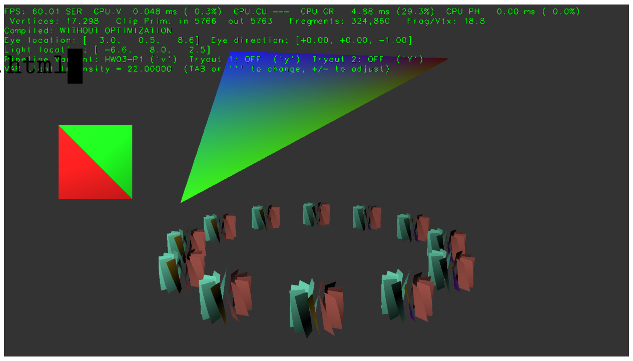
LSU EE 4702-1 Homework 3 Solution Due: 17 October 2022



```
FPS: 60.01 SER  GPU.V  0.048 ms ( 0.3%)  GPU.CU ---  CPU CR   4.88 ms (29.3%)  CPU PH   0.00 ms ( 0.0%)
 Vertices: 17,298   Clip Prim: in 5766  out 5763   Fragments: 324,860   Frag/Vtx: 18.8
Compiled: WITHOUT OPTIMIZATION
Eye location: [  3.0,   0.5,   8.6]  Eye direction: [+0.00, +0.00, -1.00]
Light location: [ -6.6,   8.0,   2.5]
Pipeline Variant: HW03-P1 ('v')  Tryout 1: OFF  ('y')  Tryout 2: OFF  ('Y')
VAR Light Intensity = 22.00000  (TAB or '`' to change, +/- to adjust)
```

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;
          }
      }
```

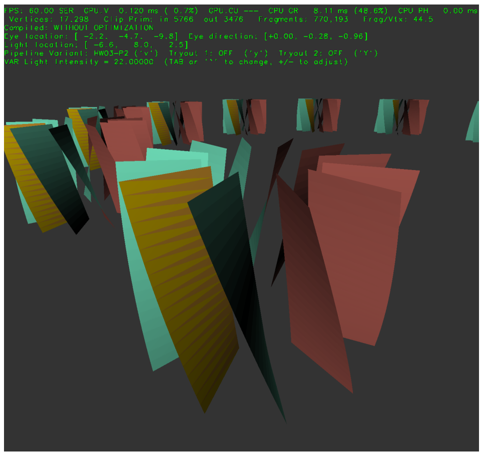
The `j` loop computes triangles for one blade, the `i` loop computes triangles for one turbine, and the `m` loop computes the set of turbines. Those concerned about computational waste will be appalled by the loop nest above. Why? First, the same color is used over and over, surely there must be a better 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.



```
FPS: 60.00 SER  GPU.V  0.120 ms ( 0.7%)  GPU.CU ---  CPU CR   8.11 ms (48.6%)  CPU PH   0.00 ms
 Vertices: 17,298   Clip Prim: in 5766  out 3476   Fragments: 770,193   Frag/Vtx: 44.5
Compiled: WITHOUT OPTIMIZATION
Eye location: [ -2.2,  -4.7,  -9.8]  Eye direction: [+0.00, -0.28, -0.96]
Light location: [ -6.6,   8.0,   2.5]
Pipeline Variant: HW03-P2 ('v')  Tryout 1: OFF  ('y')  Tryout 2: OFF  ('Y')
VAR Light Intensity = 22.00000  (TAB or '`' to change, +/- to adjust)
```

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.

To solve this the code in `case PV_Plain` that inserts colors into the buffer set must be changed so that it inserts the chosen colors. Before the change that code is:

```
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 ) {
        pNorm n = cross(v,pVect(p3,p1_last));
        bset_plain << p1_last << p3 << p2_last;
        bset_plain << p2_last << p3 << p4;
        bset_plain << n << n << n   << n << n << n;
        bset_plain << color_light_gray << color_light_gray;
        bset_plain << color_light_gray << color_light_gray;
        bset_plain << color_light_gray << color_light_gray;
      }
    p1_last = p3;  p2_last = p4;
  }
```

Notice that the code inserts six copies of `color_light_gray` into the buffer set, that colors two triangles, an upper triangle and a lower triangle. As a first step we can change those to the provided front colors for blade `i`:

```
    if ( j ) {
        pNorm n = cross(v,pVect(p3,p1_last));
        bset_plain << p1_last << p3 << p2_last;
        bset_plain << p2_last << p3 << p4;
        bset_plain << n << n << n   << n << n << n;
        bset_plain
```

```
          << f.blade_i_front_upper << f.blade_i_front_upper
          << f.blade_i_front_upper
          << f.blade_i_front_lower << f.blade_i_front_lower
          << f.blade_i_front_lower;
      }
```

Notice that two colors are used, one for the upper and one for the lower triangle. To use separate colors for blade 0 and the other blades use an `if` statement with the condition checking `i`, the blade number:

```
   if ( j ) {
       pNorm n = cross(v,pVect(p3,p1_last));
       bset_plain << p1_last << p3 << p2_last;
       bset_plain << p2_last << p3 << p4;
       bset_plain << n << n << n   << n << n << n;

       if ( !i )
         bset_plain
           << f.blade_0_front_upper << f.blade_0_front_upper
           << f.blade_0_front_upper
           << f.blade_0_front_lower << f.blade_0_front_lower
           << f.blade_0_front_lower;
       else
         bset_plain
           << f.blade_i_front_upper << f.blade_i_front_upper
           << f.blade_i_front_upper
           << f.blade_i_front_lower << f.blade_i_front_lower
           << f.blade_i_front_lower;
     }
```

The shaders used for the `PV_Plain` case use the same color for front and back so to get the back colors a new shader is needed. That's parts b and c of this problem.

(*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.

See the solution to part c.

(*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`.

To solve this problem a structure will be prepared that has two arrays of color, **front** and **back**. The different colors (blade 0 upper front, blade 0 lower front, etc) will be written into the arrays. Color index 0 will use the first element (for front and back), etc.

First, declare a structure to hold the colors. Because data in this structure will be sent to the GPU it is important the structure members are *plain old data* (POD), meaning they don't have member functions and they aren't pointers. The structure used in this solution is:

```
constexpr int ncolors = 10;
struct HW03_Colors {
  pColor front[ncolors], back[ncolors];
};
```

Next, we need to declare a **VBufferV** object to hold this structure. This will be put in the **World** class (near the end), which holds the other parts of our scene:

```
  enum Pipeline_Variant { PV_Plain, PV_HW03_P1, PV_HW03_P2, PV_SIZE };
  int pipeline_variant;

  VBufferV<HW03_Colors> uni_hw03_colors;
```

Notice that to declare a **VBufferV** we need to use the type of the data, **HW03_Colors**, in the template parameter. The name of the object is **uni_hw03_colors**.

Next, we need to initialize **uni_hw03_colors**. Initialization is done once, and we'll do it in **World::setup_and_run**, where other buffers are initialized. When initializing a buffer we need to specify what the buffer will be used for. In this case, our usage is as a uniform buffer. Here is the initialization with some context:

```
  fan_info.serial = 0;

  uni_hw03_colors.init(vh.qs,vk::BufferUsageFlagBits::eUniformBuffer);
  bset_p2_serial = 0;

  opt_pause = false;
  scene_setup_1();
```

The first argument, **vh.qs**, is something like a context and provides various information needed to complete the initialization.

Vulkan pipelines don't automatically have access to uniform variables. In order for a pipeline's shaders to access a uniform we must *bind* the uniform to the pipeline. With the course library a buffer is bound to a pipeline by calling one of the **ds_** member functions, in this case using **ds_uniform_use(LABEL,BUFFER)**. Here LABEL is a string that is used in the shader code to specify the *location* of the uniform variable in the *descriptor set*. BUFFER is a Vulkan buffer handle (type **vk::Buffer**). The Homework 3 pipeline, **pipe_hw03**, is initialized as follows (in **World::render**):

```
  if ( !pipe_hw03 )
    pipe_hw03
      .init( vh.qs )
      .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
      .ds_uniform_use( "BIND_MISC", uni_misc )
      .ds_uniform_use( "BIND_HW03", uni_hw03_colors ) // SOLUTION -- Prob 1c
      .shader_inputs_info_set<pCoor,pNorm,int>()      // SOLUTION -- Prob 2
      .shader_code_set
      ("hw03-shdr-sol.cc", "vs_main();", nullptr, "fs_main();")
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();
```

Notice that the second argument to **ds_uniform_use** is **uni_hw03_colors**, a **VBufferV** object, not a Vulkan buffer handle. The **VBufferV** object has a cast operator-overload that returns the handle, so **uni_hw03_colors** can be used wherever a **vk::Buffer** type is expected. (The **shader_inputs_info_set** change will be discussed later.)

Object **uni_hw03_colors** needs to be written with the proper colors. That is done further down in the render routine, just below the convenience variables (which weren't used, because the fully spelled out versions are more readable):

```
  pColor bi_fl [[maybe_unused]] = f.blade_i_front_lower;
  pColor bi_bu [[maybe_unused]] = f.blade_i_back_upper;
  pColor bi_bl [[maybe_unused]] = f.blade_i_back_lower;

  if ( pipeline_variant != PV_Plain )
    {
      uni_hw03_colors->front[0] = f.blade_0_front_upper;
      uni_hw03_colors->back[0] = f.blade_0_back_upper;
      uni_hw03_colors->front[1] = f.blade_0_front_lower;
      uni_hw03_colors->back[1] = f.blade_0_back_lower;
      uni_hw03_colors->front[2] = f.blade_i_front_upper;
      uni_hw03_colors->back[2] = f.blade_i_back_upper;
      uni_hw03_colors->front[3] = f.blade_i_front_lower;
      uni_hw03_colors->back[3] = f.blade_i_back_lower;

      uni_hw03_colors.to_dev();
    }
```

Notice that the code is using **uni_hw03_colors** as though it were a pointer to the structure. It's not really a pointer, it's just that the **->** operator has been overloaded to provide convenient access to an object of the type specified in the template parameter used to declare the **VBufferV<T>** object (here **T** is the template parameter). After writing the colors, member **to_dev** is called. This copies the CPU version of the structure to the GPU (the device).

The code above that writes **uni_hw03_colors** and calls **to_dev** is guarded by an **if** statement. The **if** avoids writing **uni_hw03_colors** when we are using the plain shader (which doesn't use the uniform). A good thing to add to the **if** condition would be a check of whether the colors have changed (in the **fan_info** structure). If they hadn't changed since **uni_hw03_colors** was last written there would be no reason to write them again. The code above though wastefully updates the colors every frame.

At this point a uniform buffer holding the colors has been prepared and sent to the GPU. Next, we need to stream color indices rather than colors into the rendering pipeline. Originally the pipeline was set up to expect three vertex attribute, a **pCoor**, **pNorm**, and **pColor**, set by the **VPipeline shader_inputs_info_set** member function:

```
  if ( !pipe_hw03 )
    pipe_hw03
      .init( vh.qs )
      .ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
      .ds_uniform_use( "BIND_MISC", uni_misc )
      .shader_inputs_info_set<pCoor,pNorm,pColor>()
      .shader_code_set
      ("hw03-shdr.cc", "vs_main();", nullptr, "fs_main();")
      .topology_set( vk::PrimitiveTopology::eTriangleList )
      .create();
```

As seen further above, the `pColor` type has been replaced by an `int`. The unmodified code already has a declaration ready for the integer attribute:

```
#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
```

So the vertex shader will use variable `in_color_idx` for the vertex index. Returning to the CPU code, we need to replace colors with color indices. Here, we are using 0 for blade 0 upper (front and back), 1 for blade 0 lower (front and back), etc. (The bare numbers are okay here (barely) but in real life it would be better to define some symbols with meaningful names, such as enumeration constants.) The code updating the buffer set needs to use these numbers:

```
              if ( j )
                {
                  pNorm n = cross(p1_last,p3,p2_last);
                  bset_hw03_p1 << p1_last << p3 << p2_last;
                  bset_hw03_p1 << p2_last << p3 << p4;
                  bset_hw03_p1 << n << n << n   << n << n << n;

                  if ( !i ) bset_hw03_p1 << 0 << 0 << 0 << 1 << 1 << 1;
                  else      bset_hw03_p1 << 2 << 2 << 2 << 3 << 3 << 3;
```

Next, the shader code needs to be modified to use these changes. There are two changes: first, we need to have the shader code compute two colors, front and back; second we need to use the colors from the uniform. Using the uniform colors is easy. We start by declaring the uniform object. When declaring a uniform be careful to use the same location label that we specified when binding it to the pipeline:

```
const int ncolors = 10;
layout ( binding = BIND_HW03 ) uniform I_can_forget_this_name_no_problem
{
  vec4 front[ncolors], back[ncolors];
} uc;
```

Notice that the uniform itself is named `uc` (for uniform color). Next, get the colors from `uc.front` and `uc.back` in the vertex shader routine, `vs_main`:

```
  vec3 vec_vl = uni_light.position.xyz - vertex_e.xyz;
  float dist_to_light = length( vec_vl );
  float phase = abs( dot( normal_e, vec_vl/dist_to_light ) );

  color = uni_light.color * uc.front[in_color_idx] * phase / dist_to_light;
  // color = uni_light.color * in_color * phase / dist_to_light; // Before
  color_back = uni_light.color * uc.back[in_color_idx] * phase / dist_to_light;
```

The code using the pre-solution color attribute, `in_color`, is commented out. Notice that `in_color_idx` is used to index both the front and back colors.

Finally, we need to carry both the front and back lighted color through the rendering pipeline and modify the fragment shader to use the appropriate color. Add `color_back` to the shader interfaces:

```
 // Vertex Shader Output
layout (location = 0) out Data_VF
{
  vec4 color;
  vec4 color_back;
};
```

Further below, the fragment shader input:

```
 // Fragment Shader Input
layout (location = 0) in Data_VF
{
  vec4 color;
  vec4 color_back;
};
```

The change to the fragment shader is simple, since it was already coded to use a separate front and back color, but used the same color for both:

```
void fs_main()
{
  frag_color = gl_FrontFacing ? color : color_back;
}
```

It might seem wasteful to send both the front and back colors down the pipeline when only one of them will be used. The problem is that the vertex shader can't tell which side of the primitive is facing the eye because it only has access to one vertex. If the normal always pointed out of the front face then the vertex shader could substitute the correct side. But historically (meaning compatibility profile OpenGL) did not require that a normal point out of the front face, or that it even be used at all.

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.

First, populate the buffer set for this problem, `bset_hw03_p2`, with the with the vertices for just one turbine. That turbine will be left in its local coordinate space. That is the global-from-local transformation, `m` from `fans_xforms`, won't be applied, nor will the animation angle `theta_0`. For reference, here is part of the code for `PV_HW03_P1`:

```
    // Code for Problem 1 (not this problem)
    bset_hw03_p1.reset(pipe_hw03);

    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 );
```

For this problem the `m` loop will be omitted, `theta` will be computed without `theta_0`, and (of course) `m` will not be used to transform coordinates to global space:

```
      if ( bset_p2_serial != fan_info.serial ) {
          bset_p2_serial = fan_info.serial;
          bset_hw03_p2.reset(pipe_hw03);

          for ( int i=0; i<f.n_blades; i++ ) {
              float theta = 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 = pa + r1 * v;
                  pCoor p4 = pa + f.outer_radius * v;
```

The code above lacks the `m` loop, but does include a new `if` statement (the first line in the fragment above). That `if` condition insures that the buffer set is only updated if it does not contain the latest data. As described in the Assignment Background section, variable `fan_info.serial` is incremented whenever the fan changes (for example, after pressing 1 or 2). In order to determine when `bset_hw03_p2` is *stale* (outdated) a new variable has been added, `World::bset_p2_serial`. It is initialized to zero and then, as can be seen in the fragment above, is updated whenever `bset_hw03_p2` is updated.

Another important thing to notice about the code above is that `bset_hw03_p2` is only reset and sent `to_dev` when the buffer needs is updated. (Once it is reset, the data is lost, so if it were reset every frame nothing would appear after the first frame.) Calling `to_dev` each frame would defeat the purpose of only updating it when it changes. That is, we'd be sending data from the CPU to the GPU that the GPU already has.

Finally, the transformation implementing the turbine rotation needs to be applied. The unmodified code already applies the global-from-local transform:

```
    // Code without the solution to Problem 2.
    for ( auto& m: fans_xforms ) {
        pMatrix global_from_local = m; // Need to modify this for Problem 2
        pipe_hw03.ds_set( transform * global_from_local );
        pipe_hw03.record_draw(cb, bset_hw03_p2);
      }
```

Variable `theta_0` specifies the amount of rotation (the animation rotation) and changes each frame. Since we don't want to resend `bset_hw03_p2` every frame `theta_0` can't be used to compute the coordinates in `bset_hw03_p2`. Instead, a rotation matrix will be computed and that will be used to update the set of transformation matrices used by the shaders. The shaders use the following uniforms to hold the transformations:

```
layout ( binding = BIND_TRANSFORM ) uniform Uni_Transform
{
  mat4 eye_from_object, clip_from_eye, clip_from_object;
  mat4 object_from_eye, eye_from_clip;
} ut;
```

Here they are part of the file `hw03-shdr.cc`, but they could have been included from `transform.h`. These matrices are updated by the `VTransform` class (`transform` object) that is part of the course library. Typically `transform` holds a eye-from-global and clip-from-eye transformation. A global-from-local transform can be specified for a particular pipeline by calling the `pipe.ds_set( transform * global_from_local )` member function.

The unmodified Problem 2 code already has a loop calling this function using `m` as the global-from-local transform. But `m` does not include the animation realized by `theta_0`. So to include that effect we need to compute a rotation matrix that rotates `theta_0` radians around the `az` axis, and then use it to update the global-from-local transform:

```
    pMatrix_Rotation rot(az,theta_0);
    for ( auto& m: fans_xforms ) {
        pipe_hw03.ds_set( transform * m * rot );
        pipe_hw03.record_draw(cb, bset_hw03_p2);
      }
```

Matrix `rot` rotates `theta_0` around `az`, which is the z axis in the turbine local coordinate space (the space used to construct the turbine). It is important to apply `rot` before `m`. That is had we called `.ds_set( transform * rot * m );` then the rotation would be done in the global space and would be all wrong.

Ideally, with the solution above the Problem 2 pipeline should run much faster. It does not (as of this writing) probably due to avoidable inefficiencies in updating the descriptor set.