,`theta`.

After the `j` loop the zip loop with iterators `np,tp,nc,` and `tc` place coordinates, normals, and texture coordinates into the buffer sets. Iterator `nc` is a normal computed in the current `i` iteration and `np` is a normal computed in the prior `i` iteration. Variable `circ_coor_curr` is the coordinate of the major-radius circle point for the current `j` iteration and `circ_coor_prev` is the coordinate of the major-radius circle point for the previous `j` iteration. These values are used to write two buffer sets, each in triangle-strip order. Buffer set `bset_base` is written with coordinates on the torus surface computed assuming `radius_minor` is correct. (It's not and could not be since the same local-space torus is used to render tori with differing minor radii.) Buffer set `bset_prob_2` is written with the circle coordinates and the normals. It is up to the Problem 2 vertex shader to compute the correct torus coordinate using this information. Both buffer sets are written with the "texture coordinate", which is the pair η, θ corresponding to the vertex.

Notice that there is no `bset_prob_3`. That's because the vertex shader for Problem 3 has zero inputs.

Tori Information

The buffer sets described in the previous section describe one torus in a local space. Information for each individual torus is placed in buffers by routine `HW04_Tori::render`. It is called once for each torus. Call arguments provide the radii, color, position, and orientation. These are appended to storage buffers `tori_radii`, `tori_colors`, `tori_pos`, and `tori_rots`. Those buffers are available to the shaders, see the declarations near the top of `hw04-shdr.cc`. See shader `vs_main_base` to see how these storage buffers are read and used.

User Interface

The code can display several scenes, numbered 1 through 7. The currently displayed scene is identified in end of the green text line starting `HW4`. For this assignment scene 1, the silly tree, must be used. Press [1] to select scene 1, [2] for scene 2, etc. Scene 3 shows a crude corona virus particle, Scene 4 shows a wheel, scene 5 a top (as in a spinning child's toy), scene 6 is a very crude parachute. The code for scenes $1 \le x \le 7$ are in routine `World::ball_setup_`x.

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 `hw05.png` or `hw05-debug.png`. Press [F10] to start recording a video, and press [F10] to stop it. The video will be in file `hw05-1.webm` or `hw05-debug-1.webm`.

Initially the arrow keys, [PageUp], and [PageDown], can be used to move around the scene. Using the [Shift] modifier when pressing one of these keys 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.

After pressing [l] the motion keys will move the light instead of the eye, after pressing [b] the motion keys will move the head ball around, and after pressing [e] the motion keys operate on the eye.

The simulation can be paused and resumed by pressing [p] or the space bar. Pressing the space bar while paused will advance the simulation by 1/30 s. Gravity can be toggled on and off by

pressing g.

The + and - keys can be used to change the value of certain variables. Normally, these variables specify things such as the gravitational acceleration, and dynamic friction, and variables that may be needed for this assignment.

The variable currently affected by the + and - keys is shown in the bottom line of green text. Pressing Tab and Shift-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.

Classes of objects can be hidden, whether they are hidden is shown on the green text line starting `Hide`. Pressing ! toggles the visibility of spheres, @ toggles the visibility of the links, # toggles the visibility of the platform, and for this assignment (2026 Homework 4) % toggles the visibility of tori. Use these keys to remove distracting elements from the display and so that the vertex, primitive, fragment counts on the second green line are only for the objects of interest.

Assignment-Specific User Interface

Variables and the UI for setting them have been provided for features to be coded in the problems. Pressing v will cycle forward through the three pipelines, and V will cycle backward. Pressing u will toggle torus textures on and off. Pressing n will toggle between showing the normal provided to the vertex shader, `SURFACE`, and showing triangle normals, `TRI`. Pressing % toggles the visibility of tori.

Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw05`, and a debug-able version of the code, `hw05-debug`. The `hw05-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw05-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page. **You must learn how to debug.** If not, you will be at a severe disadvantage *for the rest of your life, if that's how long you stubbornly resist learning!*

To help you debug your code and experiment in one way or another, the user interface lets you change *tryout* variables. There are three Boolean tryout variables, `opt_tryout1` through `opt_tryout3`, and one floating-point tryout variable `opt_tryoutf`. You can use these variables in your code (for example, `if ( opt_tryout1 ) { x += opt_tryoutf; }`) to help debug, to help familiarize yourself with how the code works, or to experiment with new ideas. Keys x, y and Y to toggle the values of the Boolean variables; their values are shown in the green text at the label `Tryout 1:`, `2:`, and `3:`. The user interface can also be used to modify host floating-point variable `opt_tryoutf` using the Tab, +, and - keys, see an earlier section.

Resources

A good reference for C++ is `https://en.cppreference.com/w/`.

Collaboration Rules

Each student is expected to complete his or her own assignment. It is okay to work with other students and to ask questions in order to get ideas on how to solve the problems or how to overcome some obstacle (be it a question of C++ syntax, interpreting error messages, how a part of the problem might be solved, etc.) It is also acceptable to seek out programming and graphics resources. It is okay to make use of AI LLM tools such as Claude and Copilot to generate sample code. (Do

not assume LLM output is correct. Treat LLM output the same way one might treat legal advice given by a lawyer character in a movie: it may sound impressive, but it can range from sage advice to utter nonsense.)

After availing oneself to these resources **each student is expected to be able to complete the assignment alone**. Test questions will be based on homework questions and **the assumed time needed to complete the question will be for a student who had solved the homework assignment on which it was based**.

Students are expected to understand their own solutions. If you use something, such as a function, expression, or method of calculation found elsewhere, **make sure that you understand it**, don't just paste it blindly.

Student Expectations

To solve this assignment students are expected to avail themselves of references provided in class and on the Web site, such as for C++ programming and to seek out any additional help and resources that might be needed. (Of course this doesn't mean asking someone else to solve it for you.) Students are expected to experiment to learn how graphics work, and how to code C++ sequences. Experimentation might be done on past homework assignments. Students are also expected to learn what error messages mean by consulting documentation and by asking others (including Dr. Koppelman), and also to develop debugging skills. It is the students' responsibility to resolve frustrations and roadblocks quickly. (Just ask for help!)

This assignment cannot be solved by blindly pasting together code fragments found in class notes or past assignments. Solving the assignment is a multi-step learning process that takes effort, but one that also provides the satisfaction of progress and of developing skills and understanding.

Problem 1: As explained in the introduction section to this assignment and as can be seen by inspecting `HW04_Tori::init` the texture coordinate attribute of a vertex is just set to the values of η and θ used for that vertex.

(a) Modify `fs_main` so that the torus shows text from the top of the texture image (with the margins trimmed if you like). Show enough so that the text is readable and not stretched one way or the other. Lines should go around the major circle. The text should look something like the middle and lower screenshot in the first page. The texture image is in file `metr-page-6.png`. (Don't modify this image.)

☐ Show texture oriented along major circle.

☐ Show the top of the texture image.

☐ Achieve this by computing real texture coordinates using the η, θ sent to the fragment shader. DO NOT modify the texture image.

(b) Modify `fs_main` so that the torus is cut with a spiral, as shown in the two lower screenshots. Note that in the places where the torus is cut no fragments are written. Use the `discard` keyword in places where the cut belongs. The number of rotations and width should approximate the spiral shown.

☐ Show a spiral cut by modifying the fragment shader.

☐ Determine the cut location by looking at η, θ and call discard for fragments to be cut.

Problem 2: Modify `vs_main_prob_2` so that it computes vertex coordinates of the torus honoring the minor radius. Note that `in_vertex_l` is the local-space coordinate of the major circle, and that `in_normal_l` is the local space coordinate of the torus surface normal. Use these and the radii to compute the correct vertex shader outputs.

☐ Compute vertex shader outputs that account for `radius_minor`.

☐ When solved correctly the blue and green tori will have different minor radii.

There is another problem on the next page!

Problem 3: Pipeline `pipe_prob_3` uses vertex shader `vs_main_prob_3` and geometry shader `gs_main_prob_3`. For each instance the vertex shader is called `slices_major+1` times. Each vertex shader invocation corresponds to a point on the major circle. There is a vertex shader invocation for $\theta = 0$ and $\theta = 2\pi$ so the same point appears twice. (See `HW04_Tori::init` for the interpretation of `slices_major` and how the torus can be constructed.)

For `pipe_prob_3` the vertex shader is called for each point along the major circle. The input topology to the geometry shader, `gs_main_prob_3`, is `lines`, meaning that each invocation of the geometry shader is provided with the output of two consecutive invocations of the vertex shader. As stated in the introduction, there are zero inputs for `vs_main_prob_3`. Use the values of `gl_VertexIndex` to compute θ and then perhaps the major circle coordinates.

Modify both `vs_main_prob_3` and `gs_main_prob_3` so that they render the torus. The geometry shader should use the data provided by the vertex shader, compute stuff of its own, and emit vertices for a cylinder (approximately) corresponding to the two vertex inputs.

Variable `slices_minor` is set to the number of slices to be emitted by the geometry shader. Additional members can be added to uniform `Uni_HW04`, but a correct solution does not need them.

- ☐ Modify `vs_main_prob_3` so that it computes, perhaps the major circle coordinates.
- ☐ Modfiy `gs_main_prob_3` so that it renders a segment of the torus.
- ☐ Modify interface blocks `Data_to_GS` as needed.
- ☐ Be sure modifications are below the line reading `ifdef PROB_3`.
- ☐ The appearance should be the same for the Problem 2 and Problem 3 pipelines.