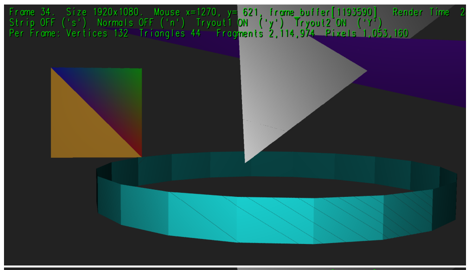
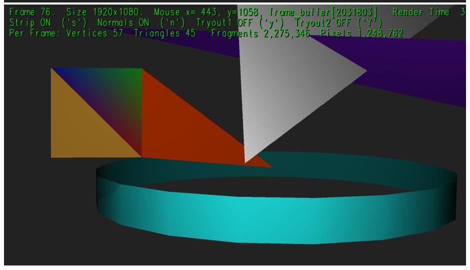
LSU EE 4702-1 Homework 3 Due: 11 October 2023

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/2023/hw03.cc.html`*. The solution code is at* `https://www.ece.lsu.edu/koppel/gpup/2023/hw03-sol.cc.html`*.*

Problem 0: If 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 code is based on the CPU-only rendering pipeline demo in directory `cpu-only/demo-06-rend-pipe.cc`. The homework code should initially show a scene with a ring and some triangles, see the screenshot to the upper right. The screenshot on the lower right is from a correct solution. There is a new red triangle (Problem 1), the scene is rendered using fewer vertices (Problem 2), and the ring looks rounder (Problem 3).



```
Frame 34.  Size 1920x1080.  Mouse x=1270, y= 621, frame_buffer[1193590]   Render Time  2
Strip OFF ('s')  Normals OFF ('n')  Tryout1 ON  ('y')  Tryout2 ON  ('Y')
Per Frame: Vertices 132  Triangles 44   Fragments 2,114,974  Pixels 1,053,160
```



```
Frame 76.  Size 1920x1080.  Mouse x= 443, y=1058, frame_buffer[2031803]   Render Time  3
Strip ON  ('s')  Normals ON  ('n')  Tryout1 OFF ('y')  Tryout2 OFF ('Y')
Per Frame: Vertices 57  Triangles 45   Fragments 2,275,346  Pixels 1,248,762
```

Homework Overview

To solve this assignment one will have to consider the same issues considered in the design of 3D acceleration libraries like OpenGL and Vulkan. The homework code contains to major classes, `World` and `Our_3D`. Code in `Our_3D` is part of a fictional 3D rendering library, the kind of code that an OpenGL or Vulkan implementation would provide. The code in `World` is application code that makes use of the 3D acceleration library. It does so by passing arrays of coordinates and other data to `Our_3D` and then calling `Our_3D::draw_rasterization` to write the frame buffer based on those coordinates. In this assignment `World` draws a simple scene, but it is supposed to represent applications such as flight simulators, video games, or visualization. The code in `Our_3D` represents code provided as part of a device driver that came with your video card or operating system. It would be written to run fast on a particular piece of hardware, such as a video card using an RTX 4090 GPU. In this assignment though the code is not even close to tuned to any hardware device.

Routine `World::render_scene` prepares a scene to be rendered. In the unmodified assignment it consists of a ring plus four additional triangles. The routine fills `coors_os` (coordinates in object space) with the coordinates of each vertex of each triangle. A second container, `colors`, holds the color of each vertex. For example, the code below inserts coordinates and colors for two triangles (note that `<<` inserts an item, it is an overload of `push_back`):

```
vector<pCoor> coors_os;
vector<pColor> colors;

coors_os << pCoor( 0, 0, 0 ) << pCoor( 9, 6, -5 ) << pCoor( 0, 7, -3 );
colors << color_white << color_white << color_white;

coors_os << pCoor(-4,2,-3) << pCoor(-2,2,-3) << pCoor(-2,0,-3);
colors << color_blue << color_green << color_red;
```

(The code samples here are condensed versions of the code in `hw03.cc`.) Note that `coors_os` is declared in `World::render_scene` and so `Our_3D` can't automatically see these coordinates. `World::render_scene` also provides transformation matrices to `Our_3D`:

```
// Specify Transformation from Object to Eye Space: eye_from_object.
pMatrix_Translate center_eye(-eye_location);
pMatrix_Rotation rotate_eye(eye_direction,pVect(0,0,-1));
pMatrix eye_from_object = rotate_eye * center_eye;
gc.transform_eye_from_object_set(eye_from_object);

// Transform from Eye to Clip
pMatrix_Frustum clip_from_eye
  ( -width_m/2, width_m/2,  -height_m/2, height_m/2,  1, 5000 );
gc.transform_clip_from_eye_set(clip_from_eye);
```

To actually render the scene the coordinate and color container information needs to be passed to `Our_3D` and then a *draw* command needs to be issued. That is done by the code below:

```
gc.vtx_coors_set(coors_os);  // Copy address of coors_os
gc.vtx_colors_set(colors);   // Copy address of colors.
gc.draw_rasterization();     // Render the scene.
```

Note that `Our_3D` does not copy the containers, it only keeps an address, so additional items can be added before `gc.draw_rasterization()` is called.

In the unmodified code `gc.draw_rasterization()` is called once per frame. But it could be called multiple times, each time with some subset of triangles. For example:

```
coors_os << pCoor( 0, 0, 0 ) << pCoor( 9, 6, -5 ) << pCoor( 0, 7, -3 );
colors << color_white << color_white << color_white;

// WARNING: Example of bad code.  Better to call draw_rasterization fewer times.
// Draw the one triangle above. Just one.
gc.vtx_coors_set(coors_os).vtx_colors_set(colors).draw_rasterization();
coors_os.clear(); colors.clear();

coors_os << pCoor(7,2,1) << pCoor(-3,3,-3.5) << pCoor(9,0,0);
colors << color_lsu_spirit_purple << color_lsu_spirit_purple << color_lsu_spirit_purple;

// WARNING: Example of bad code.  Better to call draw_rasterization fewer times.
// Draw the one triangle above. Just one.
gc.vtx_coors_set(coors_os).vtx_colors_set(colors).draw_rasterization();
coors_os.clear(); colors.clear();

coors_os << pCoor(-4,0,-3) << pCoor(-4,2,-3) << pCoor(-2,0,-3);
colors << color_lsu_spirit_gold << color_lsu_spirit_gold
       << color_lsu_spirit_gold;

coors_os << pCoor(-4,2,-3) << pCoor(-2,2,-3) << pCoor(-2,0,-3);
colors << color_blue << color_green << color_red;

// WARNING: Example of bad code.  Better to call draw_rasterization fewer times.
// Draw the two triangles above. Just two. Still wasteful.
```

```
  gc.vtx_coors_set(coors_os).vtx_colors_set(colors).draw_rasterization();
  coors_os.clear(); colors.clear();
```

The code above is bad because it calls `draw_rasterization` multiple times for no reason. A good reason for calling `draw_rasterization` multiple times is that something needed to be changed, such as vertex grouping (Problem 2) or provision of normals (Problem 3).

Problem 1 can be solved by using `Our_3D` as described above. But for Problems 2 and 3 new functionality will be added to `Our_3D`. In Problem 2 a second way of grouping vertices to triangles, *triangle strips*, will be added to `Our_3D`. In Problem 3 the option of providing per-vertex normals will be added.

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. The eye direction cannot be changed, but feel free to add this functionality if you want it. After pressing `l` the motion keys will move the light instead of the eye, and after pressing `e` the motion keys operate on the eye.

Assignment-Specific User Interface

Pressing `s` toggles between rendering using triangle strips (`Strip ON` in the green text) and individual triangles (`Strip OFF` in the green text). Before Problem 2 is solved switching to triangle-strip mode will show only 1/3 of the ring's triangles. When Problem 2 is solved correctly switching between triangle strip and individual triangle rendering will have no visual impact (though fewer vertices will be sent down the pipeline).

After Problem 3 is solved correctly pressing `n` toggles between rendering using triangle normals (`Normals OFF` in the green text) versus application-provided normals (`Normals ON` in the green text).

Display of Performance-Related Data

The top green text line shows performance-related and other information. `Size` refers to the size of the window. `Mouse` refers to the coordinates of the mouse pointer. Coordinate $(0, 0)$ is at the **lower** left of the window. Text `frame_buffer[N]` shows the index of the frame buffer corresponding to the point under the mouse pointer. (In the assignment file `frame_buffer` is abbreviated to `fb`, for convenience.)

`Render Time` and `Potential Frame Rate` show the CPU time needed to write the frame buffer. They can be ignored for this assignment. See Problem 2 for the second line, the one that starts `Frame buffer writes.`.

Don't overlook the last line of green text, which shows the number of items processed by the rendering pipeline per frame. One should be able to estimate these numbers. For example, if one is rendering using individual triangles one should be able to compute the number of triangles from the number of vertices. `Fragments` shows the number of fragments processed by what we will call the rasterizer. Recall that a fragment is a part of a triangle covering a pixel. In `draw_rasterization` look for the place where `n_frag_frame` is incremented. `Pixels` is the number of times that the frame buffer is written with a fragment. (Not every fragment is written.)

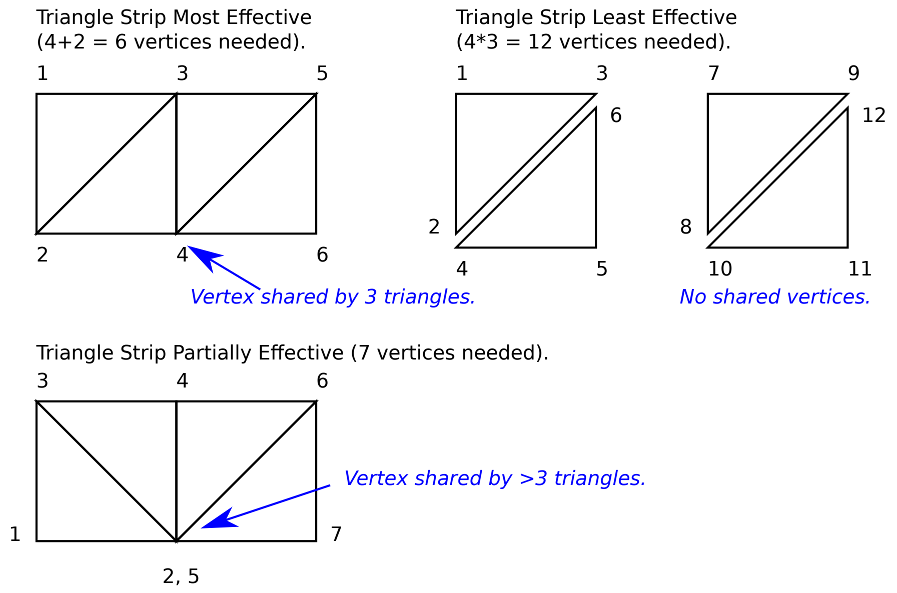
Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw03`, and a debug-able version of the code, `hw03-debug`. The `hw03-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, 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. **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` and `opt_tryout2`. 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 host Boolean variables `opt_tryout1` and `opt_tryout2`.

Problem 1: *Warning: This is an easy problem, and will be awarded at most 2 points.* For this problem modify `World::render_scene()` so that a red triangle appears in scene as follows: two vertices should connect to the green and red vertices of the multicolored triangle. The third vertex should be on the ring's axis at the height of the top of the ring. See the screenshot at the beginning of the assignment.

Problem 2: In OpenGL and Vulkan a *triangle strip* is an arrangement of n vertices numbered $v_1, v_2, \ldots, v_n$, describing $n-2$ triangles. Triangle $0 < i \leq n-2$ is constructed using vertices v_i, v_{i+1}, v_{i+2}. (This is not important for this problem, but when i is even the order is v_{i+2}, v_{i+1}, v_i.) Note that a triangle strip consisting of three vertices is just one triangle. The illustration below shows three ways triangle strips can be used to render four triangles. The one on the upper left is best.


Triangle Strip Most Effective
(4+2 = 6 vertices needed).
1
3
5
2
4
6
Vertex shared by 3 triangles.
Triangle Strip Least Effective
(4*3 = 12 vertices needed).
1
3
6
2
4
5
7
9
12
8
10
11
No shared vertices.
Triangle Strip Partially Effective (7 vertices needed).
3
4
6
1
7
2, 5
Vertex shared by >3 triangles.

.

Modify `Our3D::draw_rasterization` so that optionally the coordinates passed by `Our_3D::vtx_coors_set(COORDS)` are interpreted as triangle strips rather than individual triangles. To solve this problem one must devise a way of telling `Our_3D` whether to expect a triangle strip or individual triangles. That could be another call such as `Our_3D::triangle_arrangement_set`. Also, modify `Our_3D::draw_raseterization` to honor the setting.

When `World::opt_strip` is true the code constructing the ring inserts coordinates in triangle-strip order. (That's pre-solved, already done.) Use your new API to render the ring using triangle strips. The other triangles can still be rendered as individual triangles, perhaps by using a separate `draw_rasterization` call for the ring.

Pay attention to the number of vertices and triangles displaced in the green text when switching between the two modes. The number of triangles should not change, but the number of vertices should drop from 132 (or 135) to 54 (or 57) when triangle strips is turned on.

Do not expect a performance improvement. Though the size of the coordinate and color arrays will be much smaller, the number of fragments should not change, and that is where most of the work is done.

Problem 3: `Our_3D` is a bit unusual in that it computes triangle normals. In classic OpenGL the user code provides a normal for each vertex. (In modern OpenGL and Vulkan it is up to user-written shaders to compute lighting. Whether they do so by computing triangle normal themselves or reading a normal streamed in by the user is up to the writer of the shader, not OpenGL or Vulkan.)

Modify `World` and `Our_3D` so that when `opt_normal` is true `render_scene` passes normals to `Our_3D`, and `Our_3D` uses these normals. The normals for the ring should be computed based on the shape of the ring. (Those vectors are already computed.) (Just do this for the ring.) The rasterization loop in `Our_3D` should use these normals (when provided) for lighting. As it already does with colors and object-space coordinates, the rasterization loop should compute a normal for a fragment by blending the normals from the three vertices. When solved correctly, the ring will have a smoother appearance, as in the lower screenshot at the beginning of the assignment.

As in the previous problem, one needs to devise a mechanism to pass normals to `Our_3D`, and `Our_3D` must be able to work without normals as it does in the unmodified assignment.