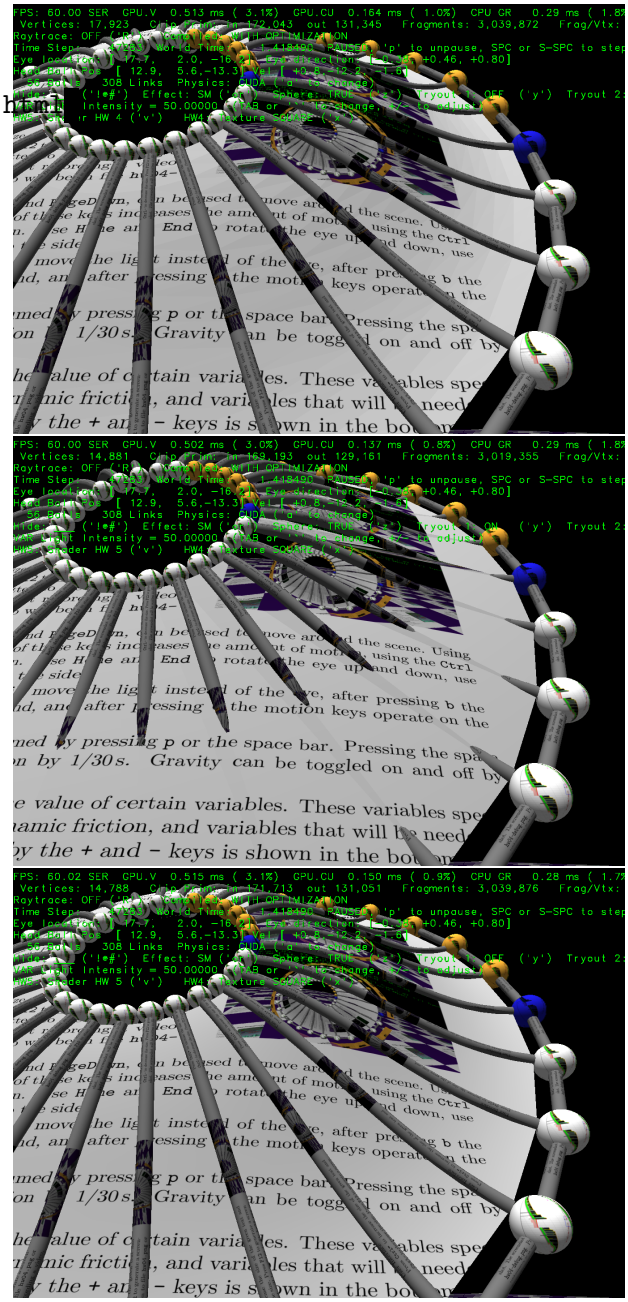
LSU EE 4702-1 **Homework 5** Solution **Due: 22 November 2023**

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/2023/hw05.cc.html` *(CPU code) and* `https://www.ece.lsu.edu/koppel/gpup/2023/hw05-shdr.cc.html` *(shader code). The solution code is at* `https://www.ece.lsu.edu/koppel/gpup/2023/hw05-sol.cc.html` *(CPU code) and* `https://www.ece.lsu.edu/koppel/gpup/2023/hw05-shdr-sol.cc.html` *(shader code).*

Problem 0: If not already done, follow the instructions on `https://www.ece.lsu.edu/koppel/gpup/proc.html` for account setup and programming homework work flow. Compile and run the homework code unmodified. The code is based on the solution to Homework 4.



User Interface

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.ogg` or `hw05-debug-1.ogg`.

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. These variables specify things such as the gravitational acceleration, dynamic friction, and variables that will 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

The code can display several scenes, numbered 1 through 5. For this assignment use scene 4 (which is how the code starts). Press 1 to select scene 1 (the silly tree), 2 for scene 2 (I'm going to ask GPT-4 to name it when it can read images), 3 for scene 3 (a crude corona virus particle), 4 for this homework assignment (the wheel), 5 for a top (a spinning child's toy). Scene 1 is initialized by the code in World::ball_setup_1, etc. Parts of the solution to this assignment should be put in World::ball_setup_4.

The scene for this assignment, 4, initializes randomly each time it is reset. (That is to avoid solutions that accidentally only work for a special case of positioning, etc.)

The code for Scene 4 can be run with two shaders, called HW 4 and HW 5. The shader in use is shown on the bottom line of green text and can be switched by pressing v. The HW 4 shader is based on the solution to Homework 4. The HW 5 shader is to be modified in this assignment. Initially it shows a cruder version of the wheel, in which the surface does closely follow the shape of spokes when the spokes are bent. The code setting up the HW 4 shader is in hw05.cc:World::render_hw04 and the shaders themselves are in hw04-shdr.cc. The code setting up the HW 5 shader is in hw05.cc:World::render_hw05 and the shaders themselves are in hw05-shdr.cc.

The projection of textures on to the surface can be changed by pressing x. There are three possible projections NONE, meaning no textures, SQUARE, which shows one copy of the texture mapped over the entire wheel, and CIRCLE, which shows the texture wrapped around the wheel, perhaps requiring several copies of the texture. Pressing x cycles between these projections. Variable opt_texture is set to the current projection, which can be TX_None, TX_Orig, TX_Square, or TX_Circle. (In hw05.cc those are enumeration constants in hw05-shdr.cc those are integer constants.)

Display of Performance-Related Data

The top green text line shows performance in various ways. The number to the right of FPS shows the frame rate, the number of frames completed per second. On some displays 60 is the target frame rate and anything significantly lower than that indicates mediocre performance. Next, the green text shows whether frames are being prepared one at a time (serially), SER, or the steps in preparing a frame are being overlapped, OVR. In overlap mode commands for one frame are recorded while commands for a prior frame may be executing. For assignments early in the semester the mode will be kept at SER.

GPU.V shows how long the GPU spends updating the frame buffer (per frame), GPU.CU shows the execution of CUDA code per frame. CUDA code is physics in some assignments, but not this one and so the time should be shown as ---. On some of the lab computers the computational accelerator GPU is different than the one performing graphics. CPU GR is the amount of time that the CPU spends recording Vulkan graphics commands (or whatever it does in the callback installed by vh.cbs_cmd_record.push_back). CPU PH is the amount of time that the CPU spends on physics or whatever it does in the callback installed by the call to vh.display_cb_set.

For this assignment rendering is done by ray tracing. (Other assignments will use rasterization.) For rasterization the second line, the one starting with Vertices, shows the number of items being sent down the rendering pipeline per frame. Clip Prim shows the number of primitives before clipping (in) and after clipping (out). The next line indicates whether the code was compiled with optimization. Use the version without optimization for debugging and the version with optimization

for performance measurements.

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.

To help you debug your code and experiment in one way or another, the user interface lets you change variables. In most assignments these include the variables `opt_tryout1`, `opt_tryout2`, `opt_tryout3`, and `opt_tryoutf`, available in CPU and shader code. You can use these variables in your code (for example, `if ( opt_tryout1 ) { x += 5; }`) to help debug, to help familiarize yourself with how the code works, or to experiment with new ideas. Keys `y`, `Y`, and `Z` toggle the value of host Boolean variables `opt_tryout1`, `opt_tryout2`, and `opt_tryout3`. The user interface can also be used to modify host floating-point variable `opt_tryoutf` using the `Tab`, `+`, and `-` keys, see the previous section.

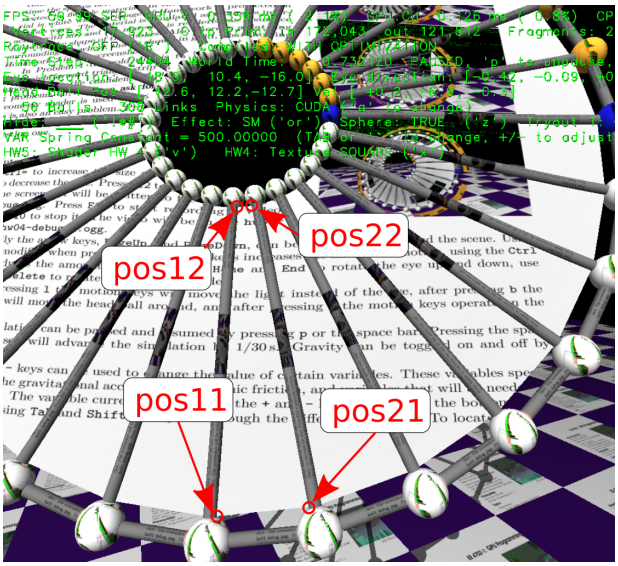
Resources

A good reference for C++ is `https://en.cppreference.com/w/`. Solutions to the shader programming problems may (will) require the use of library functions. See Chapter 8 of The OpenGL Shading Language Version 4.6 specification. Also see 2014 Homework 4-6 (especially 6), the spiral, to see examples of how the pipeline input topology can be chosen to provide data needed by the geometry shader, even though the topology class (point, line, triangle) does not geometrically correspond to what is being drawn.

The bottom half of this page intentionally left blank (other than this message).

Code Overview

The `render_hw04` routine draws the surface of the wheel using a pipeline set up for a triangle strip. The outer loop in that routine, extracted below, iterates over spokes. In each iteration it uses information from two adjacent spokes, `spoke1` and `spoke2`, to construct the strip. Each spoke has an index, such as `li1`, and that index can be used to get information about the spoke (or link). Array `lis_pos1` holds the coordinate of one spoke endpoint and `lis_pos2` holds the coordinate of the other endpoint. In the code below the two coordinates from each of the two spokes are extracted and placed in variables `pos11`, `pos12`, `pos21`, and `pos22`. The diagram to the right shows the relationship of the coordinates to the spokes and wheel.




```
const size_t n_spokes = hw05_stuff.spokes.size();
for ( size_t i = 0; i < n_spokes; i++ )
  {
    size_t i2 = (i+1) % n_spokes;
    Link *spoke1 = hw05_stuff.spokes[i], *spoke2 = hw05_stuff.spokes[i2];
    const int link_idx1 = spoke1->idx;
    const int link_idx2 = spoke2->idx;
    pCoor pos11 = lis_pos1[link_idx1];
    pCoor pos12 = lis_pos2[link_idx1];
    pCoor pos21 = lis_pos1[link_idx2];
    pCoor pos22 = lis_pos2[link_idx2]; }
```

It would be possible to use these four coordinates to draw two triangles, but those triangles would not match the shape of the spokes, which are cubic curves. An inner loop, shown below, finds points between the endpoints using routine `bez`. Point `pos1t` is between `pos11` and `pos12`, following the curved path defined by the endpoints and vectors `v11` and `v12`. It is points `pos1t` and `pos2t` that are inserted into the buffer set for use in drawing the surface. In addition to the coordinates, texture coordinates are also inserted. The `mix` routine blends the endpoint texture coordinates (not shown in the exceprt) using parameter `t`.

```
    pVect v11 = lis_v1[link_idx1], v12 = lis_v2[link_idx1];
    pVect v21 = lis_v1[link_idx2], v22 = lis_v2[link_idx2];
    const float delta_t = 1.0/opt_segments;
    for ( int s=0; s<=opt_segments; s++ ) {
        const float t = s * delta_t;
        pCoor pos1t = bez(pos11,pos12,v11,v12,t);
        pCoor pos2t = bez(pos21,pos22,v21,v22,t);
        bset_hw04 << pos1t << mix(tex11,tex12,t)
                  << pos2t << mix(tex21,tex22,t);
   }
```

The code in `render_hw05` prepares coordinates for pipeline `pipe_hw05`. The pipe has two inputs, an `int` and a texture coordinate. The pipeline also binds four arrays, `lis_pos1`, `lis_pos2`,

lis_v1, and lis_v2. Theses are the arrays holding link data used above, and they can be accessed in shader code using C array syntax.

The main loop in render_hw05 inserts just two vertices (there is no s loop) in the pipeline for each spoke (and does the first spoke again at the end):

```
  const size_t n_spokes = hw05_stuff.spokes.size();
  for ( size_t i = 0; i <= n_spokes; i++ ) {
      Link* const spoke1 = hw05_stuff.spokes[i%n_spokes];
      const int link_idx = spoke1->idx;
      pTCoor tex11 = spoke1->ball1->tca[opt_texture];
      pTCoor tex12 = spoke1->ball2->tca[opt_texture];

      bset_hw05 << link_idx << tex11;
      bset_hw05 << link_idx << tex12;
    }
```

Each vertex consists of an integer, link_idx, and a texture coordinate. Note that the same index is used twice. The integer is used to index the lis arrays. A placeholder vertex shader uses the index, named in_link_idx in the shader code, to extract a position:

```
void vs_hw05() {  // Vertex Shader in hw05-shdr.cc
  vec4 p1 = pos1[in_link_idx];  // Outer ring.
  vec4 p2 = pos2[in_link_idx];  // Inner ring.
  vec4 p = ( gl_VertexIndex & 1 ) == 0 ? p1 : p2;
  Out.vertex_o = p;
  Out.vertex_e = ut.eye_from_object * p;
  Out.tex_coor = in_tex_coor;
}
```

The code above actually finds two coordinates. It uses p1 for even-numbered vertices, and p2 for odd-numbered vertices. Built-in variable gl_VertexIndex provides the vertex number. (Zero for the first vertex in a draw.)

The pipeline is set to group vertices for a triangle strip topology, so the input to the geometry shader is a triangle. The placeholder geometry shader computes the triangle normal and a clip-space coordinate, and emits the triangle:

```
void gs_hw05() {
  vec3 normal_o =
    cross( In[1].vertex_o.xyz - In[0].vertex_o.xyz,
           In[2].vertex_o.xyz - In[0].vertex_o.xyz );
  vec3 normal_e = mat3( ut.eye_from_object ) * normal_o;

  for ( int i=0; i<3; i++ ) {
      gl_Position = ut.clip_from_object * In[i].vertex_o;
      Out.normal_e = normal_e;
      Out.vertex_e = In[i].vertex_e;
      Out.tex_coor = In[i].tex_coor;
      EmitVertex();
    }
  EndPrimitive();
}
```

To help in solving the assignment the shader file has a bez routine that can be used for finding points along the spoke. It also has a routine, bez_dt, that computes the derivative of the curve

between the two points (with respect to the parametric parameter, `t`). Routine `bez_dt` can be used for finding surface normals.

Problem 1: Modify code in `hw05.cc:World::render_05` and the shaders in `hw05-shdr.cc` so that the geometry shader computes the triangles between two spokes. Each invocation of a geometry shader should do the same work as the `s` loop from `render_hw04` described above.
Here is what needs to be done:

Modify Pipeline Input Topology
The geometry shader needs information on two adjacent spokes (links) so that it can find the four points. That's not easily possible using a triangle strip topology. Modify the topology used by `pipe_hw05` to better provide this information. This can be done by modifying the argument to `.topology_set` called where the pipeline is set up in `render_hw05`. Consider the approach used in 2014 Homework 6, in which a spiral was constructed using a line list.

Modify Pipeline Inputs
In the unmodified assignment `pipe_hw05` is set for two inputs, an int and a texture coordinate. Those are set by the `.shader_inputs_info_set` call by the types between the angle brackets. A limited number of types are accepted and they can't be repeated. The valid types are `int`, `ivec2`, `ivec4`, `pCoor`, `pColor`, and `pVect`. (This is a limitation of the course library, not Vulkan.) An input type does not have to be used for the purpose implied by its name. So a `pColor`, for example, can be used to hold four floats used for any purpose. Note that `ivec2` is a 2-element integer vector.

Modify Shader Inputs
For any change made to pipeline inputs (by modifying `.shader_inputs_info_set`) a corresponding change must be made in the shader code. For your convenience some inputs are pre-defined, guarded by ifdefs, such as `in_int4` for an `ivec4` type.

Modify the Shader Interface Blocks
A correct solution to this assignment will require changing the information sent from the vertex to the geometry shader. Modify the interface blocks for that. (These are the structure-like constructs with tags like `Data_to_GS`.) Remember that the block describing the output of the vertex shader must match the block at the input to the geometry shader.

Modify the Geometry Shader Input and Output Layouts
The geometry shader output topology should remain a triangle strip, but the `max_vertices` setting should change. Compute `max_vertices` from `opt_segments`, which is available as a constant in the shader file (trust me on that). The input layout to the geometry shader must match what was set on `pipe_hw05`.

Modify the Geometry Shader to Emit the Triangles
Modify the geometry shader so that it computes and emits the triangles between the spokes. There should be something like the `s` loop, but adapted for a geometry shader.

Compute the Correct Surface Normal Based on the Shape of the Curve
Use `bez_dt` to help get the normal. Note that `bez_dt` does not provide the surface normal, instead it is a vector in a direction of the curve from `posx1` to `posx2` at point `t`.

The solution code is in the repo, and is on the Web at
`https://www.ece.lsu.edu/koppel/gpup/2023/hw05-sol.cc.html` (CPU code) and
`https://www.ece.lsu.edu/koppel/gpup/2023/hw05-shdr-sol.cc.html` (shader code). In the discussion below solution code fragments are shown in various states of development, in some cases a code fragment is first shown in some tentative or incomplete way, and then again with improvements. After this discussion more complete versions of the code are shown.

The description of the solution will start with the geometry shader. Each invocation of the geometry shader must emit the `2*opt_segments` triangles emitted by an `s` loop iteration (described in the Code Overview section above).

Consider the simplified CPU `s` loop from the Code Overview section:

```
pCoor pos11 = lis_pos1[link_idx1];
pCoor pos12 = lis_pos2[link_idx1];
pCoor pos21 = lis_pos1[link_idx2];
pCoor pos22 = lis_pos2[link_idx2];
pVect v11 = lis_v1[link_idx1], v12 = lis_v2[link_idx1];
pVect v21 = lis_v1[link_idx2], v22 = lis_v2[link_idx2];
const float delta_t = 1.0/opt_segments;
for ( int s=0; s<=opt_segments; s++ ) {
    const float t = s * delta_t;
    pCoor pos1t = bez(pos11,pos12,v11,v12,t);
    pCoor pos2t = bez(pos21,pos22,v21,v22,t);
    bset_hw04 << pos1t << mix(tex11,tex12,t)
              << pos2t << mix(tex21,tex22,t);
 }
```

To adapt this to the geometry shader we need to make changes to data types and such, and to also plan for the needed input data.

As stated in the Code Overview, arrays `lis_pos1`, `lis_pos2`, `lis_v1`, and `lis_v2` used by the CPU code are also available to the shader (as Vulkan storage buffers). Their names are shorter, just `pos1`, `pos2`, etc. As stated in the Code Overview, the element data types for all those arrays are `vec4` (not `pCoor` and `pVect`). So the statement `pCoor pos11 = lis_pos1[link_idx1]` should be changed to `vec4 pos11 = pos1[link_idx1]`.

Also stated in the Code Overview, is that `bez` is provided in the shader code file and `mix` is a built-in GLSL function. The qualifier `const` has a different meaning in GLSL than it does in C++, so it needs to be removed. Making all those changes we get:

```
// Partially translated s loop. Still doesn't work.
vec4 pos11 = pos1[link_idx1];
vec4 pos12 = pos2[link_idx1];
vec4 pos21 = pos1[link_idx2];
vec4 pos22 = pos2[link_idx2];
vec4 v11 = v1[link_idx1], v12 = v2[link_idx1];
vec4 v21 = v1[link_idx2], v22 = v2[link_idx2];
float delta_t = 1.0/opt_segments;
for ( int s=0; s<=opt_segments; s++ ) {
    float t = s * delta_t;
    vec4 pos1t = bez(pos11,pos12,v11,v12,t);
    vec4 pos2t = bez(pos21,pos22,v21,v22,t);
    // WARNING: The lines below have no meaning in GLSL.
    bset_hw04 << pos1t << mix(tex11,tex12,t)
              << pos2t << mix(tex21,tex22,t);
 }
```

Here's what remains to be done with the code above: the buffer set insertion line, `bset_hw04 <<`, has no meaning in shader code, it must be replaced with writes to geometry shader output variables. Also, we need to provide values for `link_idx1`, `link_idx2`, `tex11`, `tex12`, `tex21`, and `tex22`. These can come from geometry shader inputs.

To start, lets replace the buffer set insertion line. That line is writing two vertex coordinates, `pos1t` and `pos2t`, which are supposed to be grouped using a triangle strip topology. The output of the geometry shader is already set for a triangle strip (that's the only kind of triangles available). So, we need to just emit the two vertices.

To emit a vertex from a geometry shader we first write the geometry shader output variables and then call `EmitVertex()`. The geometry shader has built-in output `gl_Position`, which must be written with the clip-space coordinate of the vertex. Our shader code has declared three other outputs (which are expected by our fragment shader which remember is not part of this assignment):

```
layout ( location = 0 ) out Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
} Out;
```

So, for each vertex we need to write these four output variables with appropriate values. The eye- and clip-space coordinates are easy to compute from the object-space coordinates in `pos1t` and `pos2t`. The only part that requires thought is the normal, which we'll get to later. Here is code emitting the two vertices:

```
vec4 pos11 = pos1[link_idx1], pos12 = pos2[link_idx1];
vec4 pos21 = pos1[link_idx2], pos22 = pos2[link_idx2];
vec4 v11 = v1[link_idx1], v12 = v2[link_idx1];
vec4 v21 = v1[link_idx2], v22 = v2[link_idx2];
float delta_t = 1.0/opt_segments;
for ( int s=0; s<=opt_segments; s++ ) {
    float t = s * delta_t;
    vec4 pos1t = bez(pos11,pos12,v11,v12,t);
    vec4 pos2t = bez(pos21,pos22,v21,v22,t);

    // Emit Vertex for link_idx1
    gl_Position = ut.clip_from_object * pos1t;
    Out.vertex_e = ut.eye_from_object * pos1t;
    Out.tex_coor = mix(tex11,tex12,t);
    // Out.normal_e = // Save for later.
    EmitVertex();

    // Emit Vertex for link_idx2
    gl_Position = ut.clip_from_object * pos2t;
    Out.vertex_e = ut.eye_from_object * pos2t;
    Out.tex_coor = mix(tex21,tex22,t);
    // Out.normal_e = // Save for later.
    EmitVertex();
 }
 EndPrimitive();
```

For the code above to work we need to remember to change the `max_vertices` attribute on the geometry shader output layout declaration. The loop iterates `opt_segments+1` times, and each iteration emits two vertices. So change the output to:

```
layout ( lines ) in;
layout ( triangle_strip, max_vertices = 2 * ( opt_segments + 1 ) ) out;
```

Note that `opt_segments` is provided as a constant, so the declaration above is correct. It would not be correct if `opt_segments` were a uniform variable or storage buffer. The change input topology is also shown.

At this point we still haven't described how the geometry shader gets `link_idx1` and `link_idx2` and the four texture coordinates. The code above, remember, emits the surface between two spokes, one associated with `link_idx1` and the other with `link_idx2`.

The solution is to treat each spoke as a vertex. The CPU code will insert a link index and two texture coordinates for each vertex. The vertex shader will read one such vertex, and emit it. The pipeline will be set up with a line strip topology, `vk::PrimitiveTopology::eLineStrip`, and the geometry shader will be set up to expect lines.

On the CPU side, other than changing the topology, we need to insert just one vertex per spoke, but we need to provide two texture coordinates for that vertex. This is done by packing two texture coordinates into a `pCoor` type. (One texture coordinate is for the inner end of the spoke, and one is for the outer end of the spoke.)

```
  // Code from hw05-sol.cc:World::render_hw05:
  for ( size_t i = 0; i <= n_spokes; i++ ) {
      Link* const spoke1 = hw05_stuff.spokes[i%n_spokes];
      const int link_idx = spoke1->idx;
      bset_hw05 << link_idx;

      pTCoor tex11 = spoke1->ball1->tca[opt_texture];
      pTCoor tex12 = spoke1->ball2->tca[opt_texture];
      pCoor pack;
      pack.x = tex11.x;  pack.y = tex11.y;
      pack.z = tex12.x;  pack.w = tex12.y;
      bset_hw05 << pack;
    }
```

Packing a data type intended for one purpose with data meant for another purpose is stylistically sloppy but is sometimes done for performance reasons.

The vertex shader in the solution code unpacks the textures, arriving at location `LOC_IN_POS`, into two `vec2`. The vertex shader code is shown below:

```
layout ( location = LOC_IN_INT1 ) in int in_link_idx;
layout ( location = LOC_IN_POS ) in vec4 in_pack;

layout ( location = 0 ) out Data_to_GS
{
  vec2 tc1, tc2;
  int link_idx;
} Out;

void vs_hw05() {
  Out.tc1 = in_pack.xy;
  Out.tc2 = in_pack.zw;
  Out.link_idx = in_link_idx;
}
```

In the solution shown so far, it is the geometry shader that uses the link index to retrieve the `pos1`, `pos2`, `v1`, and `v2` values. The solution checked into the repo uses the vertex shader to do that:

```
layout ( location = LOC_IN_INT1 ) in int in_link_idx;
layout ( location = LOC_IN_POS ) in vec4 in_pack;

layout ( location = 0 ) out Data_to_GS
{
  vec4 p1, p2, v1, v2;
  vec2 tc1, tc2;
} Out;

void vs_hw05() {
  Out.tc1 = in_pack.xy;
  Out.tc2 = in_pack.zw;

  Out.p1 = pos1[in_link_idx];
  Out.p2 = pos2[in_link_idx];
  Out.v1 = v1[in_link_idx];
  Out.v2 = v2[in_link_idx];
}
```

The advantage of the approach immediately above is that each data item, such as `pos1[123]` for link index value 123 is retrieved just once. In the version of the code where the geometry shader retrieves `pos1` a particular element, such as `pos1[123]` would be retrieved twice because each vertex (except the first and the last) is an input to two geometry shader invocations. There is also less code in the version immediately above. A possible disadvantage is lower performance because more data must be moved from the vertex shader to the geometry shader, but that depends on what is done with the vertex shader output data.

With the vertex shader immediately above the geometry shader simplifies to:

```
void gs_hw05() {
  float delta_t = 1.0 / opt_segments;
  for ( int s=0; s<=opt_segments; s++ )
    {
      float t = s * delta_t;
      vec4 pos0t_o = bez( In[0].p1, In[0].p2, In[0].v1, In[0].v2, t );
      //                  pos11     pos12     v11       v12
      vec4 pos1t_o = bez( In[1].p1, In[1].p2, In[1].v1, In[1].v2, t );
      //                  pos21     pos22     v21       v22

      //  Emit the vertex along spoke 0  ( In[0] )
      gl_Position = ut.clip_from_object * pos0t_o;
      Out.vertex_e = ut.eye_from_object * pos0t_o;
      Out.tex_coor = mix( In[0].tc1, In[0].tc2, t );
      // Out.normal_e = // later.
      EmitVertex();

      /// Emit the vertex along spoke 1  ( In[1] ).
      gl_Position = ut.clip_from_object * pos1t_o;
      Out.vertex_e = ut.eye_from_object * pos1t_o;
      Out.tex_coor = mix( In[1].tc1, In[1].tc2, t );
      // Out.normal_e = // later.
      EmitVertex();
    }
```

```
  EndPrimitive();
}
```

In the code above notice that **pos1t** has been renamed to **pos0t_o** to match the geometry shader input index, **In[0]**, of the former spoke 1, now called spoke 0.

Finally, the normals. The normal is the vector orthogonal to the surface. A normal can be found by taking the cross product of two vectors on the surface. One such vector is easy to compute, the one from **pos0_o** to **pos1_o**. That vector goes from one spoke to the other. Note that **pos0_o** is computed using function **bez** which takes parameter **t**, which varies from 0 to 1. To find the other vector on the surface we can take the derivative of **bez** with respect to **t**. To make things easy, that is precomputed in **bez_dt**. So by taking the cross product of those two vectors we get the normal:

```
      //  Compute direction along each spoke. Needed to find normals.
      vec3 tan0 = bez_dt( In[0].p1, In[0].p2, In[0].v1, In[0].v2, t );

      //  Compute a vector from point on spoke 0 to spoke 1. For normals.
      vec3 vx = pos1t_o.xyz - pos0t_o.xyz;

      //  Compute and emit surface normal.
      vec3 normal_1_o = cross( tan0, vx );
      Out.normal_e = normalize( mat3(ut.eye_from_object) * normal_1_o );
```

Larger excerpts of the solution are shown below, including with comments and other formatting removed above.

```
void
World::render_hw05(vk::CommandBuffer& cb)
{
  if ( opt_segments != opt_segments_pipe_hw05 )
    {
      if ( pipe_hw05 ) pipe_hw05.destroy();
      opt_segments_pipe_hw05 = opt_segments;
    }

  if ( !pipe_hw05 )
    pipe_hw05
      .init( vh.qs )
      .ds_follow( puni_light_curr )
      .ds_follow( transform )
      .ds_uniform_use( "BIND_UNI_COMMON", buf_uni_common )
      .ds_uniform_use( "BIND_HW05", uni_hw05 )
      .ds_storage_follow( "BIND_LINKS_POS1", lis_pos1 )
      .ds_storage_follow( "BIND_LINKS_POS2", lis_pos2 )
      .ds_storage_follow( "BIND_LINKS_V1", lis_v1 )
      .ds_storage_follow( "BIND_LINKS_V2", lis_v2 )
      .ds_use( sampler, texid_hw )

      /// SOLUTION
      //
      //  Change topology from eTriangleStrip to eLineStrip.
      //
```

```
    .topology_set( vk::PrimitiveTopology::eLineStrip )
    //
    //  Change shader inputs from <int,pTCoor> to <int,pCoor>
    //
    .shader_inputs_info_set<int,pCoor>()
    //
    .shader_code_set
      ("hw05-shdr-sol.cc", "vs_hw05();", "gs_hw05();", "fs_main();",
       pStringF("const int opt_segments = %d;\n", opt_segments) )
    .create();

if ( uni_hw05_needs_update )
  {
    uni_hw05_needs_update = false;
    uni_hw05->color = 1.0 * color_white;
    uni_hw05->opt_texture = opt_texture;
    uni_hw05.to_dev();
  }

bset_hw05.reset( pipe_hw05 );

const size_t n_spokes = hw05_stuff.spokes.size();

for ( size_t i = 0; i <= n_spokes; i++ )
  {
    Link* const spoke1 = hw05_stuff.spokes[i%n_spokes];
    const int link_idx = spoke1->idx;
    pTCoor tex11 = spoke1->ball1->tca[opt_texture];
    pTCoor tex12 = spoke1->ball2->tca[opt_texture];

    /// SOLUTION
    //
    //  Insert just one "vertex", corresponding to a spoke.
    //
    //  The "vertex" consists of a link index, the texture
    //  coordinate at the inner ball, and the texture coordinate at
    //  the outer ball.

    //  Insert the link index (corresponding to this spoke).
    //
    bset_hw05 << link_idx;
    //
    //  The geometry shader will retrieve the coordinates from arrays
    //  pos1 and pos2.

    //  Send two texture coordinates, one for the inner ball and
    //  one for the outer ball. Since a texture coordinate has just
    //  two components a pCoor can be used to hold two texture coordinates,
    //  and so pack the two texture coordinates into a pCoor.
    //
    pCoor pack;
```

```
      pack.x = tex11.x;  pack.y = tex11.y;
      pack.z = tex12.x;  pack.w = tex12.y;
      bset_hw05 << pack;
    }

  bset_hw05.to_dev();

  pipe_hw05.record_draw( cb, bset_hw05 );
}
```

```
/// LSU EE 4702-1 (Fall 2023), GPU Programming
//
 /// Homework 5 – SOLUTION
 //
 //  Modified this file and hw05-sol.cc.
 //  Assignment: https://www.ece.lsu.edu/gpup/2023/hw05.pdf


// Specify version of OpenGL Shading Language.
//
#version 460

#extension GL_GOOGLE_include_directive : enable
#include <light.h>
#include <transform.h>
#include "links-shdr-common.h"
#include "shader-common-generic-lighting.h"

layout ( binding = BIND_UNI_COMMON ) uniform Common_Uniform
{
  Shdr_Uni_Common com;
};

bool opt_tryout1 = bool(com.tryout.x);
bool opt_tryout2 = bool(com.tryout.y);
float opt_tryoutf = com.tryoutf.x;

const int TX_None = 0, TX_Square = 1, TX_Circle = 2;

layout ( binding = BIND_HW05 ) uniform HW_05
{
  vec4 color;
  int opt_texture;
  float ring_hole_frac, one_over_delta_r, dy_d_theta;
};

#ifdef BIND_LINKS_POS1
layout ( binding = BIND_LINKS_POS1 ) buffer sr { vec4 pos1[]; };
layout ( binding = BIND_LINKS_POS2 ) buffer sr2 { vec4 pos2[]; };
layout ( binding = BIND_LINKS_V1 ) buffer spr { vec4 v1[]; };
layout ( binding = BIND_LINKS_V2 ) buffer sc { vec4 v2[]; };
#endif


#ifdef _VERTEX_SHADER_

 /// SOLUTION
//
//   Changed declarations of vertex shader inputs.
//
//   Unsolved vertex shader inputs:
```

```
//       int   For link_idx.
//       vec2  For texture coordinate, type pTCoor on CPU.
//
//   Solution vertex shader inputs:
//       int   For link_idx.
//       vec4  For two texture coordinates. Packed into a pCoor.

layout ( location = LOC_IN_INT1 ) in int in_link_idx;

layout ( location = LOC_IN_POS ) in vec4 in_pack;

// Interface block for vertex shader output / geometry shader input.
//
layout ( location = 0 ) out Data_to_GS
{
  /// SOLUTION
  //
  vec4 p1, p2, v1, v2;
  vec2 tc1, tc2;
} Out;

void
vs_hw05()
{
  /// SOLUTION
  //
  // Unpack in_pack into a 2-element array of texture coordinates.
  Out.tc1 = in_pack.xy;
  Out.tc2 = in_pack.zw;

  // Retrieve positions and vectors in vertex shader.
  //
  Out.p1 = pos1[in_link_idx];
  Out.p2 = pos2[in_link_idx];
  Out.v1 = v1[in_link_idx];
  Out.v2 = v2[in_link_idx];
  //
  // It would also be correct to just send in_link_idx to the geometry
  // shader and have the geometry shader retrieve the positions and
  // vectors.
}


#endif

#ifdef _GEOMETRY_SHADER_

layout ( location = 0 ) in Data_to_GS
{
  /// SOLUTION
  //
```

```
  vec4 p1, p2, v1, v2;
  vec2 tc1, tc2;
} In[];

layout ( location = 0 ) out Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
} Out;

// Type of primitive at geometry shader input.
//

 /// SOLUTION
//
// Change layout from triangles to lines.
//
layout ( lines ) in;

// Type of primitives emitted geometry shader output.
//

 /// SOLUTION
//
//   Increase max vertices from 3 to 2 + 2*opt_segments.
//
layout ( triangle_strip, max_vertices = 2 * ( opt_segments + 1 ) ) out;

vec4
bez(vec4 p1, vec4 p2, vec4 v1, vec4 v2, float t)
{
  float t2 = t * t;
  float t3 = t2 * t;
  return
    vec4( ( 2*t3 - 3*t2 + 1 ) * p1.xyz
          + ( -2*t3 + 3*t2 ) * p2.xyz
          + ( t3 - 2*t2 + t ) * v1.xyz
          - ( t3 - t2 ) * v2.xyz,
          1 );
}

vec3
bez_dt(vec4 p1, vec4 p2, vec4 v1, vec4 v2, float t)
{
  float t2 = t * t;
  float t3 = t2 * t;
  return
    ( 6*t2 - 6*t ) * p1.xyz
    + ( -6*t2 + 6*t ) * p2.xyz
    + ( 3*t2 - 4*t + 1 ) * v1.xyz
```

```
    - ( 3*t2 - 2*t ) * v2.xyz;
}

void
gs_hw05()
{
  float delta_t = 1.0 / opt_segments;

  /// SOLUTION
  //
  //  The input to the geometry shader is a line to GLSL with one end
  //  point described by In[0] and the other endpoint described by
  //  In[1]. But to us, In[0] is spoke 0 and In[1] is spoke 1. (These
  //  can be any pair of adjacent spokes.)
  //
  //  In the loop below the geometry shader emits a triangle strip
  //  to form a surface between the two spokes. The loop below computes
  //  the same values as the s loop in hw05.cc:World::render_hw04.
  //
  //  Note that In[0].p1 in the code below corresponds to pos11 in
  //  render_hw04, In[1].p1 corresponds to pos21 in render_hw04, etc.

  for ( int s=0; s<=opt_segments; s++ )
    {
      float t = s * delta_t;

      // Compute interpolated coordinate along spokes 0 and 1 at position t.
      //
      vec4 pos0t_o = bez( In[0].p1, In[0].p2, In[0].v1, In[0].v2, t );
      //                  pos11     pos12     v11       v12
      //
      vec4 pos1t_o = bez( In[1].p1, In[1].p2, In[1].v1, In[1].v2, t );
      //                  pos21     pos22     v21       v22
      //
      // Note: Curve from In[0].p1 to In[0].p2 is a 3rd degree polynomial,
      // bez finds a point on this curve.

      //  Compute direction along each spoke. Needed to find normals.
      //
      vec3 tan0 = bez_dt( In[0].p1, In[0].p2, In[0].v1, In[0].v2, t );
      vec3 tan1 = bez_dt( In[1].p1, In[1].p2, In[1].v1, In[1].v2, t );

      //  Compute a vector from point on spoke 0 to spoke 1. For normals.
      //
      vec3 vx = pos1t_o.xyz - pos0t_o.xyz;

      //
      ///  Emit the vertex along spoke 0  ( In[0] )
      //

      //  Emit clip-space coordinate and eye-space coordinate.
```

```
      //
      gl_Position = ut.clip_from_object * pos0t_o;
      Out.vertex_e = ut.eye_from_object * pos0t_o;

      //  Compute and emit interpolated texture coordinate.
      //
      Out.tex_coor = mix( In[0].tc1, In[0].tc2, t );
      //
      //  Simple linear interpolation is sufficient for textures.

      //  Compute and emit surface normal.
      //
      vec3 normal_1_o = cross( tan0, vx );
      Out.normal_e = normalize( mat3(ut.eye_from_object) * normal_1_o );

      EmitVertex();

      //
      /// Emit the vertex along spoke 1  ( In[1] ).
      //
      gl_Position = ut.clip_from_object * pos1t_o;
      Out.vertex_e = ut.eye_from_object * pos1t_o;
      Out.tex_coor = mix( In[1].tc1, In[1].tc2, t );
      vec3 normal_2_o = cross( tan1, vx );
      Out.normal_e = normalize( mat3(ut.eye_from_object) * normal_2_o );

      EmitVertex();
    }

  EndPrimitive();
}



#endif


#ifdef _FRAGMENT_SHADER_

#ifdef BIND_TEXUNIT
layout ( binding = BIND_TEXUNIT ) uniform sampler2D tex_unit_0;
#endif

layout ( location = 0 ) in Data_to_FS
{
  vec3 normal_e;
  vec4 vertex_e;
  vec2 tex_coor;
} In;

layout ( location = 0 ) out vec4 out_frag_color;
```

```
void
fs_main()
{
  vec4 lit_color = generic_lighting(In.vertex_e, color, In.normal_e);

  if ( opt_texture == TX_None )
    {
      out_frag_color = lit_color;
    }
  else if ( opt_texture == TX_Circle )
    {
      float dist = length(In.tex_coor);
      float angle = atan(In.tex_coor.y,In.tex_coor.x);
      vec2 tc =
        vec2( ( dist - ring_hole_frac ) * one_over_delta_r,
              angle * dy_d_theta );
      out_frag_color = lit_color * texture(tex_unit_0,tc);
    }
  else
    {
      out_frag_color = lit_color * texture(tex_unit_0,In.tex_coor);
    }
}


#endif
```