

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/hw03.cc.html>.

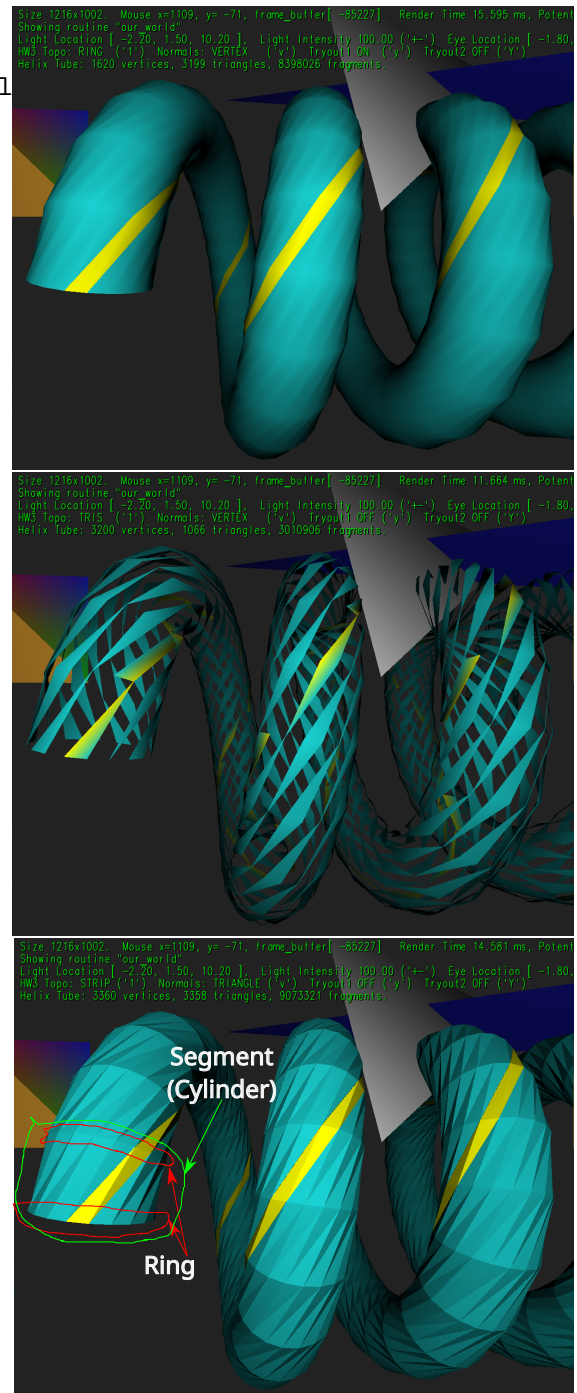
Problem 0: In the unlikely event that this is still needed, follow the instructions on the <https://www.ece.lsu.edu/koppel/gpup/proc.html> page for account setup and programming homework work flow.

Compile and run the homework code unmodified. It should initially show a cyan helix with a yellow stripe, plus four triangles. This shows the helix rendered using vertices in triangle-strip order, this order is shown by the green text to the right of HW3 Topo:, see the screenshot to the upper right. The upper shot shows the helix the way it's supposed to be rendered. In the lower screenshot, also taken from an unmodified assignment, the vertices are in triangle-strip order but the rendering system, Our_3D, expects them to be in individual triangle order. That's why the surface of the helix is not complete. When Problem 1 is correctly solved the helix rendered using individual triangles would look almost exactly like the helix rendered using a strip. There is a third way of ordering vertices, called *rings* for this assignment, that's the subject of Problem 2.

Homework Overview

In this assignment a tube helix is to be rendered using three different topologies (vertex orderings). The term *helix* refers to the curve parametricly described by $H(t) = H_0 + ta_z + \cos(\omega t)a_x + \sin(\omega t)a_y$, where a_x, a_y, a_z are local axes, ω is a rotation rate and H_0 is an initial position. That cyan thing in the screenshot is a tube helix, formed by sweeping a ring along a helix. The code uses the term helix to refer to the curve and tube helix to the 3D object.

The code places the coordinates, normals, and colors for the ring in containers `coors_ring`, `norms_ring`, and `colors_ring`. The first color in `colors_ring` is yellow, the others are cyan. Transformation matrices map the ring to



locations along the helix and also rotate slightly, which is why the yellow stripe wraps around.

The topologies used to render the tube helix can be chosen by pressing `1`. The topology in use is shown on the green text to the right of `HW3 Topo:`. The topologies are *triangle strip* (STRIP in green text, `Topology_Strip` enumeration constant), *individual triangles* (I-TRIS in green text, `Topology_Individual` enumeration constant), and *rings* (RING in green text, `Topology_Ring` enumeration constant). The triangle strip and individual triangle orderings work the same way as the like-named orderings in OpenGL and Vulkan. The rings ordering was made up for this assignment.

These orderings are described below using the tube helix as an example. To better see the triangles used to form the tube helix press `v` (which toggles whether to use tube-helix normals or triangle normals).

Let $v_0, v_1, \dots, v_i, \dots, v_{n-1}$ denote the n vertices provided to our rendering system. Using the variable names used in `hw03.cc:World::render_scene()`, the vertices provided to our rendering system are collected in containers `coors_os`, `normals_os`, and `colors`. Quantity n would be the number of elements in `coors_os`, and v_i would denote `coors_os[i]`, `normals_os[i]`, and `colors[i]`. (That is, v_i , represents all attributes provided for a vertex, in this case coordinate, normal, and color.)

Recall that in an individual triangle topology three vertices are provided for each triangle. Vertices v_0, v_1, v_2 form the first triangle, v_3, v_4, v_5 form the second triangle, etc. To render T triangles $3T$ vertices are needed.

If the set of triangles to be rendered can be ordered so that adjacent triangles share an edge then triangle strip ordering can be used. Vertices v_0, v_1, v_2 form the first triangle, vertices v_1, v_2, v_3 form the second triangle, v_2, v_3, v_4 form the third triangle, etc. The first and second triangles share edge v_1, v_2 , the second and third share v_2, v_3 , etc. So to render T triangles only $n = T + 2$ vertices are needed. That's almost $\frac{1}{3}$ the number of vertices!

The ring topology, which was made up for this assignment, is even better. Consider the problem of rendering a tube (the tube helix is an example). If triangle strips were used each vertex would need to be specified at least twice. The helix in this assignment is rendered by concatenating what are almost cylinders, each almost cylinder is called a segment in the code. A segment is specified by a ring of vertices of its top and a ring at its bottom. (The term *ring* is used in the code, the vertices in a ring form a circle.) (In the code the ring coordinates are in containers `tr_co_curr` and `tr_co_prev`.) Notice that the vertices of the top ring of one segment are identical to the vertices of the bottom ring of the next segment. Let n_r denote the number of vertices in a ring (in the code this is variable `n_ring_vtx`). Just for one segment, using individual triangles we'd need to specify $6n_r$ vertices per segment for the $2n_r$ triangles. A helix of n_s segments would need $6n_r n_s$ vertices, and most vertices are duplicated six times (there are six triangles that meet at a point, corresponding to the duplicated vertex).

Using triangle strips each segment would require $2(n_r + 1)$ vertices: $n_r + 1$ for the bottom ring, and another $n_r + 1$ for the top. The $+1$ is there to complete the last two triangles, which touch the first two. The entire tube helix requires $2n_s(n_r + 1)$ vertices to specify the $T = 2n_s n_r$ triangles, which is an improvement but still most vertices appear twice (since a top and bottom ring are shared inside the helix).

In the ring topology each ring is to be sent just once. Code in `World::render_scene()` already sends vertices in the correct order for a ring topology. Search for `case Topology_Ring:` to find this

code. Continuing to use n_r to denote the number of vertices in a ring, vertices $v_0, \dots, v_{n_r-1}$ form the first ring, vertices $v_{n_r}, v_{n_r+1}, \dots, v_{2n_r-1}$ form the second ring, and so on. Just given $v_0, \dots, v_{2n_r-1}$ `Our_3D` should render a cylinder when set for a ring topology. Suppose we append a third ring, $v_{2n_r}, \dots, v_{3n_r-1}$. Then two adjacent cylinders would be rendered. Vertices $v_{n_r}, v_{n_r+1}, \dots, v_{2n_r-1}$ would be the top of the first cylinder and the bottom of the second cylinder (assuming each succeeding ring is higher than the one before it).

In a triangle strip topology a triangle is rendered for each vertex beyond the first two. So for n vertices there would be $n - 2$ triangles. In a triangle ring two triangles are rendered for each vertex beyond the first n_r .

A convenient thing about individual triangles and triangle strips, is that adjacent vertices form a triangle, such as v_i, v_{i+1}, v_{i+2} . That's not the case with a ring topology. This is the logical place to explain precisely which vertices are grouped together to form triangles in a ring topology. Say, v_i, v_α, v_β where $\alpha =$. But, determining α and β are part of Problem 2.

Display of Performance-Related Data

The top green text line shows performance-related and other information. **Size** refers to the size of the window. **Mouse** refers to the coordinates of the mouse pointer. Coordinate (0,0) is at the **lower** left of the window. Text `frame_buffer[N]` shows the index of the frame buffer corresponding to the point under the mouse pointer. (In the assignment file `frame_buffer` is abbreviated to `fb`, for convenience.)

Render Time and **Potential Frame Rate** show the CPU time needed to write the frame buffer. They can be ignored for this assignment.

Much more important for this assignment (2026 Homework 3) are the amount of data and operations used to render objects. Data for the tube helix is shown on the last line of green text to the right of **Helix Tube:**. The number of vertices, triangles, and fragments are shown. In general, the amount of data sent from CPU to GPU is proportional to the number of vertices (all things being equal). The computation load on the GPU is proportional to the sum of the number of vertices and the number of fragments, usually the number of fragments is larger and dominates computation time. Pay attention to these numbers when solving the problems in this assignment.

General User Interface

Press `[Ctrl][=]` to increase the size of the green text and `[Ctrl][-]` to decrease the size. Pressing `[+]` (or `[=]`) increases light intensity, pressing `[-]` reduces light intensity. Press `[v]` to toggle between which normal is used to compute lighting. Initially lighting uses the normals sent to `Our_3D`. Pressing `[v]` toggles between using triangle normals and the provided normals (if any).

Initially the arrow keys, `[PageUp]`, and `[PageDown]`, can be used to move around the scene. After pressing `[l]` (lower case ell) the motion keys will move the light instead of the eye, pressing `[e]` the motion keys operate on the eye again. Press `[F12]` to generate a screenshot. The screenshot will be written to file `hw03.png` or `hw03-debug.png`.

The value of two Boolean debug-support variables, `opt_tryout0` and `opt_tryout1`, are shown in the green text, pressing `[y]` toggles `opt_tryout0` (between `true` and `false`) and pressing `[Y]` toggles `opt_tryout1`. The variables are available in routine `World::render_scene()` and `Our_3D::draw_rasterization()`, use them to try things out.

Assignment-Specific User Interface

Pressing `[1]` (digit one) toggles the topology of the vertices expected when rendering the tube

helix. The three possible topologies are *Individual Triangles* shown as I-TRIS in the green text to the right of HW3 Topo:, a *Triangle Strip* shown as STRIP, and *Rings*, shown as RING.

Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw03`, and a debug-able version of the code, `hw03-debug`. The `hw03-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, 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. **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`, `opt_tryout2`, and `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 `y`, `Y`, `x` to toggle the values of the Boolean variables; their values are shown in the green text at the label Tryout 1: and Tryout 2:. The user interface can also be used to modify host floating-point variable `opt_tryoutf` using the `Tab`, `+`, and `-` keys, see the previous section.

Resources

A good reference for C++ is <https://en.cppreference.com/w/>. A past homework assignment that may be helpful is 2025 Homework 3.

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: The code in `render_scene` near the case for `Topology_Individual` (search for “Problem 1”) inserts vertices in an order appropriate for a triangle strip, which of course is not the expected topology. Modify this code so that vertices are inserted in individual triangle order. Be sure to insert the same triangles as the triangle strip code (except for the redundant ones). It is easier to see where triangles are when using triangle-normal lighting, this can be toggled on and off by pressing `v`.

- ☐ Modify code in `render_scene` so that vertices are inserted correctly for the individual triangle topology.
- ☐ If solved correctly, the number of vertices will be $3\times$ that of triangle strip ordering, while the number of triangles and fragments will be slightly lower than triangle strip ordering.
- ☐ Code must be clear and reasonably efficient.
- ☐ Do not reformat existing code (unless that code is modified as part of solution).

Problem 2: Modify the code in `render_scene` and `draw_rasterization` so that the draw with the `Topology_Ring` ordering is shown correctly. Note that the code in `World::render_scene()` for case `Topology_Ring` is already correct. That code can be studied but should not be modified. The only change in `World::render_scene()` is to pass the number of vertices in a ring using whatever mechanism you implement in `Our_3D::draw_rasterization()`. (See 2025 Homework 3 for a similar issue with repeated fans.)

Most work for this problem is in `Our_3D::draw_rasterization()`. The code there works correctly for strips and individual triangles. Modify the code so that it works for rings. Part of this is changing the number of iterations of the main rendering loop, and changing the way `i0`, `i1`, and `i2` in that loop are computed.

- ☐ Send the number of vertices in a ring to our 3D in a way that matches how other rendering parameters are sent.`par`
- ☐ Implement ring topology rendering in `draw_rasterization`.
- ☐ Try to avoid adding another level to the big loop in `draw_rasterization`.
- ☐ Make sure that changes for rings do not break the other topologies!
- ☐ Be sure that coordinates, colors, and normals are correct.
- ☐ If solved correctly, the number of vertices will be $\frac{1}{2}$ that of triangle strip ordering, while the number of triangles and fragments will be slightly lower than triangle strip ordering.
- ☐ Code must be clear and reasonably efficient.
- ☐ Do not reformat existing code (unless that code is modified as part of solution).