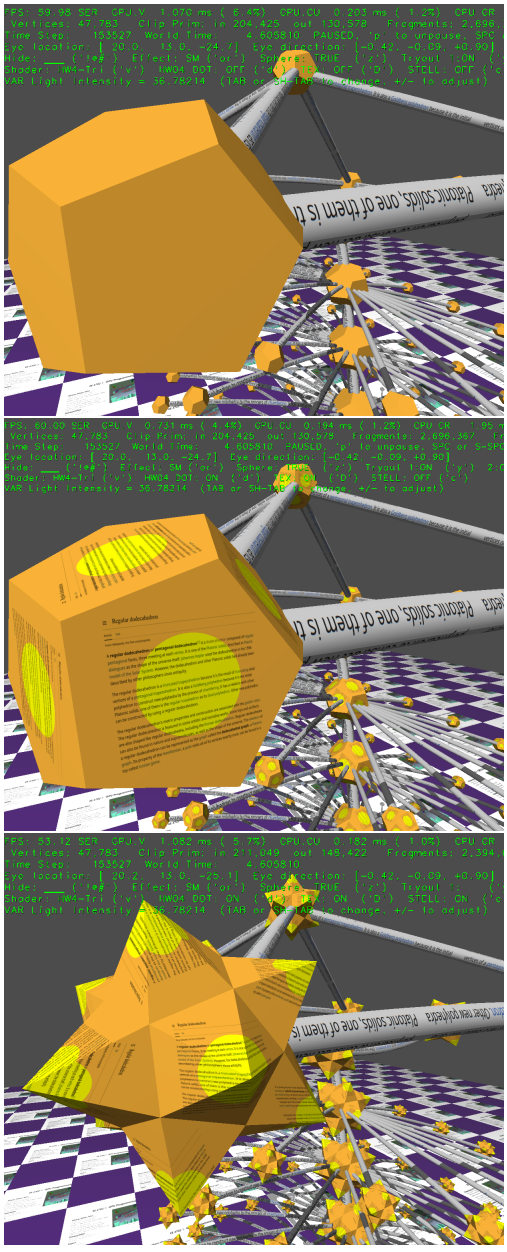
LSU EE 4702-1 **Homework 4** **Due: 21 November 2025**

Formatted 10:15, 18 November 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/hw04-shdr.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 canopy constructed from dodecahedra and links. (This is scene 2 from the links demo code, but with the balls replaced by dodecahedra.) See the screenshot to the upper right. When Problem 1 is correctly solved each dodecahedron face has a yellow dot and shows text from the Wikipedia article on dodecahedra. See the middle screenshot to the right. When Problem 2 is correctly solved the dodecahedra are *stellated*, meaning that the pentagon on each face is replaced by a five-sided pyramid. See the screenshot to the lower right.

User Interface

The code can display several scenes, numbered 0 through 7. The currently displayed scene is identified in the penultimate line of green text. For this assignment any scene can be used, Scene 1 is the default because the eye is positioned close to a large dodecahedron. Press `0` to select scene 0, `1` for scene 1, 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 $0 \leq x \leq 7$ 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 `hw04.png` or `hw04-debug.png`. Press `F10` to start recording a video, and press `F10` to stop it. The video will be in file `hw04-1.webm` or `hw04-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.

Assignment-Specific User Interface

Variables and the UI for setting them have been provided for features to be coded in the problems. Pressing `d` will toggle the value of Boolean `opt_hw04_dot`. When correctly solved the gold dot will be shown only when `opt_hw04_dot` is `true`. Pressing `D` will toggle the value of Boolean `opt_hw04_tex`. When correctly solved textures will be shown on the dodecahedron faces only when `opt_hw04_tex` is `true`. Pressing `c` will toggle the value of Boolean variable `opt_hw04_stel`. When correctly solved the dodecahedra faces will be stellated when `opt_hw04_stel` is `true`. Also available for stellation is `float` variable `opt_hw04_height_stel`, which indicates the height of the stellation pyramid in dodecahedron local space units. A value of 0 indicates no stellation. The value can be changed using the UI for changing variables described above, it will be labeled using the variable name, `opt_hw04_height_stel`. The values of the Boolean variables are shown on the penultimate line of green text. See the screenshots on the first page of this assignment.

The balls can be rendered using three different sets of shaders, labeled `HW4-Tri`, `HW4-Clean`, and `Balls`, pressing `v` cycles between these shaders. Shader set `Balls` are the ones used in the links code, they show the links attached to balls. Shader sets `HW4-Tri` and `HW4-Clean` render dodecahedra instead of balls. Shaders for both of these are in file `hw04-shdrs.cc`. The `HW4-Tri` shaders, `vs_main`, `gs_main`, and `fs_main` are to be used for the solution to this assignment. The `HW4-Clean` shaders, `vs_main_clean`, `gs_main_clean`, and `fs_main_clean`, are there for reference or for light experimentation.

Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw04`, and a debug-able version of the code, `hw04-debug`. The `hw04-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, 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. **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.

Code Overview

For this assignment only shader code, in file `hw04-shader.cc`, is to be modified. That said, the description here starts in the host code, which may be the subject of the next homework assignment.

This assignment is based on the links demo code, in directory `proj-base/v-links`. It has been modified so that the balls, by default, will be rendered using newly inserted dodechedron code. That code is in routine `World::render_doda`. (Pressing `v` will switch between the ball shader and the dodecahedron shader(s).)

Routine `World::render_doda` is called when the frame buffer (actually surface) is to be written, normally 60 times a second. The routine does any required pipeline setup, populates a buffer set to show each ball as a dodecahedron, and then *records* a *draw* with the buffer set providing vertex shader inputs to the pipeline. Be warned: the use of the pipeline is inefficient, something that will be fixed in a follow-up assignment.

Pipeline `pipe_doda` is set up by:

```
if ( !pipe_doda )
  pipe_doda
    .init( vh.qs )
    .ds_follow( puni_light_curr )
    .ds_follow( transform )
    .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
    .ds_use( sampler, texid_hw )
    .shader_inputs_info_set<pCoor, pColor, pMatrix>()
    .shader_code_set
    ("hw04-shdr.cc",
     "vs_main();",
     "gs_main();",
     "fs_main();", "#define HW4_TRI\n" )
    .topology_set( vk::PrimitiveTopology::eTriangleList )
    .create();
```

This sets up a Vulkan rasterization pipeline as follows. Call `shader_code_set` specifies that the shader code is to be read from from file `hw04-shdr.cc` and that the names of the vertex, geometry, and fragment shaders are executed using `vs_main();`, `gs_main();`, and `fs_main();`. The `.topology_set` call specifies that the inputs to the geometry shader are formed using a triangle list topology. The `shader_inputs_info_set` call specifies that the buffer set will include coordinates, colors, and 4×4 matrices.

The calls starting `ds_` specify other data available to shaders. (The "ds" is for *descriptor set*.) The `puni_light_curr` call provides lighting information. The solution to this assignment (Fall 2025 Homework 4) does not involve lighting, but those who are curious might look in `shader-include/light.h` and `shader-common-generic-lighting.h`. Call `ds_follow(transform)` specifies that `transform` is to be bound to a uniform. Look for `BIND_UNIFORM` in `hw04-shdr.cc`. Call `ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )` specifies another uniform. See file `links-shdr-common.h` for the members of that uniform. Finally, call `ds_use( sampler, texid_hw )` sets up a texture unit. Variable `sampler` indicates how the texture will be filtered, and `texid_hw` is the image object holding the texture. For this assignment the texture is constructed from a screenshot of the Wikipedia regular dodecahedron page. Remember that for this assignment the host code, including the code above, is not to be modified.

The code that follows writes, if necessary, dodecahedron coordinates to vector `dod_coors`. The dodecahedron coordinates are in a local space with the dodecahedron center at the origin and with an *inscribed radius* of 1. That is, a sphere of radius 1 would be inside the dodecahedron and touch the center of each dodecahedron face. This is the same code used in Homework 2 and 3.

Next, the code inserts vertices to render a dodecahedron for each ball. The code starts by computing a transformation matrix that will map a coordinate in the dodecahedron local space to

the global space used by ball (such as for `b->position`). The transformation translates the center of the dodecahedron to the ball center, scales the dodecahedron to the same radius as the ball, and rotates it to the same orientation:

```
  for ( Ball* b: balls )
    {
      pMatrix_Translate tra(b->position);
      pMatrix_Scale sca(b->radius);
      pMatrix rot(b->omatrix);
      pMatrix m = tra * sca * rot;
```

Next the code inserts dodecahedron vertex coordinates, colors, and the transformation matrix for this ball into the buffer set. It does so using the local-space coordinates in `dod_coors`, emitting three triangles for each face (just as in the solution to Homework 3 Problem 2):

```
      for ( size_t i=0; i<dod_coors.size(); i+=5 )
        for ( int j=1; j<4; j++ )
          bset_doda
            << dod_coors[ i ]
            << dod_coors[ i + j ]
            << dod_coors[ i + j + 1 ];
```

Note that in the code above each `i` loop iteration emits the three triangles for one face. All three of those triangles start with the same vertex, `dod_coors[i]`. This fact is important for solving this assignment.

Remember that the values in `dod_coors` are in the dodecahedron local space. We leave it to the shaders to transform them into global coordinates using `m`. The code for this assignment uses an inefficient method to send `m` to the shaders: as a vertex shader input. The next lines put `m` and also the color into the buffer set:

```
      for ( int i=0; i<3*3*12; i++ ) bset_doda << b->color << m;
}
```

The loop above iterates 108 times (three vertices per triangle, three triangles per face, twelve faces per dodecahedron). For each iteration it inserts the exact same data. That's wasteful!, why not send it just once? We will, but not in this assignment.

Resources

A good reference for C++ is `https://en.cppreference.com/w/`. For an examples on how to insert coordinates into vertex lists (*e.g.*, `coors_os`, `colors`) see 2024 Midterm Exam Problem 1.

Information on OpenGL Shading Language can be found in the reference, OpenGL Shading Language 4.60, linked to the course references page, `https://www.ece.lsu.edu/koppel/gpup/ref.html`.

Like Problem 1 in this assignment, texture coordinates are the subject of 2023 Homework 4 Problem 3 and 2024 Homework 5 Problems 1 and 2. Unlike those past assignments, in this one the texture coordinates are to be computed in the shaders (rather than being provided as vertex shader inputs).

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.)

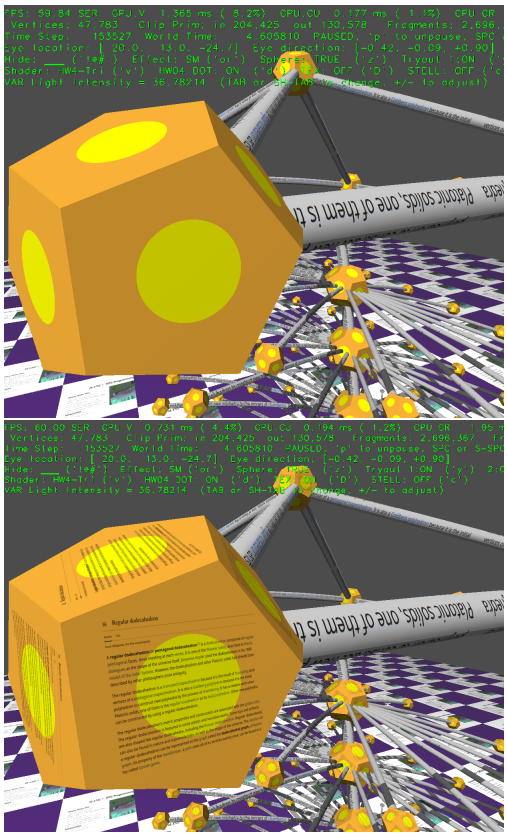
After availing oneself to these resources **each student is expected to be able to complete the assignment alone**. Test questions will be based on homework questions and **the assumed time needed to complete the question will be for a student who had solved the homework assignment on which it was based**.

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: As described below, modify the code to show the gold dot (top right screenshot) and the texture (lower right screenshot). These two tasks may seem unrelated, but they're not. In both cases one needs to provide the fragment shader with a face-local coordinate of the fragment. (Consider a fragment at the center of a face. Its face-local coordinate, which has two dimensions, might be $(0, 0)$. That face-local coordinate would be $(0, 0)$ for a fragment at the center of any dodecahedron face.) For the gold dot, the face-local coordinate could be used to determine the distance from the center (and so whether to use gold or the default color). For the texture, the face-local coordinate could be used to compute texture coordinates.



The code in the geometry shader computes quantities that might be useful for computing face-local coordinates. This includes `ctr_pentagon_e`, the center of the face. Also useful, is the fact that `In[0]` is the same vertex for all three triangles and so could be used, along with `tn`, the normal, for finding a face-local a_x and a_y. Let L_e be the origin of local space in eye-space coordinates (perhaps `ctr_pentagon_e`), and a_x and a_y be the local space axes (in eye space). Let P_e be an eye-space coordinate. Then the x component in local space would be $(P_e - L_e) \cdot a_x$, where $v \cdot u$ is the dot product of v and u.

A face-local coordinate would need to be computed for each triangle vertex and written to a new geometry shader output, fragment shader input.

To use your own image(s) edit the lines:

```
texid_syl.init
  ( vh.qs, P_Image_Read( vh.gp_root_get() / "vulkan" / "gpup.png", 255 ) );
texid_emacs.init( vh.qs, P_Image_Read("mult.png",-1) );
texid_hw.init( vh.qs, P_Image_Read("wiki-dod.png",255) );
```

and change the file names, such as `wiki-dod.png`, to files containing your own images. The dodecahedron faces use texture id `texid_hw`, the balls (when not shown as dodecahedra) use `texid_emacs`, and the checkered platform uses `texid_syl`. The images are read from files by the Image Magick library. The library can read many formats, including JPEG, PNG, and PDF. Use the command `magick -list format` to get the list.

Those who are just not satisfied guessing their own RGB components for gold have multiple options to procrastinate. For more information on the old X11 colors used in the course `~/gp/include/colors.h` see `https://en.wikipedia.org/wiki/X11_color_names`. Those who want to strike just the right balance when choosing a color that will be adjacent to others might visit `https://www.sessions.edu/color-calculator/`. Those who want to reflect on our understanding of color might visit `https://xkcd.com/1882/`.

(*a*) Modify the code in `hw04-shdr.cc` so that when `opt_hw04_dot` is true a gold circle is shown centered on the face of each dodecahedron. Do this by changing the color in the fragment shader,

not by emitting new primitives. The radius, r, should be exactly $\frac{1}{2}$ the pentagon inscribed circle radius. That is, the smallest distance from the circle circumference to a pentagon edge should be r.

☐ Show the circle centered on each face.

☐ Do not emit new primitives, instead ☐ change the color in the fragment shader.

☐ Add new geometry shader output(s) and fragment shader inputs(s).

☐ The circle radius should be half the pentagon inscribed radius for each dodecahedron, regardless of size.

☐ For help in figuring out the radius look at the code computing `ctr_pentagon_e`.

(*b*) Modify the code in `hw04-shdr.cc` so that when `opt_hw04_tex` is true the texture bound to `tex_unit_0` is mapped to each dodecahedron face. The texture should be sized so that its corners touch the pentagon inscribed circle. Show exactly one copy on each face.

In the unmodified code a texel is retrieved using a constant coordinate and nothing is done with the texel. Use this as a starting point:

```
void fs_main()
{
  vec2 tc = vec2(0.5,0.5);
  vec4 texel = texture(tex_unit_0,tc);
```

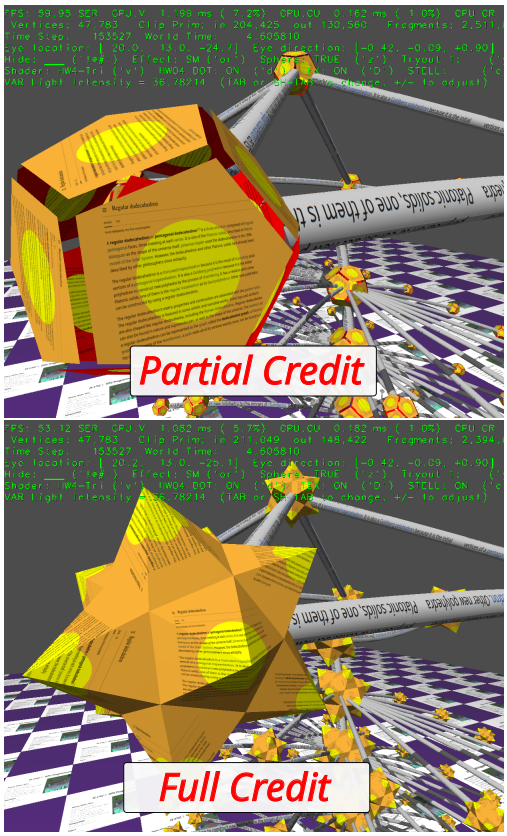
☐ Show the texture on a dodecahedron face.

☐ Show one copy per face, the edges should touch the inscribed circle.

☐ The image should not be distorted (when stellation is off).

Problem 2: Solve part (a) below for partial credit, part (b) for full credit. If part (b) is solved there is no need to solve part (a). If you solve both and just can't bring yourself to leave your work on part(a) dormant use `opt_tryout3` to switch between the two.

(*a*) *For partial credit solve this part instead of part b:* Modify the code in `gs_main` so that when `opt_hw04_stel` is true the pentagon is shrunk by about 10%. That is, each of the pentagon vertices moves 10% toward the pentagon center. Note that the coordinates of the pentagon center are already computed in the geometry shader. Also recall that the vertices provided to the geometry shader are pentagon vertices (none of them are inside the pentagon). See the screenshot to the upper right.



☐ Render a shrunken pentagon.

☐ This part can be skipped of the subpart below is solved.

(*b*) *Solve instead of (or in addition to part a), for Full Credit:* Modify the code in `gs_main` so that when `opt_hw04_stel` is `true` stellated dodecahedra with local-space height `opt_hw04_height_stel` are rendered. A stellated dodecahedron is obtained from a regular dodecahedron by putting a five-sided pyramid on each face. See the screenshot to the lower right.

Note that the coordinates of the pentagon center are already computed in the geometry shader. Another handy variable is `is_pentagon_edge`. It is `true` when coordinates `pt_i_e` and `pt_n_e` form a pentagon edge. (One or two edges of each triangle seen by the geometry shader are **not** pentagon edges.) To solve this the geometry shader may have to emit more than one triangle.

- ☐ Modify `gs_main` and surrounding code to emit a stellated dodecahedron.
- ☐ Update `max_vertices` if necessary.
- ☐ Make sure that `normal_e` written by the geometry shader is correct.
- ☐ The edges between the faces of the pyramid should be sharp (not blurry).