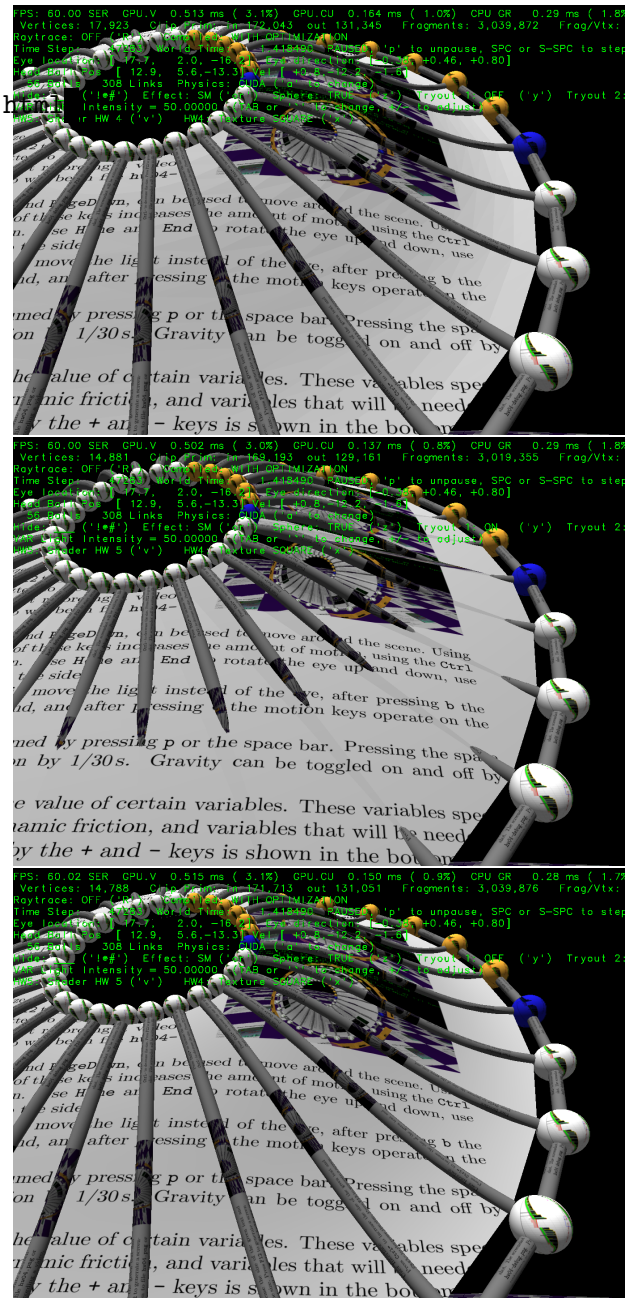
LSU EE 4702-1 **Homework 5** **Due: 22 November 2023**

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).*



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.

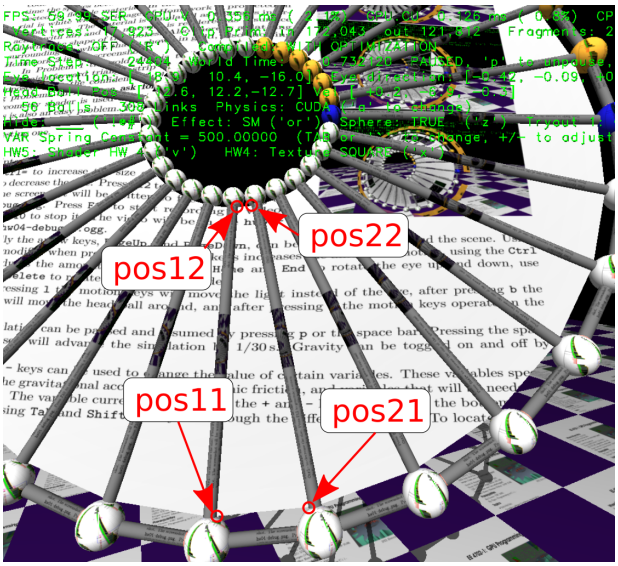
Resources

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`.