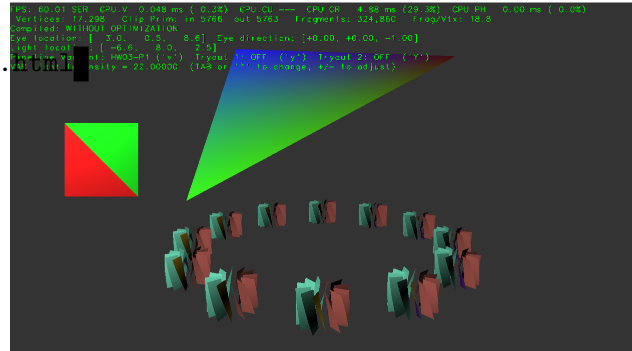
LSU EE 4702-1 Homework 3 Due: 17 October 2022

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 the square and triangle from the `cpu-only/demo-03-vulkun-one.cc` code used in class, but with spinning turbines. See the screenshot to the right, which shows the correctly solved assignment. In the unsolved code all the turbines are gray.



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 `hw03.png` or `hw03-debug.png`.

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).

Assignment-Specific User Interface

The scene can be reset by pressing `1` and `2`. When `1` is pressed the number of turbines and their positions is randomly chosen. When `2` is pressed the number and positions are the same each time. Pressing `p` will stop and start the turbines from spinning.

The scene can be rendered using three different pipelines, *Plain*, *HW03-P1*, and *HW03-P2*. The particular set of code in use is shown by the green text to the right of `Pipeline Variant`. Pressing `v` switches between the variants.

Code Generation and Debug Support

The compiler generates two versions of the code, `hw03` and `hw03-debug`. Use `hw03` to measure performance, but use `hw03-debug` for debugging. The `hw03-debug` version is compiled with optimization turned off. You are strongly encouraged to run `hw03-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page.

To help you debug your code and experiment in one way or another, the user interface lets you change variables. In this assignment these include the variables `opt_tryout1` and `opt_tryout2`, which are available both in CPU code and in 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` and `Y` toggle the value of these variables. Their values are shown in the green text.

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 rasterization. (Other assignments will use ray tracing.) 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.

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.

Assignment Background

The key element of this assignment is the rendering of will be called turbines here. (They may not be shaped like real turbines.) Structure `Fan_Info` describes the shape of a turbine and how the blades should be colored. The description is similar though not identical to the way the paddle wheel from Homework 2 was described.

```
struct Fan_Info {
  pCoor pos_axis_top, pos_axis_bottom;
  pVect axis_to_cyl;
  float outer_radius;
  int n_blades;
  pColor blade_0_front_upper, blade_0_front_lower;
  pColor blade_0_back_upper, blade_0_back_lower;
  pColor blade_i_front_upper, blade_i_front_lower;
  pColor blade_i_back_upper, blade_i_back_lower;
  int serial;
};
```

The values of coordinates and vectors used above are in a local coordinate space. A transformation matrix is used to move the turbine to its intended position. In this assignment there is one `Fan_Info` object `fan_info`, and a array of transformation matrices, `fans_xforms`. (The code to generate triangles corresponding to the turbines has already been written. Look for `case PV_Plain` in the code.)

Routine `fan_setup` writes both `fan_info` and `fans_xforms`, it is called when `1` or `2` is pressed. Each time `fan_setup` executes it increments `fan_info.serial`. The purpose of `fan_info.serial` is to make it possible for other code to detect when `fan_info` has changed. (That is relevant to this assignment.)

In this assignment the turbines are to be rendered by the three pipeline variants, plain, hw03p1,

and hw03p2. (See the assignment-specific user interface section, above.) The code for the plain variant is complete (except maybe for the colors, see Problem 1).

The plain variant uses `VPipeline` `pipe_plain` and buffer set `bset_plain`. Here is an excerpt of the code, omitting the normals for brevity (the complete code is in `hw03.cc`):

```
case PV_Plain:
  for ( auto& m: fans_xforms )
    for ( int i=0; i<f.n_blades; i++ ) {
        float theta = theta_0 + i * delta_theta;
        pCoor p1_last, p2_last;
        for ( int j=0; j<=n_slices; j++ ) {
            float eta = theta + j * delta_eta;
            pVect v = ax * cosf(eta) + ay * sinf(eta);
            pCoor pa = f.pos_axis_top + delta_vz * j;
            pCoor p3 = m * ( pa + r1 * v );
            pCoor p4 = m * ( pa + f.outer_radius * v );
            if ( j ) {
                bset_plain << p1_last << p3 << p2_last;
                bset_plain << p2_last << p3 << p4;
                bset_plain << gray << gray << gray << gray << gray << gray;
              }
            p1_last = p3; p2_last = p4;
          }
      }
```

The `j` loop computes triangles for one blade, the `i` loop computes triangles for one turbine, and the `m` loop computes the set of turbines. Those concerned about computational waste will be appalled by the loop nest above. Why? First, the same color is used over and over, surely there must be a better way of specifying the same color a zillion times. That will be fixed in Problem 1, and in a way that allows both front and back colors to be used. Another wasteful issue is the fact that each iteration of the `m` loop does almost exactly the same thing, the only difference is the value of `m`. That will be fixed in Problem 2.

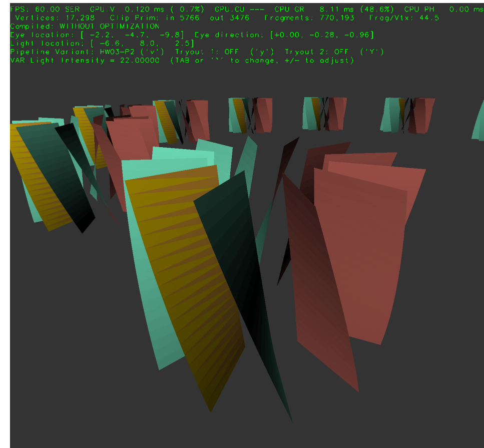
Code Overview

Routine `World::render` is called each time the frame buffer is to be prepared. It is similar to other render routines used in class, the parts relevant to this assignment will be discussed here.

Objects `pipe_plain` and `pipe_hw03` are `VPipeline` instances. `World::render` initializes both of them, though in the solution to this assignment the initialization of `pipe_hw03` will need to be modified.

Some notes on how to work with the `VPipeline`, `VVertex_Buffer_Set`, and the buffer types can be found in this under-construction Vulkan course library note set.

Problem 1: Modify the code in `render` so that the turbine uses colors specified in `fan_info` and helpfully written to variables `b0_fl`, etc. The `blade_0` colors are to be used by the blade emitted when `i=0` in the `case PV_Plain` loop nest. The `blade_i` colors are to be used when `i>0`. The front color is to be used for the front of the blade, and back for the back. (See the fragment shader routine, `fs_main`, in `hw03-shdr.cc`.) Each iteration of the `j` loop emits two triangles. The first is the upper triangle and the second is the lower triangle. See the screenshot to the right. In the screenshot blade 0 has distinct upper- and lower-triangle colors, but in the other blades they are the same. Don't assume that this will always be the case.



There are three parts to this problem, only the last, part c, really needs to be solved. But solving them in order may be helpful.

(*a*) Modify the code for the `PV_Plain` case so that it uses the front colors from the `fan_info` structure, as described above. Just the front colors for this part because the `PV_Plain` pipeline variant can't take back colors. Note that this part is much easier than the others.

(*b*) For this part (and the next one) write a new set of shaders in `hw03_shdr.cc` and modify the CPU code (in many places in `hw03.cc`) so that instead of using a color as a vertex attribute, an integer is used instead. Call this integer a *color set index*. The idea is to replace the code that uses colors:

```
bset_hw03 << color_red << color_blue << color_red;
```

with code that uses indices

```
bset_hw03 << 1 << 2 << 1;
```

The indices refers to colors that will have been placed in a uniform object as part of the solution. The example above implies that 1 refers to position 1 somewhere in the uniform (maybe an array element), which presumable holds red. But for this problem the color index should be used to refer to a pair of colors, front and back. For this subproblem use an existing uniform object to store the colors, but in the next part declare your own. To make things easier, the shader code already expects that there might be an integer input and there might not be a color input:

```
 // Vertex Shader Inputs
layout (location = LOC_IN_POS) in vec4 in_vertex_o;
layout (location = LOC_IN_NORMAL) in vec3 in_normal_o;
#ifdef LOC_IN_COLOR
layout (location = LOC_IN_COLOR) in vec4 in_color;
#endif
#ifdef LOC_IN_INT1
layout (location = LOC_IN_INT1) in int in_color_idx;
#endif
```

The following changes need to be made to the shader code:

- A color array(s) must be added to an existing uniform. Note that a color can be declared pColor in CPU code but must be declared `vec4` in shader code.

- The changes to the uniform in the shader code must be made to the corresponding structure in the CPU code.
- Modify the shaders so that they get the correct color using the color set index, named `in_color_idx` in the shader code.
- The shaders should no longer use the `in_color` input.
- Compute a front and back lighted color in the vertex shader.
- Modify the vertex shader `out` declaration and the fragment shader `in` declaration to accommodate the back color.
- Modify the fragment shader to use the back color when appropriate.

On the CPU side the following changes are needed:

- Add the color declarations to the struct used for colors (corresponding to the uniform used in the shader code).
- Populate the uniform with the `fan_info` colors.
- Modify the code in the `PV_HW03_P1` case so that it uses color set indices rather than colors.

When this problem is solved correctly the colors should match the screenshots.

(*c*) In the previous subpart the colors might have been placed in the `Uni_Light_Simple` or `Uni_Misc` structures because they were already there. In this subproblem define a new structure and use that for a new buffer. The new buffer will need to be declared in `World`, initialized and later destroyed in `World::setup_and_run`. In `World::render` the new uniform will need to be bound to the `pipe_hw03` descriptor set (that's what `ds` stands for) when `pipe_hw03` is initialized, and explicitly copied `to_dev`, preferably only when the colors change (see the `serial` member of `fan_info`). Most of the steps needed for your new buffer will match those for `uni_light_simple` and `uni_misc`.

Problem 2: As mentioned in the introductory material, the code in `PV_Plain` and `PV_HW03_P1` is wasteful because it re-computes the triangles for all the turbines, even though each turbine is identical (except for position and orientation). In this problem add code to the `PV_HW03_P2` case that writes triangles to `bset_hw03_p2` for just one turbine. In fact, only write these when the serial in `fan_info` changes. Make sure not to reset the buffer set only when it needs to be re-populated. Next, record a draw for each transformation matrix. The code should look something like this:

```
for ( auto& m: fans_xforms )
  {
    pMatrix global_from_local = ..;
    pipe_hw03.ds_set( transform * global_from_local );
    pipe_hw03.record_draw(cb, bset_hw03_p2);
  }
```

The `pipe_hw03.ds_set` line is setting the `eye_from_object` matrix used by the pipeline. A draw is then recorded using the one-turbine buffer set. The loop iterates for each pipeline. In contrast, the code for the plain and problem 1 variants record one draw for all the turbines. They must spend more time preparing the buffer set, and it takes longer to send the buffer set from the CPU to the GPU. When this problem is solved correctly the buffer set is much smaller. Though it is used many times, it is only sent from the CPU to the GPU once (or each time it changes). There is a downside to this method: there is the overhead of setting up the draw, including changing the transform. That will be fixed in later assignments (using instances and storage arrays)

If this is solved correctly then the appearance using the problem 1 and 2 variants should be the same.