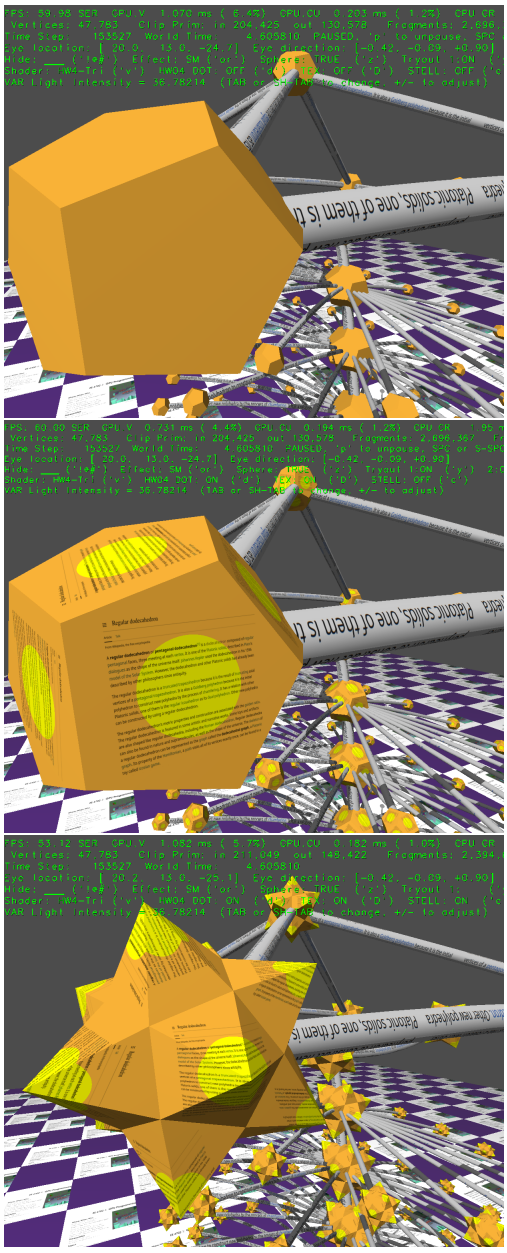
LSU EE 4702-1 **Homework 5** **Due: 30 November 2025**

Formatted 19:04, 22 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/hw05.cc.html` *and* `https://www.ece.lsu.edu/koppel/gpup/2025/hw05-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. In fact, it should look just like the scene from Homework 4. See the screenshot to the upper right. The code in this assignment is based on the solution to Homework 4, and the UI works the same way, except that there are more shaders. Solving this assignment correctly will not change what is show other than the performance data shown in the green text. The three screenshots to the right are being shown so avoid the need to find three different images.



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 `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.

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_hw05_dot`. When correctly solved the gold dot will be shown only when `opt_hw05_dot` is `true`. Pressing `D` will toggle the value of Boolean `opt_hw05_tex`. When correctly solved textures will be shown on the dodecahedron faces only when `opt_hw05_tex` is `true`. Pressing `c` will toggle the value of Boolean variable `opt_hw05_stel`. When correctly solved the dodecahedra faces will be stellated when `opt_hw05_stel` is `true`. Also available for stellation is `float` variable `opt_hw05_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_hw05_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 five different sets of shaders, labeled `PLAIN`, `HW5-Tri`, `HW5-Pts-Cln`, `HW5-Pts-P1`, and `HW5-Inst-P2`. Pressing `v` cycles between these shaders. Shader set `PLAIN` are the ones used in the links code, they show the links attached to balls. The other shader sets render dodecahedra instead of balls. Shaders for all of these are in file `hw05-shdrs.cc`. Shader sets `HW5-Pts-P1` and `HW5-Inst-P2` are to be modified as part of the solution to this assignment. The shaders are named `vs_main_pts_prob_1`, `gs_main_pts_prob_1`, `vs_main_inst_prob_2`, and `gs_main_inst_prob_2`. The other shaders are for reference and for light experimentation.

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.

Code Overview

For this assignment both shader code, in file `hw05-shader.cc`, and the host (CPU) code in file `hw05.cc` are to be modified.

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 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 prepares all of the dodecahedron shaders, so it's a bit busy. It starts by doing any required pipeline setup, populates buffer sets to show each ball as a dodecahedron, and then *records* a *draw* with the buffer set providing vertex shader inputs to the pipeline.

The routine starts by initializing Vulkan storage buffers. These only need to be initialized once. You will only need to modify this code if you want to add your own storage buffers (though you should not need to):

```
if ( !buf_hw05_color )
  {
    buf_dod_coors_l.init( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer );
    buf_hw05_color.init( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer );
    buf_hw05_matrix.init( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer );
    buf_hw05_vector.init( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer );
    buf_hw05_coor.init( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer );
  }
```

Each storage buffer holds an array of items. Some or all of the storage buffers above will be needed in this assignment. If you'd like to add your own storage buffers look for occurrences of one of the buffers, say `buf_hw05_coor`, to see what needs to be done.

Next the pipelines are initialized. As with storage buffers, this only happens once for run. Consider the initialization of the pipeline for shader set `HW5-Pts-Cln`:

```
if ( !pipe_doda_pts_clean )
  {
    pipe_doda_pts_clean
      .init( vh.qs )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .ds_use( sampler, texid_hw )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common );

    pipe_doda_pts_clean
      .ds_storage_follow
      ( "BIND_HW5_COLOR", buf_hw05_color, shader_stages_all );

    pipe_doda_pts_clean
      .shader_inputs_info_set<pCoor, pVect, pMatrix, int>()
      .shader_code_set
      ("hw05-shdr.cc",
       "vs_main_pts_clean();",
       "gs_main_pts_clean();",
       "fs_main();", "#define HW5_PTS_CLEAN\n" )
      .topology_set( vk::PrimitiveTopology::ePointList );

    pipe_doda_pts_clean.create();
  }
```

The initialization starts with the call `pipe_doda_pts_clean.init( vh.qs )` and then continues with member functions (operating on the returned reference to `pipe_doda_pts_clean`) specifying resources that will be accessed by the shaders in the pipeline. Many of these member function names start with `ds_`, the "ds" is for *descriptor set*. The first group above specifies lighting information, transformations, a texture/sampler pair, and a uniform. All of the pipelines used for this assignment access these same resources. The `puni_light_curr` call provides lighting information. The solution to this assignment (Fall 2025 Homework 5) 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 `hw05-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. 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.

Next the pipeline is set to use storage buffer `buf_hw05_color`. The string `BIND_HW5_COLOR` is defined as a macro in the shader code and its value is the buffer binding point. The third argument, `shader_stages_all`, specifies which shader stages can access the buffer.

The next group specifies the shader code. It consists of three calls, the first, `shader_inputs_info_set`, specifies the data types that will be used as inputs to the vertex shader, in this case `pCoor`, `pVect`, `pMatrix` (oh my!), and `int`. The buffer set used with the draw command, in this case `pipe_doda_pts_clean.record_draw( cb, bset_doda_pts_clean )`, must have buffers with those types populated. More on that later.

The call `shader_code_set` specifies: the name of the file holding the shader code, commands used to call the vertex shader, geometry shader, and fragment shader. The last argument specifies code to put at the top of the file when running this particular shader. The last call in the group specifies the topology used to group vertices at the input to the geometry shader. The geometry shader code must declare a consistent input topology. In this particular case the geometry shader must declare points as its inputs, which it does.

The last call is `pipe_doda_pts_clean.create()` which creates the pipeline. In the excerpt above the calls were broken into four groups to make it easier to describe them. In other cases there's just one group.

To solve both Problems 1 and 2 their respective pipelines will need to be modified. The modifications include adding or removing storage buffers, changing the inputs to the vertex shaders (by modifying `shader_inputs_info_set`), and perhaps changing the geometry shader input topology.

The code that follows sets up other dodecahedron pipelines. After that is the code that, if necessary, computes dodecahedron coordinates in local space. Those coordinates will be placed in Vulkan storage buffer `buf_dod_coors_l`. If the buffer is empty the coordinates are computed and sent to the GPU:

```
if ( buf_dod_coors_l.size() == 0 ) {
    pCoor center(0,0,0);
    pVect ax(1,0,0), ay(0,1,0), az(0,0,1);
    float ri = 1;
    const float len_edge = ri / sqrt( sqrt(5) * pow(numbers::phi,5) / 20 );
    const float rc_pentagon = len_edge / ( 2 * sinf(numbers::pi/5) );
    pCoor center_pentagon_0 = center + ri * az;
    float theta_dihedral = 2 * atanf( numbers::phi );
    const float d_theta = 2 * numbers::pi / 5;
    for ( int i=0; i<5; i++ ) {
        const float theta = d_theta * i;
```

```
        pVect dir = cosf(theta) * ax + sinf(theta) * ay;
        pCoor pos = center_pentagon_0 + rc_pentagon * dir;
        buf_dod_coors_l << pos;  }
      for ( int i=0; i<5; i++ ) {
        pCoor p0 = buf_dod_coors_l[i], p1 = buf_dod_coors_l[(i+1)%5];
        pMatrix_Translate to_p0( p0 );
        pMatrix_Rotation rot2( pNorm(p0,p1), -theta_dihedral );
        pMatrix_Translate to_Origin( pVect( p0, pCoor(0,0,0) ) );
        pMatrix m = to_p0 * rot2 * to_Origin;
        for ( int j=4; j>=0; j-- ) buf_dod_coors_l << m * buf_dod_coors_l[j]; }

      pMatrix_Rotation rot3(ay,numbers::pi);
      const int n_vtx_demi = buf_dod_coors_l.size();
      for ( int i=0; i<n_vtx_demi; i++ ) buf_dod_coors_l << rot3 * buf_dod_coors_l[i];
      buf_dod_coors_l.to_dev();
    }
```

After preparing the local coordinates the code sets up buffer sets and then does a draw using the chosen shader set (and pipeline) in the `switch( opt_shader )` construct. The switch uses enumeration constants to identify them, such as `SO_HW5_Triangles` for `HW5-Tri`.

Each shader set operates differently, though they will all result in the same images being shown. The shaders differ in how they get their data and in how much computation they do.

Shader set `HW5-Tri` (`SO_HW5_Triangles`) is similar to the Homework 4 shader. The input to a vertex shader invocation consists of a dodecahedron vertex local-space coordinate, a local-to-object transformation matrix (oh my!), and a color. Each face is rendered using three triangles, and so for each dodecahedron $3 \times 3 \times 12 = 108$ vertices are entered. Each vertex of a particular dodecahedron is given the same color and transformation matrix. How wasteful!

```
  case SO_HW5_Triangles:
    bset_doda.reset( pipe_doda );
    for ( Ball* b: balls ) {
        pMatrix rot(b->omatrix);
        pMatrix_Translate tra(b->position);
        pMatrix_Scale sca(b->radius);
        pMatrix m = tra * sca * rot;
        for ( size_t i=0; i<buf_dod_coors_l.size(); i+=5 )
          // Render the pentagon by emitting three triangles, all
          // share pentagon vertex 0 (index i).
          //
          for ( int j=1; j<4; j++ )
            bset_doda
             << buf_dod_coors_l[ i ] << buf_dod_coors_l[ i + j ] << buf_dod_coors_l[ i + j +
1 ];

        for ( int i=0; i<3*3*12; i++ ) bset_doda << b->color << m;
      }
    bset_doda.to_dev();
    pipe_doda.record_draw( cb, bset_doda );
```

The code above populates the buffer set, sends it to the device, and records a draw.

The second shader, `HW5-Pts-Cln` (`SO_HW5_Points_Clean`) uses a points topology instead of a triangle list. Each invocation of the geometry shader will render one pentagon (using the data from

the geometry one element shader input). Here the vertex shader input consists of a transformation matrix, the local-space coordinate of one vertex on a dodecahedron face, the face normal, and a *color index*. This pipeline, `pipe_doda_pts_clean`, binds the `buf_hw05_color` storage buffer. The code that sets up the buffer set also writes colors to the storage buffer. Let n denote the number of dodecahedra to be rendered. The number of elements expected in the buffer set is $12n$ (one for each face). But since one color is used for an entire dodecahedron, the number of elements in `buf_hw05_color` is just n. The color index written to the buffer set is used by the shader code to access the color from `buf_hw05_color`. (The color is accessed in the geometry shader using `vec4 color = buf_hw05_color[In[0].color_idx];`.) Here is the code preparing the buffer set and storage buffer and the code recording the draw:

```
case SO_HW5_Points_Clean:
  bset_doda_pts_clean.reset( pipe_doda_pts_clean );
  buf_hw05_color.clear();
  for ( Ball* b: balls ) {
      pMatrix rot(b->omatrix);
      pMatrix_Translate tra(b->position);
      pMatrix_Scale sca(b->radius);
      pMatrix m = tra * sca * rot;
      const int color_idx = buf_hw05_color.size();
      buf_hw05_color << b->color;

      for ( size_t i=0; i<buf_dod_coors_l.size(); i+=5 ) {
          pNorm n( cross( buf_dod_coors_l[i+2],
                          buf_dod_coors_l[i+1],
                          buf_dod_coors_l[i  ] ) );
          bset_doda_pts_clean << m << buf_dod_coors_l[i] << n << color_idx;
        }}

  bset_doda_pts_clean.to_dev();
  buf_hw05_color.to_dev();
  pipe_doda_pts_clean.record_draw( cb, bset_doda_pts_clean );
```

Note that the geometry shader must render the pentagon using just one pentagon vertex, a normal, and a transformation matrix. That's possible because the coordinate is in local space, in which the dodecahedron center is at the origin. See the geometry shader `gs_main_pts_clean()` to see how that's done.

Alert students at this point may have realized at this point that `HW5-Pts-Cln` sends much less data from the CPU to the GPU, but that even less can be sent. One improvement is that the number of Vulkan vertices per dodecahedron has dropped from $3 \times 3 \times 12 = 108$ to just 12. And instead of sending a color, a color index is sent, which is one quarter the size. But, **but**, an entire transformation matrix is sent for each face even though it will be the same for each of the 12 faces. If a color index can be sent in place of a color, why not do the same for the transformation matrix? Yes, we could use the color index (which is really just a dodecahedron index) to retrieve the transformation matrix from a transformation matrix storage buffer. But Problem 1 goes further than that: not only will the transformation matrix be removed from the `HW5-Pts-P1` vertex shader input, but the vertex shader will have zero inputs. Nothing. Except for vertex shader built-in input `gl_VertexIndex` and geometry shader built-in input `gl_PrimitiveIDIn`. These provide a consecutive numbering of the shader inputs, starting at zero.

The third shader, `HW5-Pts-P1`, (`SO_HW5_Points_Prob_1`, `pipe_doda_pts_prob_1`) is for Problem 1. As described above, this is like `HW5-Pts-Cln`, except that the vertex shader has zero inputs

(and that should not be changed). Because it has zero inputs no buffer set is needed. The vertex shaders use `gl_VertexIndex` and geometry shaders use `gl_PrimitiveIDIn` to determine which item they are working on and retrieve any data they need from storage buffers. Because there is no buffer set, the draw command must be told how many times to invoke the vertex shader. In the unmodified assignment that's hardcoded to 1. See the code below, that sets up the color storage buffer and for reference has defined-out code for sending coordinates.

```
  case SO_HW5_Points_Prob_1:
    buf_hw05_color.clear();

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

        int dod_idx = buf_hw05_color.size();
        buf_hw05_color << b->color;

        // Zero inputs, so defined-out code below not needed.
#if 0
        for ( size_t i=0; i<buf_dod_coors_l.size(); i+=5 ) {
            pNorm n( cross( buf_dod_coors_l[i+2], buf_dod_coors_l[i+1],
                            buf_dod_coors_l[i  ] ) );
            bset_doda_pts_prob_1 << m << dod_idx << buf_dod_coors_l[i] << n;
          }
#endif
      }
     const int n_vertices = 1; // This number is way too low.
     buf_hw05_color.to_dev();
     pipe_doda_pts_prob_1.record_draw( cb, n_vertices );
```

The last shader set `HW5-Inst-P2` (`SO_HW5_Inst_Prob_2`, `pipe_doda_inst_prob_2`) takes a different approach to reducing the amount of data sent from the CPU to the GPU. This pipeline is used for 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 dodecahedra 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. When correctly solved the `HW5-Inst-P2` shaders will draw one dodahedron for each instance. The vertices in the buffer set will describe just one dodecahedron. In an instanced draw the vertex shader uses `gl_InstanceIndex` to determine the instance number. An 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 dodecahedron location in local space and of course that doesn't change. Instead, any per-instance data is placed in storage buffers. The

existing code just uses storage buffers for colors. 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. The existing code renders just one instance and that one instance renders all of the dodecahedra. That will need to be changed.

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`.

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: Modify the code in `hw05.cc` and `hw05-shdr.cc` so that when shader set `HW5-Pts-P1` is used the dodecahedra are rendered using a point list input topology with zero vertex shader inputs. The pipeline is already set for zero vertex shader inputs and a point topology. But other changes need to be made. To help you get started the code in `gs\_main\_pts\_prob\_1` has the "clean" shader code, but that code won't work because it lacks the vertex shader inputs.

- ☐ Modify the code setting up `pipe_dota_pts_prob_1` so that any additional storage buffers are bound to the pipeline. ☐ Don't overlook `buf_dod_coors_1`, which the already code prepares and sends to the GPU.
- ☐ Modify the code in the `SO_HW5_Points_Prob_1` switch case to ☐ populate and send any additional buffer objects and to ☐ call the `draw` with `n_vertices` set to the appropriate value.
- ☐ Modify the `gs_main_pts_prob_1` shader (and surrounding code) in `hw05-shdr.cc` to render the dodecahedron faces, using `gl_PrimitiveIDIn` to get the vertex number. (Because the input topology is points and there are no vertex shader inputs there's no need to modify the vertex shader.)
- ☐ Unlike the "clean" points shader, avoid computing the pentagon vertices, instead read them.

Problem 2: Modify the code in `hw05.cc` and `hw05-shdr.cc` so that when shader set `HW5-Inst-P2` is used the dodecahedra are rendered using an instanced draw. One big difference in an instanced draw is that the buffer set only need be sent once. So there is less of a need to keep it small. In this problem use the buffer set for triangle coordinates and texture coordinates, and anything else that's needed. Also, to get the stellation effect, insert coordinates for five triangles per face (one vertex is at the pentagon center, the other two form a pentagon edge). Have the geometry shader change the location of the center coordinate to get the stellation effect.

- ☐ Bind any additional storage buffers that are needed to `pipe_doda_inst_prob_2`. Don't bind more than are needed.
- ☐ Use the buffer set, `bset_doda_inst_prob_2`, for dodecahedron local coordinates and ☐ texture coordinates, and anything else that's needed.
- ☐ Write the buffer set once, not every frame.
- ☐ Write five triangles per face, with one vertex of each triangle in the pentagon center.
- ☐ Modify the shaders to use the instance number to get the appropriate data.
- ☐ The shaders should implement the stellation effect.