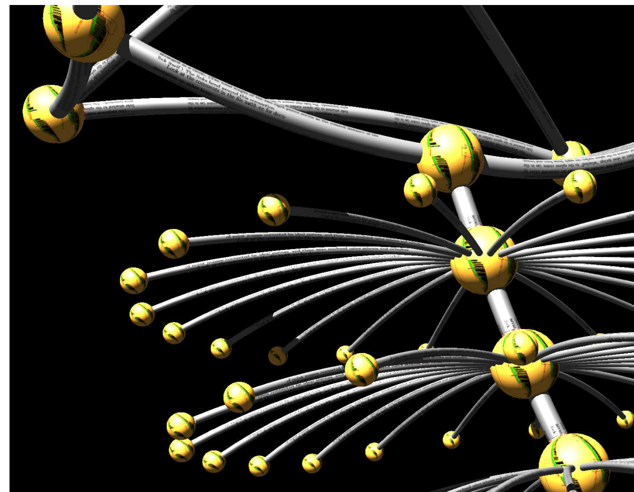
LSU EE 4702-1 Homework 6 Solution Due: 16 November 2016

The solution has been checked into the repo. For a syntax highlighted version of the shader code see `http://www.ece.lsu.edu/koppel/gpup/2016/hw06-shdr-links-sol.cc.html` and for the CPU code see `http://www.ece.lsu.edu/koppel/gpup/2016/hw06-sol.cc.html`.

Problem 0: If necessary, follow the instructions on the `http://www.ece.lsu.edu/koppel/gpup/proc.html` page for account setup and programming homework work flow. For this assignment only edit files `hw06-shdr-links.cc` and `hw06.cc`. Compile and run the homework code unmodified. It should initially show Scene 3, a spikey ball. The vaguely umbrella-shaped object hanging in space from prior assignments but with bristles or something on the vertical tail of balls is in Scene 2 and Scene 1 shows a tree (the kind that grows outdoors).

Pressing `v` will cycle through three different sets of shaders. The shader set that is being used is shown in the penultimate line of green text. Shader set PLAIN is a conventional set of shaders, and is there for comparison purposes. Shader set SET 1 is comprised of vertex shader `vs_main_1`, geometry shader `gs_main_1`, and fragment shader `fs_main`, all in file `hw06-shdr-links.cc`, and is used for Problem 1. Shader set SET 2 is comprised of vertex shader `vs_main_2`, geometry shader `gs_main_2`, and fragment shader `fs_main`, also in file `hw06-shdr-links.cc`, and is used for Problem 2.

The plain set of shaders should be slow, while the other sets show the links in a crude manner. Options that affect performance are: turning shadows on and off `o` and turning mirroring on and off `r`. (Pressing `O` will show shadow volumes.) Note that the links still don't cast shadows.



Press digits `1` through `2` to initialize different scenes, the program starts with scene 2. Scene 1 shows a balls connected in a rectangular spiral. *Promptly report any problems.*

Use key `h` to toggle between the first (head) ball being locked in place and free. Use key `t` to do the same for the last (tail) ball.

Initially the arrow keys, PageUp, and PageDown can be used to move around the scene. Press (lower-case) `b` and then use the arrow and page keys to move the first ball around. Press `l` to move the light around and `e` to move the eye (which is what the arrow keys do when the program starts).

When using the arrow and other keys to move the eye, light, or ball using `Shift` will move by a $5\times$ greater amount and using `Ctrl` will move by one $\frac{1}{5}$ the amount than the motion without either modifier.

Look at the comments in the file `hw06.cc` for documentation on other keys.

A goal of Homework 3 was to reduce the amount of communication between the CPU and GPU by using buffer objects, especially for almost straight links. In this assignment two methods will be used to improve performance: computing the surface of the curved link in the GPU, and using an instanced draw to reduce the number of rendering passes.

See the OpenGL Shading language documentation linked to the references page, `http://www.ece.lsu.edu/koppel/gpup/ref.html`, for syntax and library functions that can be used in shader code.

Problem 1: The code in `vs_main_1` and `gs_main_1` are used to crudely render a link with an instanced draw. Examine the code in `vs_main_1` and `render_link_2_render`. Built-in variable `gl_InstanceID` indicates which link is being rendered, its value can range from 0 to one minus the number of links (or instances). The number of instances is determined by the last argument to `glDrawArraysInstanced`. The built-in vertex shader variable `gl_VertexID` indicates the vertex number being rendered, its range of values is determined by the third argument to `glDrawArraysInstanced`.

Note that in the rendering pass set up in `render_link_2_render` *no vertex coordinates nor any other vertex shader inputs* are sent to the vertex shader. The shader must rely on `gl_InstanceID` and `gl_VertexID` to find its inputs.

Notice that `vs_main_1` reads link endpoint coordinates from buffer objects `pos1` and `pos2`, and using a scaled version of `gl_VertexID`, computes a point partway between the endpoints, `vertex_o`. That coordinate is transformed and sent to the geometry shader for constructing triangles.

Since it renders a straight link there is no reason to use more than two vertices per link. (The curved link will be rendered using `vs_main_2` an `gs_main_2`.)

(a) Modify `vs_main_1`, `gs_main_1`, and `render_link_2_render` so that the complete straight link is rendered using only two vertices per instance without changing the primitive at the input to the geometry shader (a line strip).

- Split work between the vertex shader and the geometry shader to void duplication of effort.
- Make sure that changes to `render_link_2_render` don't affect `vs_main_2`.

See the code checked into the repo for details. In the original code the vertex shader computed a point partway between `pos1` and `pos2` based on t, which varied from 0 to 1. Since there are only two vertices now t would be either 0 or 1 and there's no point in computing it. Instead, of the vertex ID is 0 `pos1` is used, otherwise `pos2` is used. The vertex shader appears below. The geometry shader is modified to adjust the width of the links. See the code in the repo or at the link above for details.

```
void vs_main_1() {
  iid = gl_InstanceID;
  vec3 vertex_1 = pos1[iid].xyz;
  vec3 vertex_2 = pos2[iid].xyz;
  vec4 vertex_o = vec4( gl_VertexID == 0 ? vertex_1 : vertex_2, 1 );
  gl_Position = gl_ModelViewProjectionMatrix * vertex_o;
  vertex_e = gl_ModelViewMatrix * vertex_o;
}
```

Problem 2: The code in `vs_main_2` and `gs_main_2` which also crudely render a link, are placeholders for code rendering the curved link.

(a) Modify `vs_main_2` and `gs_main_2` so that they render the curved link based on the data copied into buffer objects by routines `render_link_2_gather` and `render_link_2_render`. Base your code on the code in `render_link_1`. Be sure that any changes made for the prior problem don't interfere with this one.

- Split work between the vertex shader and the geometry shader to void duplication of effort.

- Write your routine for a link with a maximum of 20 sides.

See the code for details. In the solution the vertex shader computes a point on the curve (the cylinder axis) and vectors orthogonal to the cylinder axis. These are sent to the geometry shader which draws the cylinder.

Problem 3: Estimate the amount of data sent from CPU to GPU for each link for the plain and set 1 and set 2 shaders. (The set 1 and set 2 shaders should send the same amount of data.) Do this by examining the code in `render_link_2_gather` and `render_link_2_render`.

The plain links are rendered in `render_link_1`. As can be seen by the `glEnableClientState` calls, each vertex in the link consists of a coordinate (`GL_VERTEX_ARRAY`, $16\,\mathrm{B}$), a normal ($12\,\mathrm{B}$), and a texture coordinate ($8\,\mathrm{B}$). The number of vertices is $2gs$, where g is the number of segments and s is the number of sides (defined in the code). So the total amount of data per link is $(16\,\mathrm{B} + 12\,\mathrm{B} + 8\,\mathrm{B})sg = 36sg\,\mathrm{B}$. For the default values of $g = 15$ and $s = 20$ that's $10800\,\mathrm{B}$ per link. (The color would add just 12 more bytes.)

The code in `render_link_2_render` sends over 7 buffer objects (for `pos1`, `pos2`, `v1`, `v2`, `b1_ydir`, `b2_ydir`, and `misc`). Each buffer object holds an array of `vec4`s. (A `vec4` is $16\,\mathrm{B}$.) There is one array element per link, so the total amount of buffer object data per link is $7 \times 16 = 112\,\mathrm{B}$. The call to `glDrawArraysInstanced`, which initiates the rendering of many links, does not send any per-vertex data because no client arrays were set up. It would need to send over the type of primitive (a line strip), the number of vertices per instance, and the number of instances. That might take 12 bytes for the entire rendering pass. Finally, in `render_link_2_render` we see that several uniforms are sent over, such as the colors. The total uniform data is 12 floats and 3 integers, which total to $(12 + 3) \times 4 = 60\,\mathrm{B}$. So if n is the number of links to be rendered, the amount of data per link would be $112\,\mathrm{B} + \frac{12\,\mathrm{B}+60\,\mathrm{B}}{n}$.

Obviously $112\,\mathrm{B}$ per link (for large n) is much lower than $10800\,\mathrm{B}$ per link required by the plain shader. Much less data is needed because the shaders compute points on the link given only data about its endpoints.