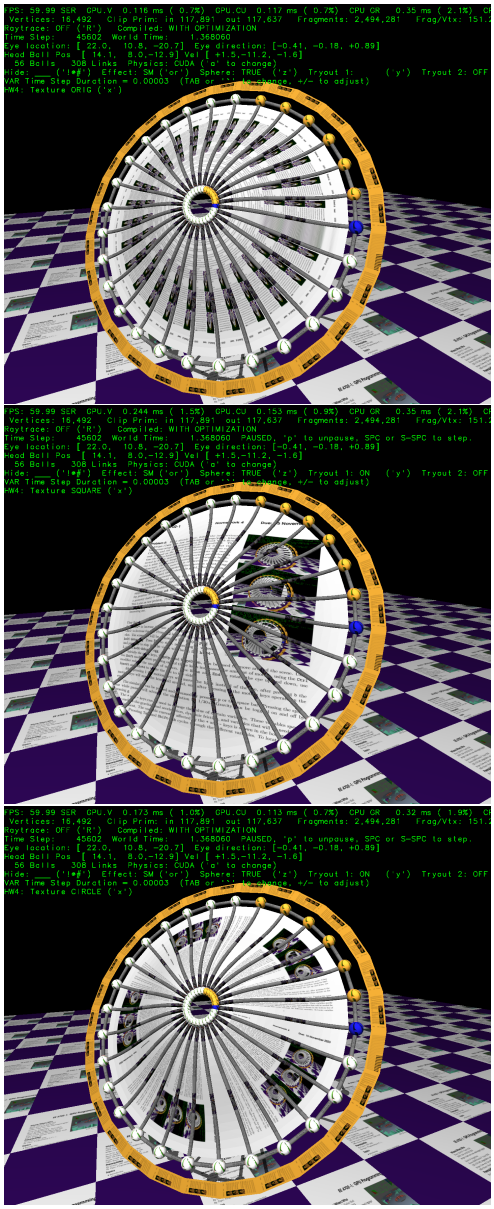


Problem 0: If not already done, follow the instructions on `https://www.ece.lsu.edu/koppel/gpup/proc.html` for account setup and programming homework work flow. Compile and run the homework code unmodified. The code is based on the wheel from Homework 2, but this time the spokes between the inner and outer ring are connected by flexible material. In the screenshots the material is white with an image of the first page of this homework assignment handout projected in various ways. The material is simulated as triangles interacting with air, and the material is rendered as a surface following the shape of the spokes. Also there is a flat ring surrounding the wheel, that is shown in gold, also showing this homework in the screenshots.

In Problem 1 the gold ring is to be drawn. This is a straightforward triangle strip drawing problem. It's supposed to be the confidence builder so if you're having problems on it **please ask for help!** In Problem 2 a geometry shader is used to compute surface normals. This is also an easy problem. Problem 3a, finding texture coordinates, is of medium difficulty. Problem 3b is the interesting one.

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 `hw04.png` or `hw04-debug.png`. Press `F10` to start recording a video, and press `F10` to stop it. The video will be in file `hw04-1.ogg` or `hw04-debug-1.ogg`.

Initially the arrow keys, `PageUp`, and `PageDown`, can be used to move around the scene. Using the `Shift` modifier when pressing one of these keys 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.

After pressing `l` the motion keys will move the light instead of the eye, after pressing `b` the motion keys will move the head ball around, and after pressing `e` the motion keys operate on the eye.

The simulation can be paused and resumed by pressing `p` or the space bar. Pressing the space bar while paused will advance the simulation by 1/30 s. Gravity can be toggled on and off by pressing `g`.

The `+` and `-` keys can be used to change the value of certain variables. These variables specify things such as the gravitational acceleration, dynamic friction, 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 and Shift-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-Specific User Interface

The code can display several scenes, numbered 1 through 5. For this assignment use scene 4 (which is how the code starts). Press 1 to select scene 1 (the silly tree), 2 for scene 2 (I'm going to ask GPT-4 to name it when it can read images), 3 for scene 3 (a crude corona virus particle), 4 for this homework assignment (the wheel), 5 for a top (a spinning child's toy). Scene 1 is initialized by the code in World::ball_setup_1, etc. Parts of the solution to this assignment should be put in World::ball_setup_4.

The scene for this assignment, 4, initializes randomly each time it is reset. (That is to avoid solutions that accidentally only work for a special case of positioning, etc.)

In this assignment the projection of textures on to the spoke material can be changed by pressing x. There are four possible projections NONE, meaning no textures, ORIG, meaning the one shown on the top screenshot, SQUARE, which, when solved, shows the middle screenshot, and CIRCLE, which, when solved, shows the bottom screenshot. Pressing x cycles between these projections. Variable opt_texture is set to the current projection, which can be TX_None, TX_Orig, TX_Square, or TX_Circle. (In hw04.cc those are enumeration constants in hw04-shdr.cc those are integer constants.)

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

Code Generation and Debug Support

The compiler generates an optimized version of the code, hw04, and a debug-able version of the code, hw04-debug. The hw04-debug version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run hw04-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`, `opt_tryout2`, `opt_tryout3`, and `opt_tryoutf`, available in CPU and 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`, `Y`, and `Z` toggle the value of host Boolean variables `opt_tryout1`, `opt_tryout2`, and `opt_tryout3`. The user interface can also be used to modify host floating-point variable `opt_tryoutf` using the `Tab`, `+`, and `-` keys, see the previous section.

Resources

A good reference for C++ is `https://en.cppreference.com/w/`. Solutions to the shader programming problems may (will) require the use of library functions. See Chapter 8 of The OpenGL Shading Language Version 4.6 specification.

Problem 1: Modify the code in `render_hw04`, excerpted below, so that it renders a gold ring around the wheel (see the screenshots).

```
/// Put Homework 4,  Problem 1 solution below.
//

// Compute center of outer ring.
pCoor ctr(0,0,0);
const auto& balls_outer = hw04_stuff.balls_outer;
for ( Ball* b: balls_outer ) ctr += b->position;
const size_t n_balls = balls_outer.size();
ctr.w = n_balls;
ctr.homogenize();
pNorm wz [[maybe_unused]] =    // Wheel z
  cross(ctr,balls_outer[0]->position,balls_outer[n_balls/4]->position);
for ( size_t i = 0; i < n_balls; i++ )
  {
    Ball *b = balls_outer[i];
    pCoor pos = b->position;
    pNorm n(ctr,pos);
    pCoor p = pos + 2 * b->radius * n;
    bset_hw04_p1 << p << color_cyan << pNorm(0,1,0) << pTCoor(0,0);
  }
```

To help with this problem the current center of the wheel is in variable `ctr` and an approximate normal is in `wz`. Also, there is a loop that iterates over each outer-ring ball. In that loop body a vector, `n`, and coordinate, `p`, are computed that can be used to construct the ring. These are used to compute a point, but not necessarily the point(s) needed to solve the problem. Also, a placeholder color, normal and texture coordinate are inserted.

Pipeline `pipe_hw04_p1` is used to render the ring. It expects a vertex coordinate, color, normal, and a texture coordinate. Placeholder values are inserted for each of these attributes, which must be modified for the correct solution. Textures are optional for this problem. Texture coordinates might be chosen to help understand how texture coordinates work, to assist in solving Problem 3.

The Problem 1 solution code appears below. The code is entirely within `World::render_hw04`.

```
/// SOLUTION – Problem 1
```

```
//
for ( size_t i=0; i<=n_balls; i++ )
  {
    /// SOLUTION – Problem 1
    //
    // Visit the i=0 ball twice so that a triangle strip around the
    // wheel can be completed. To do that change loop end condition
    // above from "i<n_balls" to "i<=n_balls" and change the index
    // to balls_outer, below, from i to i%n_balls.

    Ball *b = balls_outer[i%n_balls];

    /// SOLUTION – Problem 1
    //
    //  Compute two coordinates (per iteration). Both points are on
    //  a line starting at the wheel center, ctr, and and passing
    //  through the ball center, pos. The first, p, is at a distance
    //  b->radius from the ball center. (This coordinate was
    //  provided in the unmodified assignment.) The second, pb, is
    //  at a distance 2 * b->radius from p.

    pCoor pos = b->position;
    pNorm n(ctr,pos);
    pCoor p = pos + 2 * b->radius * n;
    pCoor pb = p + 2 * b->radius * n;

    //  Insert those too coordinates into the buffer set.
    //
    bset_hw04_p1 << p << pb;

    //  Also insert colors.
    //
    bset_hw04_p1 << color_lsu_spirit_gold << color_lsu_spirit_gold;

    //  Problem 1 did not require that any particular texture coordinates
    //  be inserted, so it's hard to get this wrong. Here texture
    //  coordinates are inserted so that an entire copy of the texture
    //  appears between each pair of adjacent balls.
    //
    bset_hw04_p1 << pTCoor(0,i) << pTCoor(1,i);

    //  Problem 1 did not specifically require that the approximate
    //  normal, wz, had to be improved on.  If the wheel were rigid
    //  then wz would be a correct normal, but the wheel can flex,
    //  so its not. Here the approximate normal is used.
    //
    bset_hw04_p1 << wz << wz;

    // Commented out code from unmodified assignment.
    // bset_hw04_p1 << p << color_cyan << pNorm(0,1,0) << pTCoor(0,0);
  }
```

Problem 2: Pipeline `pipe_hw04_p2` has only two inputs, a vertex coordinate and a texture coordinate. A normal is needed to compute lighting. Modify the shaders in `hw04-shdr.cc` so that they compute a normal based on triangle coordinates. *Hint: This is easy too.*

The solution code, shown below, is entirely within shader routine `gs_main`, in file `hw04-shdr.cc`.

```
void gs_main() {
  // SOLUTION -- Problem 2
  //
  //  Compute normal by taking the cross product of vectors formed
  //  by the triangle vertices.
  //
  vec3 normal_o =
    cross( In[1].vertex_o.xyz - In[0].vertex_o.xyz,
           In[2].vertex_o.xyz - In[0].vertex_o.xyz );
  vec3 normal_e = mat3( ut.eye_from_object ) * normal_o;

  // Note: The code below is not part of the solution.

  for ( int i=0; i<3; i++ )
    {
      gl_Position = ut.clip_from_object * In[i].vertex_o;
      Out.normal_e = normal_e;
      Out.vertex_e = ut.eye_from_object * In[i].vertex_o;
      Out.tex_coor = In[i].tex_coor;
      EmitVertex();
    }
  EndPrimitive();
}
```

Problem 3: Texture coordinates for the spoke material is assigned in routine `World::ball_setup_4()`, search for Homework 4 in that routine. Coordinates are assigned to `Ball` members `Ball::tc.orig`, `Ball::tc.square`, and `Ball::tc.circle`.

```
  for ( int i=0; i<n_balls; i++ ) {
      const float theta = d_theta * i;
      pVect pos_n = cosf(theta) * ax + sinf(theta) * ay;
      pCoor pos_o = center_pos + ring_outer_radius * pos_n;
      pCoor pos_i = center_pos + ring_inner_radius * pos_n;

      // Construct new ball and add it to the ball list.
      Ball* ball_o = balls_outer.emplace_back( new Ball(pos_o,ballo_default) );
      Ball* ball_i = balls_inner.emplace_back( new Ball(pos_i,balli_default) );
      Link *link = new Link(ball_o,ball_i, {.spring_constant=opt_spring_constant/200});
      links.push_back(link);
      hw04_stuff.ribs.push_back(link);

      // Homework 4 -- Change these values for experimentation, if you like.
      ball_o->tc.orig = pTCoor(i,0);     ball_i->tc.orig = pTCoor(i,1);

      //  HW 4: Modify the assignments to the tc.square and tc.circle members.
```

```
ball_o->tc.square = pTCoor(i,0);   ball_i->tc.square = pTCoor(i,1);
ball_o->tc.circle = pTCoor(i,0);   ball_i->tc.circle = pTCoor(i,1);
```

In the unmodified assignment all three are assigned the same texture coordinate. Note that the code assigns texture coordinates for an outer-ring ball, `ball_o`, and an inner-ring ball, `ball_i`. (*a*) Assign `Ball:tc.square` so that the texture is mapped as a square. The blue ball is on the middle right-hand side and the yellow balls are on the upper-left quadrant. See the middle screenshot on *the first page, not the one below*.

To find the texture coordinates imagine that a circle, the wheel, were drawn over the texture image. The texture coordinates range from $\begin{bmatrix}0\\0\end{bmatrix}$ to $\begin{bmatrix}1\\1\end{bmatrix}$ with $\begin{bmatrix}0.5\\0.5\end{bmatrix}$ in the center of the texture image. So to find the texture coordinates for the inner and outer balls of the wheel we need to determine where on the circle we are. Consider the code computing the balls' positions, `pos_o` and `pos_i`:

```
for ( int i=0; i<n_balls; i++ ) {
    const float theta = d_theta * i;
    pVect pos_n = cosf(theta) * ax + sinf(theta) * ay;
    pCoor pos_o = center_pos + ring_outer_radius * pos_n;
    pCoor pos_i = center_pos + ring_inner_radius * pos_n;
```

The balls' positions are found using vector `pos_n` which is a unit vector pointing from the center of the wheel toward the intended ball positions. We can construct a similar vector to find the texture coordinates by replacing `ax` and `ay` with $\begin{bmatrix}1\\0\end{bmatrix}$ and $\begin{bmatrix}0\\1\end{bmatrix}$. Or more simply by just computing `vec2( cosf(theta), -sinf(theta) );`. To get the actual texture coordinate compute $\begin{bmatrix}0.5\\0.5\end{bmatrix} + r\begin{bmatrix}\cos(\theta)\\-\sin(\theta)\end{bmatrix}$, where $r = 1$ for the outer ball and $r = r_i/r_o$, where r_i is the inner radius, `ring_inner_radius` and r_o is the outer radius, `ring_outer_radius`.

The solution code appears below, taken from `World::ball_setup_4` in file `hw04.cc`:

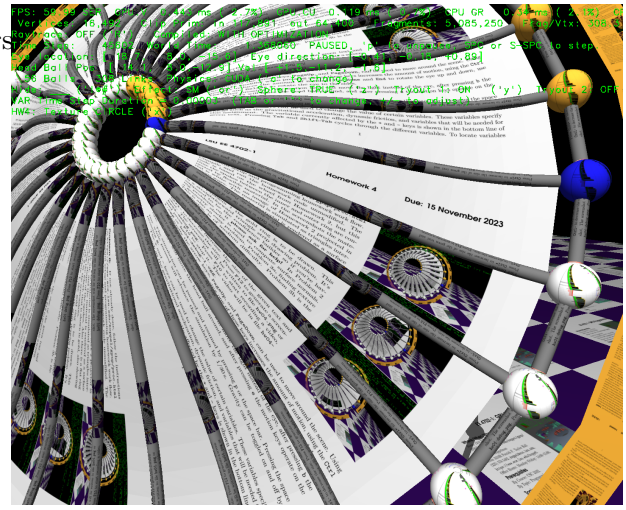
```
//  First, compute a vector of length 0.5 from the circle enter to the circle:
vec2 tex_sq_norm = 0.5 * vec2( cosf(theta), -sinf(theta) );

//  Prepare a coordinate of the circle center.
vec2 tex_ctr(0.5,0.5);

//  Add the vector to the circle center to get the outer-ring texture coordinate.
ball_o->tc.square = tex_ctr + tex_sq_norm;

//  Use a smaller radius for the inner ball.
float ring_io = ring_inner_radius / ring_outer_radius;
ball_i->tc.square = tex_ctr + ring_io * tex_sq_norm;
```

(*b*) Assign `Ball:tc.circle` and modify the shaders so that the texture is mapped wrapping around the ring as shown in the screenshot to the right and the bottom screenshot on the first page. In those screenshots about four copies of the image are shown.



The exact number of copies is not important, however the text away from the center must be readable (not stretched or compressed). Notice that the lines of text are parallel to the spokes (actually parallel to their centers, not the surfaces).

This is more difficult than part a because it requires more than just assigning the right texture coordinates. To solve this correctly one might need to provide additional uniform variables. Here is how one might add a new variable `factor_x`.

Step 1 – Declare in CPU Code
Add the variable to `struct Uni_HW04`:

```
struct Uni_HW04
{
  pColor color;
  int opt_texture;

  float factor_x; // Added a new variable
};
```

Step 2 – Assign in CPU Code
Assign the value, a convenient place is in `hw04_render()`.

```
  uni_hw04_needs_update = true; // Set this if x changes every frame.

  if ( uni_hw04_needs_update )
    {
      uni_hw04_needs_update = false;
      uni_hw04->color = 1.0 * color_white;
      uni_hw04->opt_texture = opt_texture;
      uni_hw04->factor_x = x; // Assign new member.
      uni_hw04.to_dev();
    }
```

Step 3 – Declare in Shader Code
Declare in the uniform shown below. The type and order must match the CPU code. Putting things in the wrong order or using the wrong type will result in incorrect execution but there won't be an error message.

```
layout ( binding = BIND_HW04 ) uniform HW_04
{
  vec4 color;
  int opt_texture;
  float factor_x;  // New member.
```

```
};
```

Step 4 – Use The Variable in Shader Code
Here the value is assigned to the red component of the lit color.

```
void fs_main()
{
  vec4 lit_color = generic_lighting(In.vertex_e, color, In.normal_e);
  lit_color.r = factor_x;
```

For 3b the texture should wrap around the ring. To achieve this effect the x component of the texture coordinate should be proportional to the distance of the ball from the wheel center. If we want the inner ring to be on the left-hand edge of the texture and the outer ring on the right-hand edge of the texture we would set the x component on the inner balls to 0 and the x component on the outer balls to 1.

To get the wrapping effect the y component should be proportional to the angle, θ. Call the proportionality factor ω. Then for the inner ball we would choose texture coordinate $\begin{bmatrix} 0 \\ \omega\theta \end{bmatrix}$ and for the outer ball $\begin{bmatrix} 1 \\ \omega\theta \end{bmatrix}$. We want to choose ω so that the text in the texture image does not appear squished together (condensed) or stretched out. The distance between the inner and outer edge is 1 in texture space (because of the way the x component of the texture coordinate was chosen) and $r_o - r_i$ in global space, where r_o is `ring_outer_radius` and r_i is `ring_inner_radius`. Assuming we want to maintain the texture aspect ratio a distance of 1 in texture space along a circle between the inner- and outer-ring must correspond to a distance of $r_o - r_i$ in global space. Let θ_1 be the angle at which the distance along that mid circle is the distance between the edges: $\theta_1 \frac{r_o+r_i}{2} = r_o - r_i$, solving $\theta_1 = 2\frac{r_o-r_i}{r_o+r_i}$. If we choose $\omega = 1/\theta_1$ the aspect ratio will match the original texture. In the solution code, including the fragments below, `delta_r` $= r_o - r_i$ and `r_mid` $= \frac{r_o+r_i}{2}$.

In an imperfect solution would insert the texture coordinates computed above into the pipeline:

```
const float delta_r = ring_outer_radius - ring_inner_radius;
const float r_mid = ( ring_outer_radius + ring_inner_radius ) / 2;
const float omega  = r_mid / delta_r;

ball_o->tc.circle = vec2( 1, theta * omega );
ball_i->tc.circle = vec2( 0, theta * omega );
```

The texture coordinates above are correct for the inner and outer balls, and these are the texture coordinates sent to the vertex shader. Recall that each vertex of a triangle has a texture coordinate, and that for a fragment within the triangle the texture coordinate is computed by interpolating the values at the three vertices (using perspective-correct barycentric coordinates). An invocation of the fragment shader, which is invoked for each fragment of a triangle, will get a texture coordinate interpolated from the three vertices of the triangle. This interpolation works correctly for typical mappings of textures to triangles, but not for the circular wrapping we are attempting here. As a result the text in the texture image will be slightly zig-zagy.

To see that this *will not* compute the texture coordinates we want, consider texture coordinates assigned to two adjacent outer-ring balls. The x component of both are 1, which is correct. Now consider a fragment halfway between the two balls. The interpolated x texture coordinate component will still be 1. But that's not correct because the balls are on a circle and the halfway point between the two balls is *inside* the circle and so the x component of the texture, based on the radius, must also be less than one.

The solution to this dilemma is actually quite simple. Choose texture coordinates using a method similar to the one used for the square, except now do so with a center at $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and with the radius of the outer ring set to 1 and the inner ring set to the inner/outer ratio, and also place ω and other useful constants in a uniform:

```
vec2 loc_n = vec2( cosf(theta), -sinf(theta) );
```

```
    float ring_io = ring_inner_radius / ring_outer_radius;
    ball_o->tc.circle = loc_n;
    ball_i->tc.circle = ring_io * loc_n;

    uni_hw04->r_scale_factor = 1 / ( 1 - ring_io );
    uni_hw04->omega = r_mid / delta_r;
```

In the fragment shader we will use these coordinates to compute the distance of the fragment from the wheel center (in fragment space) and the angle, θ, and from those get texture coordinates:

```
if ( opt_texture == TX_None )
  {
    out_frag_color = lit_color;
  }
else if ( opt_texture == TX_Circle )
  {
    float dist = length(In.tex_coor);
    float angle = atan(In.tex_coor.y,In.tex_coor.x);
    vec2 tc =  vec2( ( dist - 1 ) * r_scale_factor,
                     angle * omega );

    out_frag_color = lit_color * texture(tex_unit_0,tc);
  }
else
  {
    out_frag_color = lit_color * texture(tex_unit_0,In.tex_coor);
  }
```