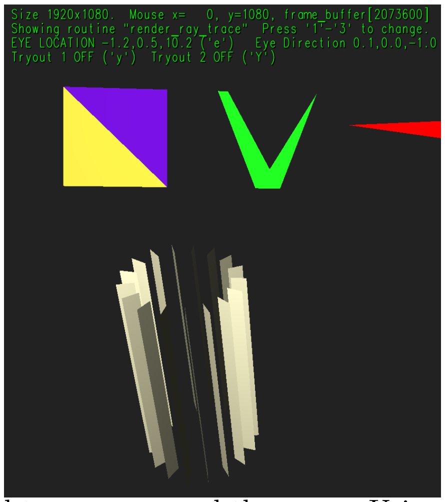
LSU EE 4702-1 Homework 2 Solution Due: 4 October 2022

Important parts of the solution are described below the complete solution code is in the repository and an html version is available at `https://www.ece.lsu.edu/koppel/gpup/2022/hw02-sol.cc.html`.



```
Size 1920x1080.  Mouse x=   0, y=1080, frame_buffer[2073600]
Showing routine "render_ray_trace"  Press '1'-'3' to change.
EYE LOCATION -1.2,0.5,10.2 ('e')   Eye Direction 0.1,0.0,-1.0
Tryout 1 OFF ('y')  Tryout 2 OFF ('Y')
```

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 triangles from the `cpu-only/demo-05-ray-tracing.cc` code used in class. See the screenshot to the right, which shows the correctly solved assignment. Solving this adds the green V and the paddle wheel (the object below the square and green V).

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 `hw02.png` or `hw02.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 is rendered by using a cpu-only ray tracing routine. (So don't complain if it's slow.) There are two ray-tracing routines, the one in use is shown in the green text to the right of `Showing routine`. Pressing `1` switches to `render_ray_trace`, this should be used for Problems 1 and 2. Pressing `2` switches to `render_ray_trace_os` (the `os` is for object space). Routine `render_ray_trace_os` won't work until Problem 3 is solved correctly.

Code Generation and Debug Support

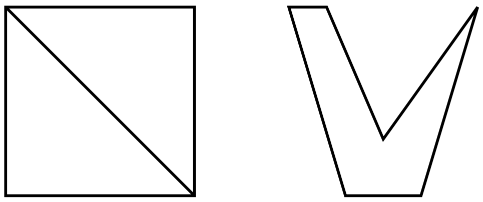
The compiler generates two versions of the code, `hw02` and `hw02-debug`. Use `hw02` to measure performance, but use `hw02-debug` for debugging. The `hw02-debug` version is compiled with optimization turned off. You are strongly encouraged to run `hw02-debug` under the GNU debugger, `gdb`. See the material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

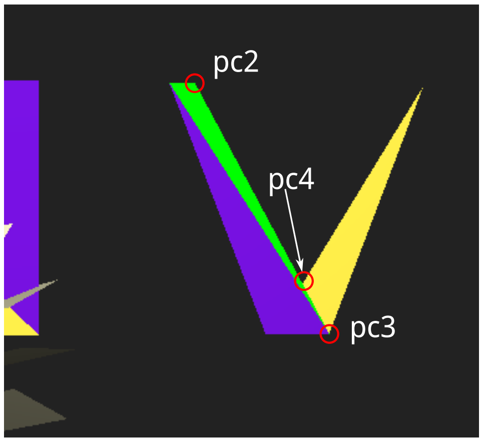
To help you debug your code and experiment in one way or another, the user interface lets you change variables. In this assignment these include the variables `hw02_info.opt_tryout1` and `hw02_info.opt_tryout2`. You can use these variables in your code (for example, `if ( hw02_info.opt_tryout1 ) { x += 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-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.`.





Problem 1: The code in routine `prep_secene` includes a fragment that draws a square (by writing the frame buffer) consisting of a purple and gold triangle. Add code beneath that that draws the V-like figure illustrated to the right. Draw the figure using only a few triangles, and do so without overlapping triangles. It might help to make each triangle a different color. The screenshot at the beginning of the assignment shows the V-like figure in green.

See 2016 Midterm Exam Problem 1 and 2018 Final Exam Problem 1 for roughly similar problems.

The solution code is shown below (along with some preceding code, and the resulting figure appears in the screenshot to the right. Notice that point `pc4` was computed so that the gold triangle does not overlap the green triangle. In most submissions the corresponding point was hand-computed and then its value placed in a `pCoor` constructor. Having the code compute it is more reliable and easier—once you get used to working with coordinates and vectors.

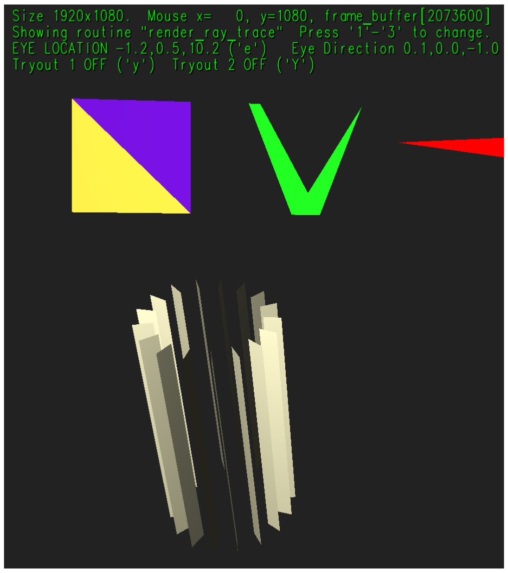
```
// Add a square consisting of a red and green triangle.
colors << color_lsu_spirit_gold << color_lsu_spirit_purple;
coors_os << pCoor(-7,0,-2) << pCoor(-7,2,-2) << pCoor(-5,0,-2);
coors_os << pCoor(-7,2,-2) << pCoor(-5,2,-2) << pCoor(-5,0,-2);

// SOLUTION -- Problem 1
pCoor pc2(-3.8,2,-2), pc3(-2.75,0,-2);
pCoor pc4 = pc3 + 0.2 * pVect(pc3,pc2);
coors_os << pCoor(-4,2,-2) << pc2                << pc3;
coors_os << pCoor(-4,2,-2) << pCoor(-3.25,0,-2) << pc3;
coors_os << pc4            << pCoor(-2,2,-2)    << pc3;
colors << color_green << color_lsu_spirit_purple << color_lsu_spirit_gold;
```

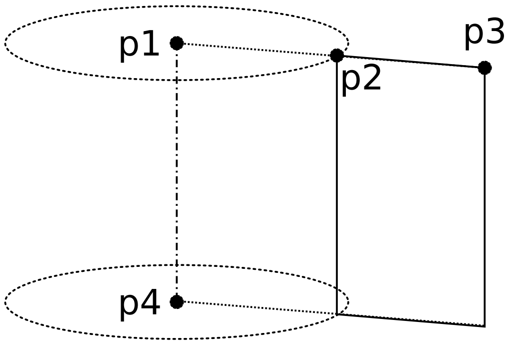
Problem 2: Modify the code in `prep_scene` below the comment reading `Problem 2` so that it renders a paddle-wheel-like object, as shown in the lower part of the screenshot to the right. The position and shape of the paddle wheel is determined by variables `p1`, `p2`, `p3`, `p4`, and `n_pieces`, see the diagram below the screenshot. Points `p1` and `p4` define the axis of a cylinder (actually two cylinders sharing the same axis but of different radii). Point `p2` is on the surface of one cylinder and point `p3` is on the surface of the other, larger, cylinder. Points `p1`, `p2`, and `p3` will always form a line, and that line is orthogonal to the cylinder axis. The first paddle wheel blade is positioned as shown in the diagram (the rectangle with solid lines). A complete paddle wheel should have `n_pieces-1` additional blades equally spaced around the cylinder axis.

Each time `1` (or `2` when Problem 3 is solved) is pressed new values for the variables will be chosen. Pressing those keys might help in debugging.

Hint: Review the code for drawing a circle from the Circles section of the math slides. For this problem there will be four circles, two at the top of the cylinder, and two at the bottom. At each iteration of an `n_pieces` *iteration loop find a point on each circle and connect them to form a blade. The illustration shows just one top circle and one bottom circle. Point* `p2` *is on the illustrated top circle, and point* `p3` *is on a top circle that's not shown.*



```
Size 1920x1080.  Mouse x=   0, y=1080, frame_buffer[2073600]
Showing routine "render_ray_trace"  Press '1'-'3' to change.
EYE LOCATION -1.2,0.5,10.2 ('e')   Eye Direction 0.1,0.0,-1.0
Tryout 1 OFF ('y')  Tryout 2 OFF ('Y')
```



The solution appears below. Note that only one sine and cosine are computed per iteration. They are used to construct a vector `v`. From `v` a longer version, `v2` is computed which requires just 3 multiplications. The only remaining operation needed to compute the four points on the blade is coordinate/vector addition.

Grading Note: Many solutions unnecessarily computed a separate `ax` and `ay` for the upper and lower circles. If the two circles have the same normal and are not rotated with respect to each other then the same `ax` and `ay` can be used for both. Also, its less expensive to compute `vx=r*ax` and `vy=r*ay` outside the loop, and then use `vx` and `vy` inside the loop.

```
  pVect vx(p1,p2), vz(p1,p4);
  pNorm ax(vx), ay( cross(vz,ax) );
  pVect vy = ay * ax.magnitude;
  float r2or1 = pNorm(p1,p3).magnitude / ax.magnitude;
  float delta_theta = 2 * M_PI / n_pieces;
  for ( int i=0; i<n_pieces; i++ ) {
      float theta = i * delta_theta;
      pVect v = vx * cosf(theta) + vy * sinf(theta);
      pVect v2 = v * r2or1;
      coors_os <<  p1 + v   <<  p1 + v2  <<  p4 + v;
      coors_os <<  p1 + v2  <<  p4 + v2  <<  p4 + v;
      colors << color_lemon_chiffon << color_lemon_chiffon;  }
```

Problem 3: Modify routine `render_ray_trace_os` so that checks for ray/triangle intercepts using object-space coordinates. This routine is used after `2` is pressed. Routine `render_ray_trace_os` starts out by computing the pixel coordinate (`px_e`) and `ray` in eye space:

```
  for (uint yw=0;  yw < win_height; yw++ ) for (uint xw=0;  xw < win_width; xw++ )
      {
        // Eye-Space Coordinate of Pixel.
        pCoor px_e = window_ll_e + window_dx_e * xw + window_dy_e * yw;

        // Ray From Eye to Pixel in Eye Space.
        pVect ray( pCoor(0,0,0), px_e );
```

But the routine iterates over the object space coordinates of triangles:

```
        for ( auto it = coors_os.begin(); it != coors_os.end(); ) {
            pCoor o0 = *it++,  o1 = *it++,  o2 = *it++;
            uint32_t color = *ic++;

            pVect tn(o0,o1,o2);  // Triangle normal.
            float t = dot( pVect(pCoor(0,0,0),o0), tn ) / dot( ray, tn ); /// WRONG!
            pCoor s = t * ray; /// WRONG!
```

This code is incorrect because the ray is in eye space but the triangle normal and vertex `o0` are in object space. Also, `s` is not computed correctly.

Fix these problems by computing a `px_o` and `ray`, both in object space. **Do not** fix this by using a transformation matrix within the loop nest. Instead directly compute a `px_o` in object space, consider using variables `window_ll_o`, and others to compute `px_o`. Also, please remove the code for `px_e`.

When this problem is correctly solved there should be no difference between rendering with `1` and `2`.

Please review the material on line/plain intercepts to help with this problem.

There are two sets of changes that need to be made: computing the pixel coordinate in object space, and modifying the parts of the code that refer to the eye location.

To compute the pixel location in object space we need the window corners in object space, which are given (`window_ll_o`, ...), and the object space `dx` and `dy` (referred to as derivatives in this context), which are not given. Like their eye-space counterparts, they are computed by dividing the object-space width and height by the window-space width and height:

```
  pVect window_dx_o = pVect( window_ll_o, window_lr_o ) / win_width;
  pVect window_dy_o = pVect( window_ll_o, window_ul_o ) / win_height;
```

This is correct because object space is mapped to eye space only by a rotation and translation.

The eye location in eye space is at the origin. In object space the eye location is kept in variable `hw02_info.eye_location`. Places that refer to the eye location implicitly, need to be updated to use the variable. There are two such places, the one which computes the `ray` vector, and one that computes the intercept. For the ray the eye-space pixel coordinate is replaced with the object-space pixel coordinate and the object-space eye location is used:

```
        // Ray From Eye to Pixel.
        // pVect ray( pCoor(0,0,0), px_e );  // <- Before change.
        pVect ray( eye_location, px_o );     // <- After change.
```

(In the original assignment the ray was confusingly computed using `pVect ray(px_e)`*. This form of the* `pVect` *constructor in effect computes* `ray = px_e - pCoor(0,0,0)`*, which forms a vector as a difference between two coordinates. Of course, computationally subtracting zero does nothing and the constructor doesn't try to subtract zero. The problem is that beginners looking at* `pVect ray(px_e)` *might start to forget the difference between a vector and coordinate.)*

The code computing the ray triangle intercept computes a vector from a triangle vertex, `o0`, and the eye. That too needs the eye-space eye coordinate, `pCoor(0,0,0)`, replaced with the variable:

```
  // float t = dot( pVect( pCoor(0,0,0), o0 ), tn ) / dot( ray, tn ); // <- Beffore
 float t = dot( pVect( eye_location, o0 ), tn ) / dot( ray, tn );
```

Finally, the intercept point needs to be computed from the `eye_location`:

```
 pCoor s = eye_location + t * ray;
```