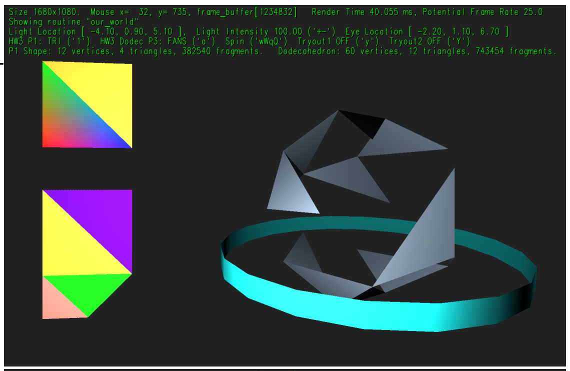
LSU EE 4702-1 **Homework 3** **Due: 20 October 2025**

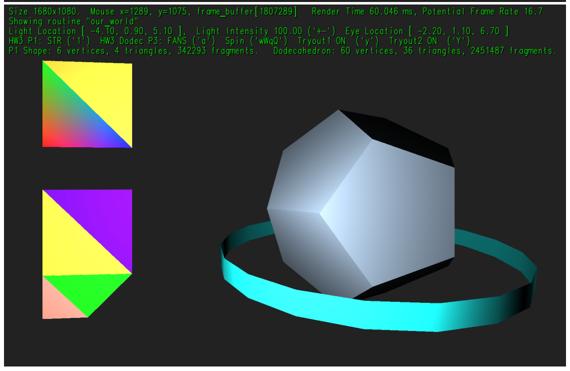
Formatted 18:49, 13 October 2025

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/2025/hw03.cc.html`.

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 with a cyan ring, colorful triangles, and a jumble of slate gray triangles. See the screenshot to the upper right. When the assignment is solved the jumble is replaced by a regular dodecahedron and the group of colorful triangles is rendered more efficiently. See the screenshot to the lower right, which is from a correctly solved assignment.



```
Size 1680x1080.  Mouse x=  32, y= 735, frame_buffer[1234832]   Render Time 40.055 ms, Potential Frame Rate 25.0
Showing routine "our_world"
Light Location [ -4.10, 0.90, 5.10 ],  Light Intensity 100.00 ('+-')  Eye Location [ -2.20, 1.10, 6.70 ]
HW3 P1: TRI ('1')  HW3 Dodec P3: FANS ('a')  Spin ('wWqQ')  Tryout1 OFF ('y')  Tryout2 OFF ('Y')
P1 Shape: 12 vertices, 4 triangles, 382540 fragments.   Dodecahedron: 60 vertices, 12 triangles, 743454 fragments.
```



```
Size 1680x1080.  Mouse x=1289, y=1075, frame_buffer[1807289]   Render Time 60.046 ms, Potential Frame Rate 16.7
Showing routine "our_world"
Light Location [ -4.10, 0.90, 5.10 ],  Light Intensity 100.00 ('+-')  Eye Location [ -2.20, 1.10, 6.70 ]
HW3 P1: STR ('1')  HW3 Dodec P3: FANS ('a')  Spin ('wWqQ')  Tryout1 ON  ('y')  Tryout2 ON  ('Y')
P1 Shape: 6 vertices, 4 triangles, 342293 fragments.   Dodecahedron: 60 vertices, 36 triangles, 2451487 fragments.
```

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` 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 `q`, `Q`, `w`, and `W` rotates the dodecahedron (or what will be the dodecahedron). Each keypress rotates by $\pm\pi/30$ radians around one of two axes. Rotating the dodecahedron may help in figuring out a bug when solving Problems 2 or 3, and it's also fun.

Pressing `1` toggles the topology of the vertices expected when rendering the group of triangles shown on the lower left. The two possible topologies are *Individual Triangles* shown as `TRI` in the green text to the right of `HW3 P1:`, and *Triangle Strips* shown as `STR` in the green text to the right of `HW3 P1:`. See Problem 1.

Pressing `a` toggles the topology of the vertices expected when rendering the dodecahedron. The two possible topologies are *Individual Triangles* (for Problem 2) and *Triangle Fans* (for Problem 3). The expected topology is shown to the right of `HW3 Dodec P2:` in the penultimate line of green text.

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 (2025 Homework 3) are the amount of data and operations used to render objects. Data is shown for the group of triangles in the lower right on the last line of green text to the right of `P1 Shape:`, and for the dodecahedron, to the right of `Dodecahedron:`. For each, 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.

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 two Boolean tryout variables, `opt_tryout1` and `opt_tryout2`. 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` and `Y` to toggle the values of the Boolean variables; their values are shown in the green text at the label `Tryout 1:` and `Tryout 2:`.

Regular Dodecahedron

A *regular dodecahedron* is constructed from 12 regular pentagons of edge length a by joining the pentagons at their edges so that each vertex is on the surface of a sphere, called the *circumscribed sphere*, of radius $r_c = a\phi\sqrt{3}/2$. There are a total of 20 vertices and 30 edges. The angle between two adjacent pentagons, called the *dihedral angle*, is $2\arctan(\phi)$ radians, where ϕ is the Golden Ratio, $\phi = \frac{1+\sqrt{5}}{2}$. The largest sphere that can fit inside a dodecahedron is called the *inscribed sphere*, its radius is $r_i = a\sqrt{\frac{\sqrt{5}\phi^5}{20}}$.

The code in `World::render_scene()` computes the coordinates of the 12 pentagons forming a dodecahedron and places them into container `coors`. Unlike Homework 2, the initial position is fixed (not randomly chosen). The orientation is specified by variables `dod_angle1` and `dod_angle2` which can be changed using keys `q` and `w`. For this assignment it will only be necessary to work with the coordinates in `coors`. There will be no need to transform them, find a set of unique values, nor to find interesting normals.

It is important to understand that there are 5×12 coordinates in `coors`. The first five are coordinates of the first dodecahedron face, etc. In Problem 2 it will be necessary to examine the coordinates for each face and determine the coordinates of triangles needed to render the face.

Code Overview

The code in this assignment is based on `cpu-only/demo-06-rend-pipe.cc`, which shows how code for rendering 3D objects might be separated into two parts, one for someone writing graphics applications (such as a game) and one for writing a graphics driver for a particular computing device (such as a GPU). In `demo-06` and this assignment the graphics application is in class `World`, and the graphics driver is in object `Our_3D`.

Class `Our_3D` exposes an API that users can use to draw 3D graphics. The API consists of calls (member functions) such as `vtx_coors_set` to specify coordinates, `transform_eye_from_object_set`, to specify a transformation to eye space, and so on. The API in `Our_3D` is based on the APIs provided by an OpenGL or Vulkan implementation, though much simpler.

Code in `World` uses this API (as described in class). To render a scene `World` must provide transformation matrices, vertex coordinates, colors, and lighting information. That is done in `World::render_scene()`.

Variable `gc` points to an `Our_3D` object. First, transformation matrices are provided to `Our_3D`:

```
// Compute transformation from Object Space to Eye Space ..
pMatrix_Translate center_eye(-eye_location);
pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
pMatrix eye_from_object = rotate_eye * center_eye;
// .. and give the transformation to Our3D:
gc.transform_eye_from_object_set(eye_from_object);

// Compute transformation from Eye Space to Clip Space ..
pMatrix_Frustum clip_from_eye
  ( -width_m/2, width_m/2,  -height_m/2, height_m/2,  qn, 5000 );
// .. and give the transformation to Our3D:
gc.transform_clip_from_eye_set(clip_from_eye);
```

We also need to specify the light location and color:

```
light_location = pCoor(-4.1,.9,5.1);
light_color = color_white * 100;
gc.light_location_set( light_location ).light_color_set( light_color );
```

To render something, coordinates and colors (and optionally normals) of each vertex must be sent. The code below populates containers `coors_os` and `colors` with coordinates and colors and sends them to `Our_3D`:

```
vector<pCoor> coors_os;
vector<pColor> colors;

// Yellow and Multicolored Triangles
coors_os << pCoor(-7,2,-2) << pCoor(-7,4,-2) << pCoor(-5,2,-2);
colors   << color_red      << color_green    << color_blue;

coors_os << pCoor(-7,4,-2) << pCoor(-5,4,-2) << pCoor(-5,2,-2);
colors << color_lsu_spirit_gold << color_lsu_spirit_gold
       << color_lsu_spirit_gold;

gc.vtx_coors_set(coors_os);      // Coordinates to render.
gc.vtx_colors_set(colors);       // Colors to render.
```

After execution of the code above `Our_3D` knows about the coordinates and colors. Remember (from class) that *vertex* refers to a coordinate, color and other information. The code above sent data on six vertices, for each a color and coordinate. Next we tell `Our_3D` how to group our vertices into triangles:

```
  gc.topology_set(Topology_Individual);  // Use individual triangle grouping.
```

The three possible options are `Topology_Individual`, `Topology_Strip`, and `Topology_Fan`. For the excerpt above individual triangles is correct because it matches how the vertices in `coors_os` are ordered. (Other topologies are considered in Problem 1 and 3.)

So far we've only provided information on what we want to do, but nothing has been drawn yet. To actually draw (write the frame buffer) the draw command must be issued:

```
  gc.draw_rasterization();         // Tell Our3D to render now.
```

If we want to render objects with different topologies, the need to be done with separate draw commands. See the code in `World::render_scene()`.

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 ChatGPT 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**.

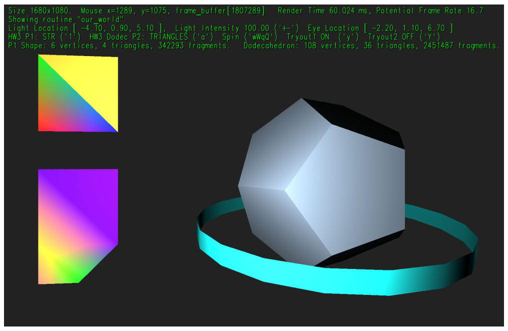
Student Expectations

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!)

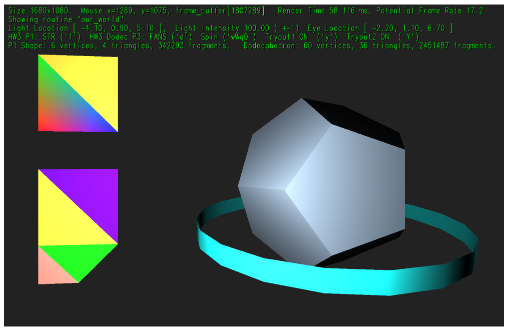
This assignment cannot be solved by blindly pasting together code fragments found in class notes or past assignments. Solving the assignment is a multi-step learning process that takes effort, but one that also provides the satisfaction of progress and of developing skills and understanding.

Problem 1: The code in `World::render_scene()` near "Problem 1 solution" inserts coordinates into `coors_os` in an order suitable for individual triangles, but the draw command expects a triangle strip:

```
if ( opt_hw3_p1_strips ) {
    // Render shape using a triangle strip. (When Problem 1 solved.)
    gc.topology_set( Topology_Strip );

    //  Problem 1 solution goes here.

    coors_os << pCoor(-6,-2,-2) << pCoor(-7,-2,-2) << pCoor(-7,-1,-2);
    colors   << color_salmon    << color_salmon    << color_salmon;

    coors_os << pCoor(-7,1,-2) << pCoor(-5,1,-2) << pCoor(-5,-1,-2);
    colors << color_lsu_spirit_purple << color_lsu_spirit_purple
           << color_lsu_spirit_purple;

    coors_os << pCoor(-7,-1,-2) << pCoor(-7,1,-2) << pCoor(-5,-1,-2);
    colors << color_lsu_spirit_gold << color_lsu_spirit_gold
           << color_lsu_spirit_gold;

    coors_os << pCoor(-7,-1,-2) << pCoor(-6,-2,-2) << pCoor(-5,-1,-2);
    colors   << color_green     << color_green     << color_green;
```



```
Size 1680x1080.  Mouse x=1289, y=1075, frame_buffer[1807289]   Render Time 60.024 ms, Potential Frame Rate 16.7
Showing routine "our_world"
Light Location [ -4.10, 0.90, 5.10 ],  Light Intensity 100.00 ('+-')  Eye Location [ -2.20, 1.10, 6.70 ]
HW3 P1: STR ('1')  HW3 Dodec P2: TRIANGLES ('a')  Spin ('wWqQ')  Tryout1 ON  ('y')  Tryout2 OFF ('Y')
P1 Shape: 6 vertices, 4 triangles, 342293 fragments.   Dodecahedron: 108 vertices, 36 triangles, 2451487 fragments.
```



```
Size 1680x1080.  Mouse x=1289, y=1075, frame_buffer[1807289]   Render Time 58.116 ms, Potential Frame Rate 17.2
Showing routine "our_world"
Light Location [ -4.10, 0.90, 5.10 ],  Light Intensity 100.00 ('+-')  Eye Location [ -2.20, 1.10, 6.70 ]
HW3 P1: STR ('1')  HW3 Dodec P3: FANS ('a')  Spin ('wWqQ')  Tryout1 ON  ('y')  Tryout2 ON  ('Y')
P1 Shape: 6 vertices, 4 triangles, 342293 fragments.   Dodecahedron: 60 vertices, 36 triangles, 2451487 fragments.
```

(*a*) Modify the code so that vertices are in an order suitable for a triangle strip. When solving this make sure the green text shows `STR` to the right of `HW3 P1:` (press `1` if necessary). In a correct solution the shape should be rendered using six vertices forming 4 triangles. (In an unsolved file there will be 12 vertices forming 10 triangles.) Before the next subproblem is finished the triangle colors will blur into each other, see the screenshot to the upper left.

(*b*) When rendering individual triangles we sometimes want each vertex to have its own color, perhaps for colorful gradient effects such as the red/green/blue triangle in the upper left. But, more often we want one color for a triangle. In the unmodified code `Our_3D` uses a separate color for each vertex. Modify `Our_3D::draw_rasterization()` so that when triangle strips are rendered the color of a triangle is based on one vertex, not a blend. If necessary modify the solution to the previous subpart so that the colors are correct. When solved correctly the Problem 1 shape triangle colors will appear as in the screenshot to the upper right.

Problem 2: The code in `World::render_scene()` renders the dodecahedron two ways, using individual triangles (this problem) or triangle fans (next problem). For individual triangles the code takes dodecahedron coordinates from `coors` and places them in `coors_os`:

```
switch ( opt_hw3_variation ){
case HW3_Triangles:
  // Homework 3 Problem 2

  for ( size_t i=0; i<coors.size(); i+=5 )
    for ( int j=0; j<5; j++ )
      coors_os << coors[ i + j ];

  gc.topology_set(Topology_Individual);
  break;
```

The code above won't render the dodecahedron correctly because the order of vertices (and their number) in `coors` does not specify individual triangles.

Modify the code so that it emits individual triangles. To do so vertices in `coors` must be inserted into `coors_os` multiple times. Do not worry about colors or normals, they are inserted further below in the code.

To see the effect of these changes make sure that the green text shows `TRIANGLES` to the right of `HW3 Dodec P3:`.

Problem 3: The code in `World::render_scene()` for rendering the dodecahedron using triangle fans is correct (the code in `Our_3D` is not):

```
case HW3_Fans:
  // Problem 3
  // Don't change this code, modify the code in Our_3D.
  gc.topology_set(Topology_Fan,5); // Use a fan grouping.
  coors_os = coors;
```

The `topology_set` command above tells `Our_3D` to expect triangle fans **and** that each triangle fan will be described by five vertices. It then just copies the dodecahedron vertex coordinates, `coors`, into the container we will send to `Our_3D`, `coors_os`. The vertices in `coors_os` are grouped into pentagons. These work as triangle fans.

The problem is that the code in `Our_3D` works with individual triangles and triangle strips, but not triangle fans:

```
const bool topology_strip = topology == Topology_Strip;
const bool topology_fan [[maybe_unused]] = topology == Topology_Fan;
size_t inc = topology_strip ? 1 : 3;
const ssize_t stop = coors_os.size() - 2;

for ( ssize_t i=0; i<stop; i += inc )
  {
    ssize_t idx_0 = i;
    ssize_t idx_1 = i + 1;
    ssize_t idx_2 = i + 2;

    pCoor o0 = coors_os[idx_0],  o1 = coors_os[idx_1],  o2 = coors_os[idx_2];
```

Modify this code so that it renders n-vertex triangle fans correctly, where n is the value of variable `topology_n_vertices`. For the dodecahedron we set $n = 5$, but the code should work for any n. Code for implementing a triangle fan appeared on the 2024 midterm exam. However, in the midterm exam there was just one fan per draw, here there can be many fans, each consisting of `topology_n_vertices` vertices. The midterm exam code is in the repo in the same directory as the 2024 homework assignments.

To see the effect of these changes make sure that the green text shows `FANS` to the right of `HW3 Dodec P3:`.

When this is solved correctly the green text should show $5 \times 12 = 60$ vertices and $3 \times 12 = 36$ triangles for the dodecahedron.

Of course, `Our_3D` should continue to show other geometry correctly.