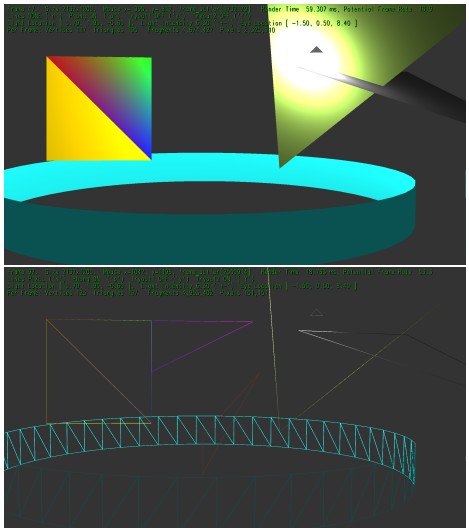
LSU EE 4702-1 **Homework 3** **Due: 11 October 2024**

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/2024/hw03.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` and on the solution to 2023 Homework 3. The homework code should initially show a scene with a ring (or squat cylinder) and some triangles, one of which looks like it is dangerously close to a bright light, see the screenshot to the upper right. The screenshot on the lower right shows a correct solution to this assignment, in particular, Problem 3 in which only triangle edges are rendered, shown as pixel-width lines.



Homework Overview

The code in the this assignment takes the frame buffer demo code, such as `demo-04-z-light`, and re-organizes it into a form closer to that of 3D acceleration libraries by separating the code that computes the location of triangles and the location of the eye, from the code rendering triangles. The homework code contains two 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 plus *shaders* provided by application programs. The code in `World` is application code that makes use of the 3D rendering library.

It does so by passing arrays of coordinates and other data to `Our_3D` and then by calling routine `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 plus any application-programmer written shaders. The device-driver code 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 several triangles and a ring (a short cylinder), which is itself made from triangles. The triangle vertex locations are specified by inserting coordinates into container `coors_os`, vertex normals into `norms_os`, and vertex colors (material properties) into `colors`. The suffix `_os` is for object space.

For example, the code below inserts coordinates and colors for two triangles (note that `<<` inserts an item, it is an overload of `vector::push_back`):

```
vector<pCoor> coors_os;
vector<pVect4> norms_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`.) The vertex normals for the triangles are computed by a loop further below. 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);
```

After populating the `coors_os` container our user code computes the triangle normals:

```
for ( size_t i = 0;  i < coors_os.size();  i += 3 ) {
    pCoor p1 = coors_os[i], p2 = coors_os[i+1], p3 = coors_os[i+2];
    pVect4 n = cross( p1, p2, p3 ).normal();
    norms_os << n << n << n;  }
```

Notice that each group of three vertices describes a triangle. That's the case for the triangles specified by the code so far, but not for the ring. 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.topology_strip_set(false);  // Triangles *do not* form a strip.
gc.vtx_coors_set(coors_os);    // Copy address of coors_os.
gc.vtx_normals_set(norms_os);  // Copy address of norms_os.
gc.vtx_colors_set(colors);     // Copy address of colors.
gc.draw_rasterization();       // Render the scene.
```

`Our_3D` does not copy the containers, it only keeps an address, so additional items can be added up until `gc.draw_rasterization()` is called. Routine `gc.draw_rasterization()` paints the frame buffer (actually a pixel array which will eventually be composited into the frame buffer).

Routine `gc.draw_rasterization()` is called twice per frame, the first time to render the individual triangles, and the second time for the ring. It is possible to call `gc.draw_rasterization()` once per triangle, but that would be foolish because there is a substantial per-call overhead that would have to be borne for each triangle. So the code loads all the triangles into `coors_os` and calls `draw_rasterization` once. Calling draw routines fewer times with more triangles per call is especially important for "real" 3D acceleration libraries like OpenGL and Vulkan.

After preparing the individual triangles, the code prepares the ring (cylinder). The ring is described by variables declared near the top of the routine:

```
pCoor cyl_center(-1,-1,-3);
int n_segs = 50;
pVect cyl_axis( 0, 0.75, 0.05 );
float cyl_radius = 4;
```

Further ahead in the routine code inserts triangles for the ring. It starts by clearing (emptying) the containers and then computing axes, `ax` and `ay`, of the ring circles.

```
// Empty the containers so they can be used for the next draw.
coors_os.clear(); colors.clear(); norms_os.clear();

pNorm ax(0,-cyl_axis.z,cyl_axis.y);  // Find an axis orthogonal to cyl_axis.
pNorm ay = cross( cyl_axis, ax );
float delta_theta = 2 * M_PI / n_segs;
```

For the individual triangles above (before the ring) three vertices were inserted into `coors_os` for each triangle, this is referred to as an *individual triangle grouping*. For the ring we will tell `Our3D` to expect the vertices to be in a *triangle strip grouping*. In a triangle strip grouping the first triangle is constructed from `coors_os[0]`, `coors_os[1]`, and `coors_os[2]`; the second triangle from `coors_os[3]`, `coors_os[2]`, and `coors_os[1]`, the third from `coors_os[2]`, `coors_os[3]`, and `coors_os[4]`, etc. To render T triangles requires $T + 2$ vertices rather than $3T$. (Sharp-eyed readers may have noticed that the order of vertices of the first and second triangle are different. That's intentional since vertex order determines which side is the front. The default is a counter-clockwise [right-handed] winding order.)

A triangle strip grouping works well for the ring. A loop iterates around the circle, with the first and last point overlapping. At each iteration a point on the lower circle, `c`, and upper circle, `c+cyl_axis` are computed and added to the `coors_os` container. Both of these points have the same surface normal, `n` in the code. That is added to `norms_os`. Note that `n` is the normal of the cylinder surface, not of the triangles used to approximate the cylinder. Code in the loop body also adds colors to the list.

```
for ( int i=0; i<=n_segs; i++ ) {
    float theta = i * delta_theta;
    pVect4 n = ax * cosf(theta) + ay * sinf(theta);
    pCoor c = cyl_center + cyl_radius * n;

    coors_os << c + cyl_axis << c;
    norms_os << n << n;
    colors << color_cyan << color_cyan;
  }
```

Finally, `Our3D` is told to expect a triangle-strip grouping, pointers to the containers are provided, and the draw routine is called:

```
gc.topology_strip_set(true);
gc.vtx_normals_set(norms_os);
gc.vtx_coors_set(coors_os).vtx_colors_set(colors);
gc.draw_rasterization();
```

In this assignment code in `Our_3D::draw_rasterization` will also have to be modified. This routine uses the vertex containers and transformations provided by `Our_3D::draw_rasterization`

to write the frame buffer. The code itself is based on code from `cpu-only` demos such as `demo-04-z-light`, but is re-organized to look a tiny bit more like acceleration code in a Vulkan implementation. Some of the code in `Our_3D::draw_rasterization` would be driver code in a Vulkan implementation (not normally seen by an application program) and some would be *shader code*, something application programs do write.

The code in `Our_3D::draw_rasterization` starts by preparing the window-from-clip coordinate space transformation:

```
Our_3D& Our_3D::draw_rasterization() {
  const uint win_width = frame_buffer.width_get();
  const uint win_height = frame_buffer.height_get();
  // Transform from Clip to Pixel
  //
  pMatrix_Translate recenter(pVect(1,1,0));
  pMatrix_Scale scale( win_width/2, win_height/2, 1 );
  pMatrix window_from_clip = scale * recenter;
  pMatrix ws_from_os = window_from_clip * clip_from_eye * eye_from_object;
  pCoor eye_location = -eye_from_object.col_get(3);
```

Notice that `clip_from_eye` and `eye_from_object` were prepared by application code, but that `ws_from_os` (window-space from object space) is prepared by `Our3D`. That's because `clip_from_eye` and `eye_from_object` depend on things that are of a concern to the application programmer, such as where the eye is and how big the monitor is.

After preparing convenience pointers like `zbuffer` and `colors` and updating statistics, like `frame_buffer.n_vtx_frame` (not shown here), the code starts the main loop which iterates over triangles:

```
  const size_t inc = topology_strip ? 1 : 3;
  for ( size_t i=0; i+2 < coors_os.size(); i += inc ) {
      frame_buffer.n_tri_frame++; // Count of number of triangles.
      // Get next triangle's object space coordinates ..
      pCoor o0 = coors_os[i+0],  o1 = coors_os[i+1],  o2 = coors_os[i+2];
      // .. convert them into unhomogenized clip-space coordinates ..
      pCoor u0( ws_from_os * o0 ), u1( ws_from_os * o1 ), u2( ws_from_os * o2 );
      // .. and homogenize them.
      pCoor_Homogenized w0( u0 ), w1( u1 ), w2( u2 );

      //  Extract colors and normals too.
      pColor c0 = colors[i], c1 = colors[i+1], c2 = colors[i+2];
      pVect4 n0 = vtx_normals[i], n1 = vtx_normals[i+1], n2 = vtx_normals[i+2];
```

The loop body starts by retrieving information about a triangle, which is described by three consecutive elements of `coors_os`, etc. Note that this loop works both with individual triangle and triangle strip groupings, the only difference is the value of `inc`. The retrieved object-space coordinates, `o0`,`o1`,`o2`, are converted to window-space coordinates in two steps. Coordinates in `w0`, `w1`, and `w2` are in window space, meaning that, for example, `w0.x` and `w0.y` could be used to compute a frame-buffer coordinate.

A deeper loop nest (`b0`,`b1`) iterates over barycentric coordinates. Recall that given barycentric coordinates b_0, b_1, and b_2, and points P_0, P_1, and P_2, point $S = b_0P_0 + b_1P_1 + b_2P_2$ is inside triangle $P_0P_1P_2$ if $b_0 \geq 0$, $b_1 \geq 0$, $b_2 \geq 0$, and $b_0 + b_1 + b_2 = 1$. Here P_0 is the coordinate of a point, but mathematically is treated like a vector from the origin to the point. (In class barycentric

coordinates were described in a different but equivalent way: $S = b_1\overrightarrow{P_0P_1} + b_2\overrightarrow{P_0P_2}$.) Some special cases: If $b_i = 1$ then $S = P_i$ for $i \in \{0, 1, 2\}$. If $b_0 = 0$ then S is on the edge between points P_1 and P_2. *That may be useful for 2024 Problem 3.*

Since we aren't interested in points outside the triangle the inner loop limits $b_1 \in [0, 1 - b_0]$ and $b_2 = 1 - b_0 - b_1$. See the code below.

```
for ( float b0=0; b0<=1; b0 += db0 )
  for ( float b1=0; b1<=1-b0; b1 += db1 ) {
      frame_buffer.n_frag_frame++; // Count of number of fragments.
      const float b2 = 1 - b0 - b1;
      pCoor wf = b0*w0 + b1*w1 + b2*w2;
      if ( uint(wf.x) >= win_width || uint(wf.y) >= win_height ) continue;
      const size_t idx = wf.x + int(wf.y) * win_width;
```

Since the goal is to find a frame-buffer coordinate the barycentric coordinates are used in the code above to compute `wf` (window-space fragment) using window-space coordinates of the triangle vertices. After computing `wf` the code makes sure it is within the frame buffer.

To compute the lighted color of the fragment the code needs to find the object-space coordinates of the fragment. That **can not** be computed by the expression `b0*o0 + b1*o1 + b2*o2` because `b0`, `b1`, and `b2` are window-space barycentric coordinates. To find the fragment object-space coordinate and to interpolate any other vertex properties we need object-space (perspective-correct) barycentric coordinates. These are computed and used to interpolate the object-space coordinates, vertex normal, and color:

```
// Compute perspective-correct barycentric coordinates.
float bc0 = b0 / ( b2 * u0ou2 + b1 * u0ou1 + b0 );
float bc1 = b1 / ( b2 * u1ou2 + b0 * u1ou0 + b1 );
float bc2 = 1 - bc0 - bc1;

pCoor of =     bc0*o0 + bc1*o1 + bc2*o2;
pNorm n =      bc0*n0 + bc1*n1 + bc2*n2;
pColor color = bc0*c0 + bc1*c1 + bc2*c2;
```

For this assignment it is not important to understand the derivation of the object-space barycentric coordinates, however it is important to understand the difference between object-space and window-space coordinates.

With these, the lighted color can be computed and written to the frame buffer:

```
// Compute Lighted Color
pNorm frag_to_light(of,light_location);
pVect frag_to_eye(of,eye_location);
float cos_angle_eye = dot( n, frag_to_eye );
float cos_angle_light = dot( n, frag_to_light );
bool illumination_visible =
   ( cos_angle_light > 0 )  ==  ( cos_angle_eye > 0 );
float clamped_phase = illumination_visible ? fabs(cos_angle_light) : 0;
pColor lighted_color =
  ( light_ambient
    + clamped_phase / frag_to_light.magnitude * light_color ) * color;
// Write the frame (color) buffer with the lighted color
frame_buffer[ idx ] = lighted_color.int_rgb();
```

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-`x`.png` or `hw03-debug-`x`.png`, where x is the number of previous screenshots generated. The first screenshot may be garbled.

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.

The light intensity can be adjusted by pressing `+` and `-`. (This only works for this assignment.)

Assignment-Specific User Interface

Pressing `p` toggles between computing the lighted color per fragment or per vertex. The setting is shown in the second line of green text to the right of label Lighting per. The value is shown in variable `opt_light_frag`, set to `true` for per-fragment lighting.

Pressing `n` cycles through three ways of display triangle edges NONE, meaning the triangles are shown in the usual way, PIXEL, meaning only triangle edges are to be shown and that the edges are a few pixels wide, and WORLD, meaning that triangle edges are to be 0.1 object-space units wide. Variable `opt_lines` is cycled by `n`, see Problem 3 for more information. Until Problem 3 is solved correctly only NONE will work. (Of course, it's possible even that won't work. If that happens to you don't feel shy about asking for help.)

Display of Performance-Related Data

The top green text line shows performance-related and other information. `Frame` shows the number of times `render_scene` has been called. It should only increment on key presses and certain window manager events (such as minimizing and unminimizing). `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.

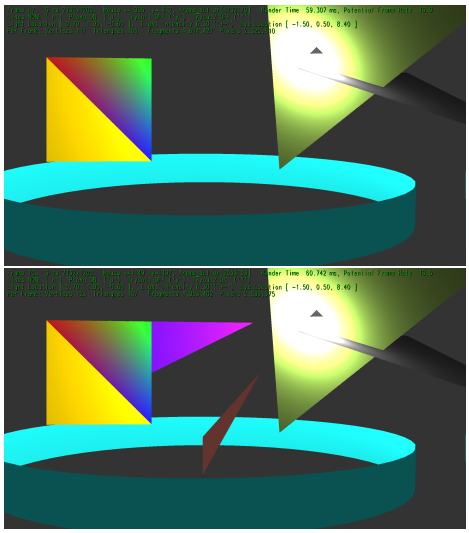
The last line of green text shows the number of items processed by the rendering pipeline per frame. One should be able to estimate these numbers. For example, if one is rendering using individual triangles one should be able to compute the number of triangles from the number of vertices. `Fragments` 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: For this problem modify the code in routine `World::render_scene()` so that it emits the two triangles described below. The screenshot to the upper-right shows the unmodified assignment, the screen shot to the lower right shows the scene with the two new triangles: a purple triangle in the the center and a salmon triangle (it looks brown) in the lower center.

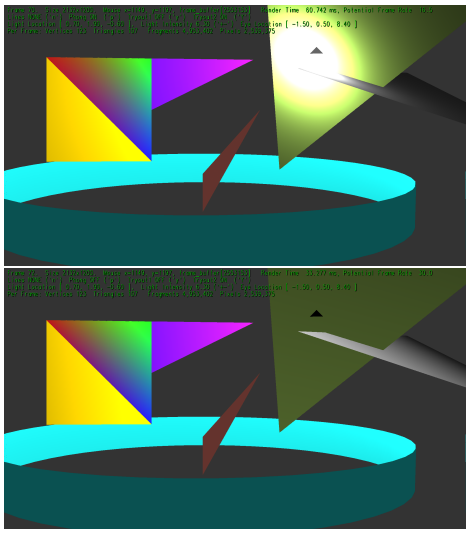


(*a*) Modify the code so it emits a purple triangle adjacent to the multicolored (red/green/blue) triangle. The top-left vertex of the purple triangle should be at the same coordinate as the green vertex of the multi-colored triangle. The lower vertex of the purple triangle should be halfway between the green and blue vertex. The upper edge of the purple triangle should be on the same line as the red/green edge and should be the same length as the red/green edge. *Note: This is an easy problem once you find the place in the code that emits the multi-colored triangle.*

(*b*) A common problem encountered by some working with graphics demos is figuring out where the light is. Here we will fix that problem by adding a triangle that points at the light: the brown (the color is actually salmon) triangle in the cyan ring.

Modify the code so that it renders the salmon triangle as follows. Two vertices are on the cylinder axis (one at the bottom and one at the top). The third vertex points toward the light. Place it on the line from the top vertex to the light. The length of the edge from the top vertex to the light should be two. Use variable `color_salmon` for the color, or choose your own color. The light location is in variable `World::light_location`.

Problem 2: When the code starts it shows a scene with the light close to a green (actually dark olive green) and a gray triangle. As with all demos covered in class so far, lighting is computed at each fragment. Back in the twentieth century the amount of computation needed to compute lighting was considered substantial. So rather than compute the lighted color for every fragment in a rasterized triangle, the lighted color would only be computed at each vertex. The lighted color at a fragment would be a blend of the lighted colors of the three vertices, using the barycentric coordinates as the blend weights. Note that the code already does such blending for color (material property) which can be seen in the multi-colored triangle.



Computing the lighted color only for vertices can save a substantial amount of computation. In our default scene (in a draft of this homework assignment) there are 57 vertices and 2141934 fragments, so our code does $\frac{2141934}{57} = 37577.8$ times more lighted color computations

than it would do if a lighted color were only computed for each vertex.

In the screenshot to the upper right a lighted color is computed for each fragment, in the screenshot to the lower right a lighted color is computed for each vertex. In some places the difference is obvious, such as the disappearance of the bright spot on the green triangle. There's no spot because lighted colors are a blend of the three vertices, and each vertex is further from the light than the center of the bright spot. The difference in lighting is much more subtle for primitives further from the light and for smaller primitives, such as the cyan ring.

(*a*) ☐ Modify the code in `Our_3D::draw_rasterization` so that when `opt_light_frag` is `false` lighting is only computed for each vertex. When `opt_light_frag` is `true` lighting should be computed the way it is now. ☐ Make sure that lighting for each vertex is computed just once, even when primitives are offered as a triangle strip.

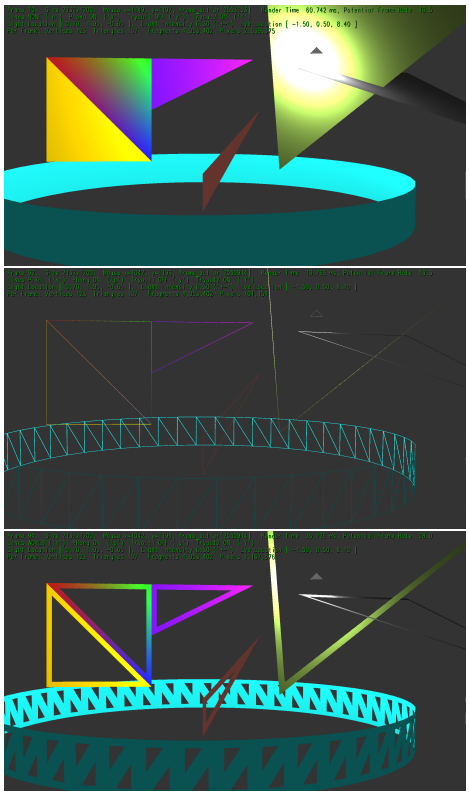
To get a better understanding of how many times lighting is computed put the line `if ( opt_tryout1 ) frame_buffer.n_vtx_frame--;` in the place where the lighted color is being computed in your code. In the unmodified assignment that would be in this code:

```
float clamped_phase =
  illumination_visible ? fabs(cos_angle_light) : 0;
pColor lighted_color =
  ( light_ambient
    + clamped_phase / frag_to_light.magnitude * light_color )
  * color;

if ( opt_tryout1 ) frame_buffer.n_vtx_frame--;  // Add this for familiarization.

//
// Write the frame (color) buffer with the lighted color
//
frame_buffer[ idx ] = lighted_color.int_rgb();
```

When running the code with this modification look at the bottom line of green text near Vertices and press `y` (lower-case wye). If you've solved the assignment correctly the text should show Vertices 0 when Tryout1 is on (look at the middle of the second line of green text). If the number is instead a large negative value, say -1573915, then the code is computing lighted colors way too many times. It is also possible that it is a much smaller magnitude negative number, say $117 - 105 \times 3 = -198$. That's better but not a full credit answer.

There's another problem on the next page.

Problem 3: For many 3D graphics systems surfaces are approximated by triangles. To help with debugging, tuning, and familiarization it can be helpful to make the triangles' locations obvious by showing only the edges. See the two lower screenshots.



Variable `opt_lines` specifies how triangles should be rendered. When `opt_lines= Lines_None` the triangles should be filled, which is how they are rendered normally, shown in the upper screenshot. When `opt_lines= Lines_Pixel` rendered triangle edges should be only about two pixels thick, see the middle screenshot. When `opt_lines= Lines_World` triangle edges should be 0.1 object-space units thick, see the lower screenshot. Note that in all cases the lighted color of the edges is computed in the same way as the full triangle. The second line of green text shows the value of `opt_lines`, and its value can by cycled by pressing `n`. (The letter el was already used for light.)

There is an important difference in appearance between pixel mode and world mode. In pixel mode the rendered thickness is the same for every triangle, regardless of its size, distance from the eye, and orientation. In contrast in world mode the closer an edge is to the eye the larger its thickness in its projection on the frame buffer. Look closely at the thickness of the edges of the triangles making up the near and far portions of the cyan ring. The closer edges are thicker (even if your perceptual system is trying to tell you otherwise). Also look at the gray triangle in the upper left. Its edges appear thinner than nearby triangles because the gray triangle is facing up and so we see the triangle (and its edges) edge-on.

Modify the code in `Our_3D::draw_rasterization` so that it renders triangle edges as described above. One might be tempted to solve the problem by using a line-drawing routine (such as one from the early frame buffer demos or past homework assignments). Try to avoid that since it can not easily be made to work for world mode once edge thickness and lighting are taken into account. Instead, use the barycentric coordinates to determine how close a fragment is to an edge.

To help you get started there is a `switch ( opt_lines )` statement with the three cases. Code in a case should test the barycentric coordinates and if some condition is true, continue. For example,

```
switch ( opt_lines ){
case Lines_None: break;
case Lines_Pixel:
  if ( bx0 ... ) continue;
  break;
```

where `bx0` is either `b0` or `bc0`. A correct solution depends on using the right barycentric coordinates in each case either the window-space coordinates, `b0,b1,b2`, or the perspective-correct coordinates, `bc0,bc1,bc2`. If necessary, re-read the material on barycentric coordinates in the homework overview above.