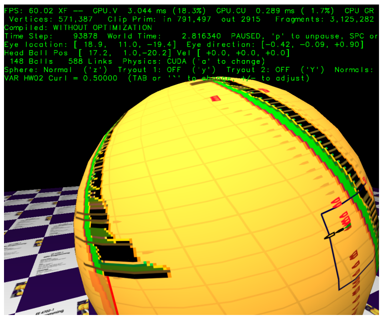
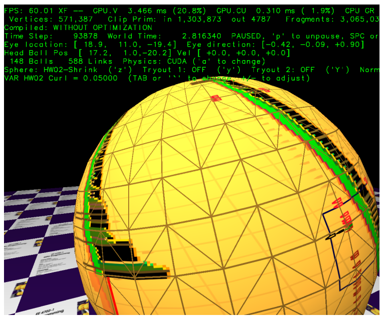
LSU EE 4702-1 **Homework 2** **Due: 9 November 2021**

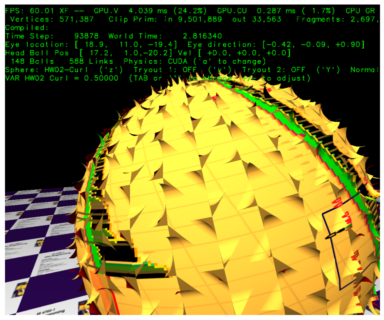
Problem 0: Follow the instructions on the `https://www.ece.lsu.edu/koppel/gpup/proc.html` page for account setup and programming homework work flow. Compile and run the homework code unmodified. It should initially show a scene from the Vulkan links code (in `projbase/v-links`), the one showing a vaguely tree-like form, the *silly tree*, constructed from flexible links and balls. In the original links code the balls are ordinary spheres, which look brand-new no matter how long you run the program. See the screenshot below on the left. In this assignment the spheres are to show effects of aging. Imagine that the texture were applied to the sphere by applying triangular stickers. As time passes two kinds of degradation are possible. For Problem 1 that degradation is shrinkage. The triangular stickers should shrink, reveling the untextured and slightly darker sphere underneath. See the screenshot below in the center. For Problem 2 the triangles should start pealing off at the corners. See the screenshot below on the right.



```
FPS: 60.02 XF --  GPU.V  3.044 ms (18.3%)  GPU.CU  0.289 ms ( 1.7%)  CPU GR
 Vertices: 571,387   Clip Prim: in 791,497  out 2915   Fragments: 3,125,282
Compiled: WITHOUT OPTIMIZATION
Time Step:    93878  World Time:    2.816340  PAUSED, 'p' to unpause, SPC or
Eye location: [ 18.9,  11.0, -19.4]  Eye direction: [-0.42, -0.09, +0.90]
Head Ball Pos  [ 17.2,  1.0,-20.2] Vel [ +0.0, +0.0, +0.0]
 148 Balls   588 Links  Physics: CUDA ('a' to change)
Sphere: Normal  ('z')  Tryout 1: OFF  ('y')  Tryout 2: OFF  ('Y')  Normals:
VAR HW02 Curl = 0.50000  (TAB or '`' to change, +/- to adjust)
```



```
FPS: 60.01 XF --  GPU.V  3.466 ms (20.8%)  GPU.CU  0.310 ms ( 1.9%)  CPU GR
 Vertices: 571,387   Clip Prim: in 1,303,873  out 4787   Fragments: 3,065,03
Compiled: WITHOUT OPTIMIZATION
Time Step:    93878  World Time:    2.816340  PAUSED, 'p' to unpause, SPC or
Eye location: [ 18.9,  11.0, -19.4]  Eye direction: [-0.42, -0.09, +0.90]
Head Ball Pos  [ 17.2,  1.0,-20.2] Vel [ +0.0, +0.0, +0.0]
 148 Balls   588 Links  Physics: CUDA ('a' to change)
Sphere: HW02-Shrink  ('z')  Tryout 1: OFF  ('y')  Tryout 2: OFF  ('Y')  Norm
VAR HW02 Curl = 0.05000  (TAB or '`' to change, +/- to adjust)
```



```
FPS: 60.01 XF --  GPU.V  4.039 ms (24.2%)  GPU.CU  0.287 ms ( 1.7%)  CPU GR
 Vertices: 571,387   Clip Prim: in 9,501,889  out 33,563   Fragments: 2,697,
Compiled:
Time Step:    93878  World Time:    2.816340
Eye location: [ 18.9,  11.0, -19.4]  Eye direction: [-0.42, -0.09, +0.90]
Head Ball Pos  [ 17.2,  1.0,-20.2] Vel [ +0.0, +0.0, +0.0]
 148 Balls   588 Links  Physics: CUDA ('a' to change)
Sphere: HW02-Curl  ('z')  Tryout 1: OFF  ('y')  Tryout 2: OFF  ('Y')  Normal
VAR HW02 Curl = 0.50000  (TAB or '`' to change, +/- to adjust)
```

Non-Assignment-Specific User Interface

Press digits `1` through `4` to initialize different scenes, the program starts with scene 1. Scene 1 starts with the balls arranged in the tree-like form.

Press `p` to pause the simulation.

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size. Initially the arrow keys, `PageUp`, and `PageDown` can be used to move around the scene. Using the `Shift` modifier 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. Press `l` to move the light around and `e` to move the eye (which is what the arrow keys do when the program starts).

The `+` and `-` keys can be used to change the value of certain variables. These variables control things like light intensity and options needed for this assignment. The variable currently affected by the `+` and `-` keys is shown in the bottom line of green text next to VAR. Pressing `Tab` cycles forward through the different variables.

Look at the comments in the file `hw02.cc` for documentation on other keys.

Assignment-Specific User Interface

The sphere can be rendered by four different shaders, `Normal`, `HW02-Shrink`, `HW02-Curl`, and `True`. The shader being used is shown to the right of `Sphere` in the green text. To cycle through the shaders (except for `True`) press `z`. To toggle between the `True` shader and some other press `Z`.

The amount of curling or shrinkage is specified by variable `opt_hw02_curl`. (This variable should affect curling and shrinkage.) This can be modified using the UI, look for that variable name to the right of VAR.

Pressing `n` toggles between computing sphere lighting based on triangle normals, `TRI`, and sphere normals, `SPHERE`. When solving the problems it might help to rendering using triangle normals so that you can see the triangle boundaries. It might also be helpful to adjust light intensity to make effects more visible.

Graphics and Performance Investigation Options

The user interface can be used to toggle various rendering options and for generating a screenshot.

The scenes differ in the number of objects, which include spheres, links, and the platform (which for this assignment we'll consider one object). The rendering of objects by type can be toggled on and off by pressing `!`, `@`, `#`, for spheres, links, and the platform. See the green text line starting with Hide.

Pressing `F12` will write a screenshot to file with a name like `hw02-00.png`. The number at the end will start at `00` and will be incremented for each screenshot *within a run*. When the code is run again the count starts at zero, so be sure to rename files that you want to keep.

The rendering of shadows is toggled by `o` and the rendering of reflections it toggled by `r`. Their state is shown in the green text next to Effect:. Pressing `n` will toggle how surface normals are computed for tessellated spheres, the possibilities are to use the triangle normal or the sphere normal. The use of the triangle normals makes it easier to see the triangles from which the sphere was tessellated.

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.

GPU.V shows how long the GPU spends updating the frame buffer (per frame), GPU.CU shows how long the computational accelerator takes per frame. The computational accelerator computes 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 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`.

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.

The `+` and `-` keys can be used to change the value of certain variables. These variables specify things such as the light intensity, sphere radius, 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` 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.

Code Generation and Debug Support

The compiler generates two versions of the code, `hw02` and `hw02-debug`. Use `hw02` to measure performance, but use `hw02-debug` for debugging. The `hw02-debug` version is compiled with optimization turned off and with Vulkan validation turned on. You are strongly encouraged to run `hw02-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page.

When Vulkan validation is on (which is currently in both debug and normal versions) helpful error and warning messages will be printed about misuse or abuse of the Vulkan API. These will appear on the terminal window (which might be a gdb session) from which `hw02-debug` was started. Currently, the code is set so that all warnings are fatal (except for a select few, which won't generate messages).

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`. 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.

Instanced Draws, and the Assignment Sphere Instance Shaders

The code in the assignment package can render the sphere two ways, using the true sphere shader, which is not a part of this assignment (but still can be run), and the instanced shaders which are.

The sphere rendering that was described in `demo-09-shader.cc` rendered a single sphere. Recall that an input buffer set was prepared holding coordinates of the surface of the sphere in an order appropriate for triangle strips. The coordinates are for a sphere with its center at the origin and of radius one. A transformation matrix was used to scale and move the sphere to wherever we like. In homework 1 multiple spheres are rendered by recording multiple draws each with the same sphere buffer set but with different transformation matrices.

An instanced draw does something like that using a single draw command. The Vulkan draw command, `vkCmdDraw` and the equivalent Vulkan hpp draw member function `Command-Buffer::draw` (See Vulkan Chapter 21.3) take an `instanceCount` argument that specifies the number of instances to draw. So far in this class `instanceCount` has been set to 1, but when using our Sphere Instanced shaders the `instanceCount` will be set to the number of spheres we want to render. The code preparing buffer sets and recording the instanced draw is not part of this assignment. But for those who are curious, the Vulkan record draw calls are made in `VPipeline::record_draw_general` in file `include/vutil-pipeline.h`. Code using the course helpers might call `pipe.record_draw_instanced(cb, bufset, n_instances)`. For the homework assignment spheres `record_draw_instanced` is called in not-well-organized `Sphere::render_bunch_render` in file `shapes.h`. Remember, the host-side code discussed above is not part of the assignment. (Shader code is, which is discussed below.)

Suppose `instanceCount` is set to 2. Then the buffer set will be streamed into the pipeline inputs twice. The first time OpenGL Shading Language variable `gl_InstanceIndex` will be set to 0, the second time it will be set to 1. It is up to the shader code to use the value of `gl_InstanceIndex` to retrieve, in the case of our spheres, the location, radius, and orientation of the sphere. That is done by the vertex shader `vs_main_instances_sphere` in file `hw02-shdr.cc` (and other files). It reads information from three storage buffers, `sphere_pos_rad`, `sphere_rot`, and `sphere_color`, and uses these to compute clip- and eye-space coordinates, as well as texture coordinates. (Note that it would be possible to replace `sphere_pos_rad` and `sphere_rot` by a single transformation matrix.) The subsequent shader stages are no different than in an un-instanced draw.

In this assignment a geometry shader is also used to apply the special affects described in the problems. In the unmodified assignment the geometry shaders, `gs_main_shrink` and `gs_main_curl`, just omit an ordinary triangle.

Problem 1: Modify `gs_main_shrink` (in file `hw02-shdr.cc`) and the fragment shader `fs_main_common` so that they show a shrunken triangle as described below. The amount of shrink should be based on `opt_hw02_curl`. A value of 0 indicates no shrink, a value of 1 indicates that the triangle should shrink to a point. This shader is used when `HW02-Shrink` is shown in the user interface.

The shrunken triangle should carry the entire texture. (That's an easy thing to do.) Underneath the shrunken triangle there should be another triangle at the original size, but with a darker color and without the texture applied.

Here are some things that need to be done:

- Find the coordinates of the shrunken triangle. Do so so that the center of both triangles is the same, except that the shrunken triangle is slightly further from the sphere center.
- Find a way of rendering the original triangle so that it does not show texture and has a darker color. Changing the color is easy. Suppress the color by sending something down the rendering pipeline as a signal to the fragment shader. Don't just set the texture coordinates to zero.
- Don't forget to increase the number of output vertices in the geometry shader declaration.
- Note that the fragment shader uses red for the back face of primitives.

Problem 2: Modify `gs_main_curl` (in file `hw02-shdr.cc`) and the fragment shader `fs_main_common` so that they show some triangles peeling off at one vertex. See the screenshot. Base the fraction of the triangle peeled off on the value of `opt_hw02_curl`.

- The peeling should start at a vertex of your choosing, and the peeled part should be in the shape of a cylinder.
- As with the shrunken triangles, show an untextured, darker-colored triangle underneath.
- The normals on the unpeeled part should be sphere normals (as they are in the unmodified code). The normals on the peeled part should be correctly chosen for the cylinder.
- The cylinder should be placed so the peeling starts smoothly, as in the screenshot. The radius of the cylinder can be chosen to taste.

As a resource for solving this problem look at the shader rendering the bump in demo-10. (Those files are in the `gp/vulkan/` subdirectory.) Also check out past homework OpenGL shading language assignments.

To help you get started `gs_main_curl` has code to compute the coordinates of vertices of the peeled triangles. Sorry for spoiling the fun.

- Do not expect shadows to work correctly. In particular, light won't go through the holes interrupting the shadow cast by the part of the sphere that is present.
- Don't forget to increase the number of output vertices in the geometry shader declaration. Note that a single value is used for both. Use the value of `slices` for that. Set the value to the smallest value that would work for both problems.
- Note that the fragment shader uses red for the back face of primitives.