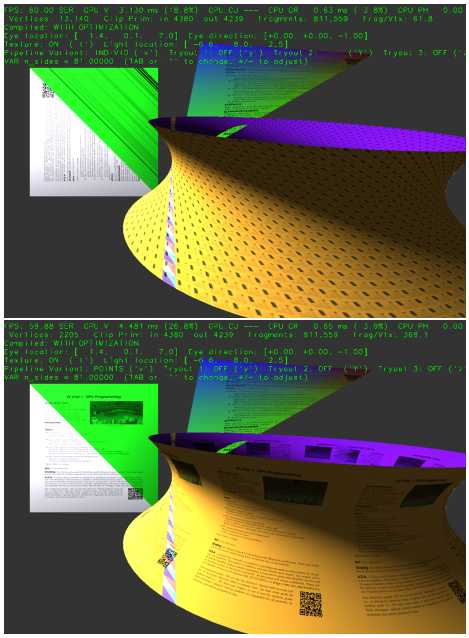
LSU EE 4702-1 **Homework 5** **Due: 1 December 2024**

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/2024/hw05.cc.html` *and* `https://www.ece.lsu.edu/koppel/gpup/2024/hw05-shdr.cc.html`.



Problem 0: In the unlikely event it was 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 homework code should initially show a scene with a rotating hyperboloid of one sheet, see the screenshot on the upper right. In the shot on the lower right, taken from a correctly solved assignment, the square formed by the light gray (it looks white but the name of the color is `color_light_gray`) and green triangles shows the course syllabus right-side-up, and the course syllabus is much larger and better positioned on the hyperboloid. Also notice that the hyperboloid is drawn using a points grouping which uses only half the number of vertices required by the triangle strip grouping.

Homework Overview

The code in this assignment explores methods of applying textures and of using a rendering pipeline in which there are no inputs other than indices. It is up to the shaders to fetch the information they need from storage buffers.

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

Pressing 1 generates a new hyperboloid. The first two are hard-coded, and after that they are randomly generated. Pressing 2 shows the first hyperboloid again.

Pressing v switches between using a pipeline using a triangle list (individual triangles), a triangle strip, and points. The setting is shown in the second line of green text to the right of label Pipeline Variant.

Pressing t toggles textures on and off.

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 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, opt_tryoutf, and opt_tryouti 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 Boolean variables opt_tryout1, opt_tryout2, and opt_tryout3, available in host and shader code. The user interface can also be used to modify floating-point variables opt_tryoutf and opt_tryouti using the Tab, +, and - keys, see the previous section. In shader code use variables opt_tryouti and opt_tryoutf. In host code (2024 Homework 5) use variables uni_misc->opt_tryoutf and for for opt_tryouti use uni_misc->opt_tryout.w.

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. The names of many OGSL functions match corresponding functions in the C++ standard library, but not all of them including one needed for a complete solution to 2024 Homework 5. The OGSL library functions are described in 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.

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++ and OGSL 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 each student's duty to himself or herself 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.

Problems start on the next page!

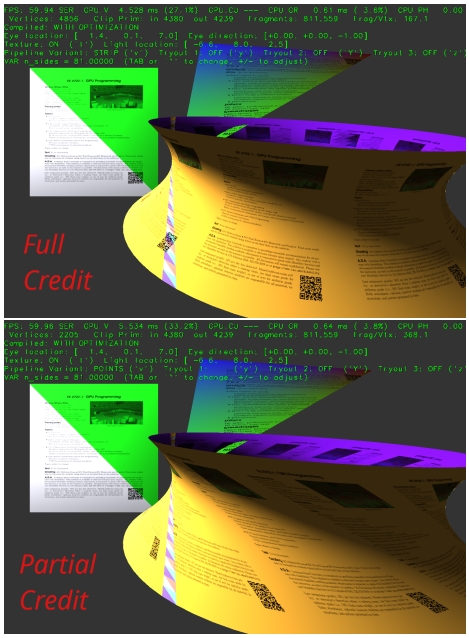
Problem 1: Modify the code in `World::render` so that the square formed by the green and light gray triangle shows the course syllabus right-side-up, as shown in the lower screenshot on the first page of the assignment. Do so by changing the texture coordinates inserted into the buffer set below the "Problem 1" comment:

```
/// Problem 1: Modify texture coordinates below.

// Add a square consisting of a light gray and green triangle.
bset_plain << pCoor(-1,-1,-2) << pCoor(-5,3,-2) << pCoor(-5,-1,-2);
bset_plain << color_light_gray << color_light_gray << color_light_gray;
bset_plain << pTCoor(0,0) << pTCoor(1,1) << pTCoor(0,1);  // Modify this line.

bset_plain << pCoor(-1,-1,-2) << pCoor(-1,3,-2) << pCoor(-5,3,-2);
bset_plain << color_green << color_green << color_green;
bset_plain << pTCoor(0,0) << pTCoor(1,1) << pTCoor(0,1);  // Modify this line.
```

Problem 2: The code in the unmodified assignment assigns texture coordinates to hyperboloid vertices in such a way that there are tiny syllabi repeated many times over the hyperboloid. See the top screenshot on the first page of this assignment. In the unmodified assignment and in the code fragment below texture coordinates are assigned for the triangle-strip hyperboloid so that the height of the syllabus matches one edge of a triangle and the width matches another edge, which is why each syllabus is so small.




```
case PV_Strip:
  bset_hyperb.reset(pipe_hyperb_strip);
  for ( int side = 0; side < n_sides; side++ )
    for ( int lev = 0; lev <= n_levels; lev++ ) {
        pCoor p00 = coords[ idxf( lev, side ) ];
        pCoor p01 = coords[ idxf( lev, side + 1 ) ];
        pNorm n00 = norms[ idxf( lev, side ) ];
        pNorm n01 = norms[ idxf( lev, side + 1 ) ];

        bset_hyperb << p00 << p01;
        bset_hyperb << n00 << n01;
        bset_hyperb << pTCoor(0,lev%2) << pTCoor(1,lev%2);
```

Modify this code so that there is one copy of the syllabus from top to bottom. The number of copies around the circumference can be chosen for readbility. For the solution shown in the upper screenshot there are 3π copies around the circumference. *The factor π is a hint.*

The vertical axis of the texture (syllabus) must be aligned with the axis of the hyperboloid, as shown in the upper screenshot. The lower screenshot shows a **partial-credit** solution in which the vertical axis of the texture is slanted (following the line used to construct the hyperboloid).

Problem 3: Modify the code in `World::render` and the code in `hw05-shdr.cc` so that the hyperboloid is correctly drawn for a points topology (grouping). When the grouping is set to `PV_Points`

(shown as POINTS in the green text) the code uses pipeline pipe_hyperb_points which is set to a vk::PrimitiveTopology::ePointList topology. This pipeline is set up so that the vertex shader is not provided any attributes from the CPU. No color, no normal, not even a vertex coordinate. Since there are no vertex shader inputs there is no buffer set.

Though the vertex shader does not have inputs from the CPU it does have one input that may be useful: gl_VertexID. Suppose the vertex shader is invoked n times. Then for the first invocation gl_VertexID=0, for the second invocation gl_VertexID=1, and so on. Ordinarily the number of invocations is the number of elements in the buffer set. But pipe_hyperb_points does not use a buffer set. Instead, the number of "vertices" is specified in the draw command:

```
case PV_Points: {
    const int n_vtx = 100; // Needs to be changed!
    pipe_hyperb_points.ds_set( transform * trot );
    pipe_hyperb_points.record_draw(cb, n_vtx );
  }
```

In the code above the number of vertices is hardcoded to 100, and so the vertex shader will be invoked 100 times. In a correct solution to this problem the value of n_vtx needs to be changed to the number of invocations needed to draw the hyperboloid.

The geometry shader has a similar input, gl_PrimitiveIDIn. Like the vertex shader, it is set to the number of invocations of the geometry shader. In a point grouping its value will range from 0 to n_vtx-1, where n_vtx is the value used in the record_draw above.

The shaders for pipeline pipe_hyperb_points have access to two arrays, hyperb_coords and hyperb_norms. These arrays have the same contents as coords and norms used by code in World::render to populate the buffer sets for individual triangle and triangle strip groupings. Containers coords and norms are used in the excerpt shown in Problem 2 and elsewhere in World::render(). Here is an excerpt from the solution to this problem showing how a shader can retrieve a coordinate and normal from the arrays:

```
int idx = level + side * uc.s_levels;
vec4 p = hyperb_coords[ idx ];
vec4 n = hyperb_norms[ idx ];
gl_Position = ut.clip_from_object * p;
```

The code above is retrieving a vertex coordinate, p, transforming it to clip space and writing it to a shader output variable. To retrieve p it computed idx using level and side. The value of level and side were themselves computed using one of the gl_..._ID variables. (That part of the solution won't be shown until after the ultimate due date.)

In the unsolved assignment hyperb_coords and hyperb_norms are available, as are the transformation matrices in ut. To solve the problem additional information from the host is needed. That should be placed in uniform uc, which in the CPU code is in variable uni_hw05 of type HW05_Uniform. Add needed members to HW05_Uniform and assign values to those members in uni_hw05. The code above uses member s_levels which does not exist in the unsolved assignment.

The input to the pipeline is points, but we need triangles to render the hyperboloid. The input to the geometry shader is set to points (that's correct, don't change it), and the output is set to triangle_strip, which is also correct. However, the value of max_vertices might need to be changed.

Here is an outline of what needs to be done to solve this problem.

- [ ] Decide how many triangles a geometry shader invocation should emit. Call that g. Don't make g too large.

- [ ] Let T denote the total number of triangles in the hyperboloid. Set `n_vtx` (in the CPU code) to T/g.
- [ ] In the appropriate shader compute `level` and `side` from `gl_..ID`.
- [ ] Use `level` and `side` to retrieve coordinates and normals.
- [ ] Use the retrieved coordinates and normals to compute shader outputs. (See the shaders in `hw04-shdr.cc` to see what needs to be computed for the fragment shader and how to compute it.)
- [ ] Use `level` and `side` to compute the color index.
- [ ] Use `level`, `side`, and maybe coordinates to compute texture coordinates. [ ] For the full-credit texture appearance additional data will be needed from the host. The slanted textures are much easier.
- [ ] If needed, declare vertex-shader outputs and geometry shader inputs.