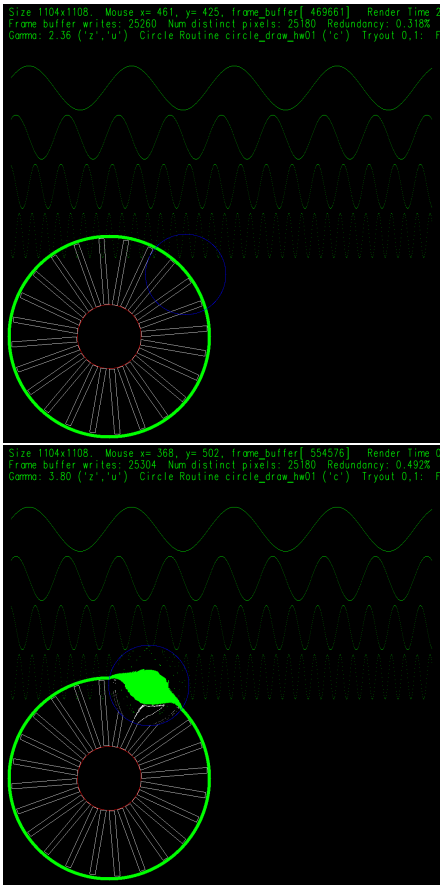
LSU EE 4702-1 **Homework 1** **Due: 9 September 2025**

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/2025/hw01.cc.html`.

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 some sine waves and some radially arranged white rectangles with an inner red circle and an outer green ring (which is just another name for a thick green circle). See the screenshot to the upper right. dots. In the lower image, taken from a correct solution. Though it can't be seen in the screenshots, the render time is much lower. Also, there is a blue circle inside of which the image appears distorted.



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.

General 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 `hw01.png` or `hw01-debug.png`.

The value of two Boolean debug-support variables, `tryout0` and `tryout1`, are shown in the green text, pressing `y` toggles `tryout0` (between `true` and `false`) and pressing `Y` toggles `tryout1`. The variables are available in routine `render_hw01` and `circle_draw_hw01`, use them to try things out.

Problem-Specific User Interface

The circle can be drawn by two routines `circle_draw_hw01` (where part of the solution is to be placed) and `circle_draw_parametric` (which is there for reference). Pressing `c` will toggle between two routines. The routine currently being used is shown in the last line of green text to the right of `Circle Routine`.

Pressing `z` will increase the distortion factor, γ, and pressing `u` will decrease it. Distortion, which should appear in the blue circle, won't be seen until Problem 2 is solved correctly.

Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw01`, and a debug-able version of the

code, `hw01-debug`. The `hw01-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw01-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. These are `tryout0` and `tryout1`. You can use these variables in your code (for example, `if ( 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` to toggle their values, which are shown in the green text at the label `Tryout 0,1:`.

Homework Code Overview

For this assignment the code in routines `hw01_render` and `circle_draw` will be modified. The code in `hw01_render` writes the frame buffer with sine waves and a circle of rectangles radiating from a point. The code for the sine waves and circle of rectangles is there for references, but it does not need to be modified for this assignment.

Resources

A good reference for C++ is `https://en.cppreference.com/w/`. See Homework 1 assigned in the past few semesters for similar problems. In particular, in 2023 Homework 1 Problem 2 part of the frame buffer around the mouse pointer was to be copied to another part of the frame buffer. Something like this copying is also done in Problem 2 in this assignment.

Collaboration Rules

Each student is expected to complete his or her own assignment. It is okay to work with other students and to ask questions in order to get ideas on how to solve the problems or how to overcome some obstacle (be it a question of C++ syntax, interpreting error messages, how a part of the problem might be solved, etc.) It is also acceptable to seek out programming and graphics resources. It is okay to make use of AI LLM tools such as ChatGPT and Copilot to generate sample code. (Do not assume LLM output is correct. Treat LLM output the same way one might treat legal advice given by a lawyer character in a movie: it may sound impressive, but it can range from sage advice to utter nonsense.)

After availing oneself to these resources **each student is expected to be able to complete the assignment alone**. Test questions will be based on homework questions and **the assumed time needed to complete the question will be for a student who had solved the homework assignment on which it was based**.

Student Expectations

To solve this assignment students are expected to avail themselves of references provided in class and on the Web site, such as for C++ programming and to seek out any additional help and resources that might be needed. (Of course this doesn't mean asking someone else to solve it for you.) Students are expected to experiment to learn how graphics work, and how to code C++ sequences. Experimentation might be done on past homework assignments. Students are also expected to learn what error messages mean by consulting documentation and by asking others (including Dr. Koppelman), and also to develop debugging skills. It is the students' responsibility to resolve frustrations and roadblocks quickly. (Just ask for help!)

This assignment cannot be solved by blindly pasting together code fragments found in class notes or past assignments. Solving the assignment is a multi-step learning processes that takes effort, but one that also provides the satisfaction of progress and of developing skills and understanding.

Problem 1: Routine `circle_draw_parametric` draws a circle (ring) into the frame buffer:

```
void circle_draw_parametric( pFrame_Buffer& fb, HW01_Data& hw01_data,
  int ctr_x, int ctr_y, int r_outer, int r_inner, uint32_t co)
{
  int win_width = fb.width_get();
  float delta_theta = asinf(1.0f/r_outer);
  for ( float theta = 0; theta < 2*numbers::pi; theta += delta_theta )
    {
      float cth = cosf(theta),       sth = sinf(theta);
      for ( int r = r_inner; r<=r_outer; r++ )
        {
          int x = ctr_x + r * cth,   y = ctr_y + r * sth;
          fb[ y * win_width + x ] = co;
        }}}
```

The center of the circle is at pixel coordinate `ctr_x`, `ctr_y`, the inner radius is `r_inner` and the outer radius is `r_outer`. Pixels of the frame buffer from the inner and outer radius are set to color `co`, or at least they should, but some pixels are skipped. The routine is based on a parametric description of a circle: $P(\theta) = C + \begin{bmatrix} r\cos\theta \\ r\sin\theta \end{bmatrix}$.

The routine above is not the best way of drawing a circle in pixel space (which is what we are using) because it calls trig functions (sine and cosine) many times (and that's computationally costly), because it doesn't write every pixel, and because it writes some pixels multiple times. (If `delta_theta` is made small enough it would write every pixel at least once, but it would be writing many more pixels multiple times.)

The fundamental problem with the routine above is that we are iterating over θ and r, a better approach is to iterate over pixel coordinates. In the unmodified assignment the code in `circle_draw_hw01` does iterate over pixel coordinates:

```
void circle_draw_hw01 ( pFrame_Buffer& fb, HW01_Data& hw01_data,
  int ctr_x, int ctr_y, int r_outer, int r_inner, uint32_t co)
{
  const int win_width [[maybe_unused]] = fb.width_get();
  const int win_height [[maybe_unused]] = fb.height_get();
  /// WARNING: Ridiculously inefficient.
  for ( int x = 0; x < win_width; x++ )
    for ( int y = 0; y < win_height; y++ )
      {
        int dx = x - ctr_x,   dy = y - ctr_y;
        int dsq = dx * dx + dy * dy;
        if ( dsq > r_outer * r_outer ) continue;
        if ( dsq < r_inner * r_inner ) continue;
        fb[ y * win_width + x ] = co;
      }
```

The code iterates over every pixel *in the frame buffer*, and checks whether that pixel is on the circle (between the inner and outer radii). On the plus side this code won't skip any pixels, it won't write the same pixel twice, it doesn't call any trigonometry functions, and it even avoids computing a square root. But all of those positives are outweighed by the fact that it checks every single pixel **in the frame buffer**!

Modify the code so that it iterates over fewer pixels. Consider solutions that keep the two-level loop nest, but in which the loop bounds are chosen so that the body is executed fewer times.

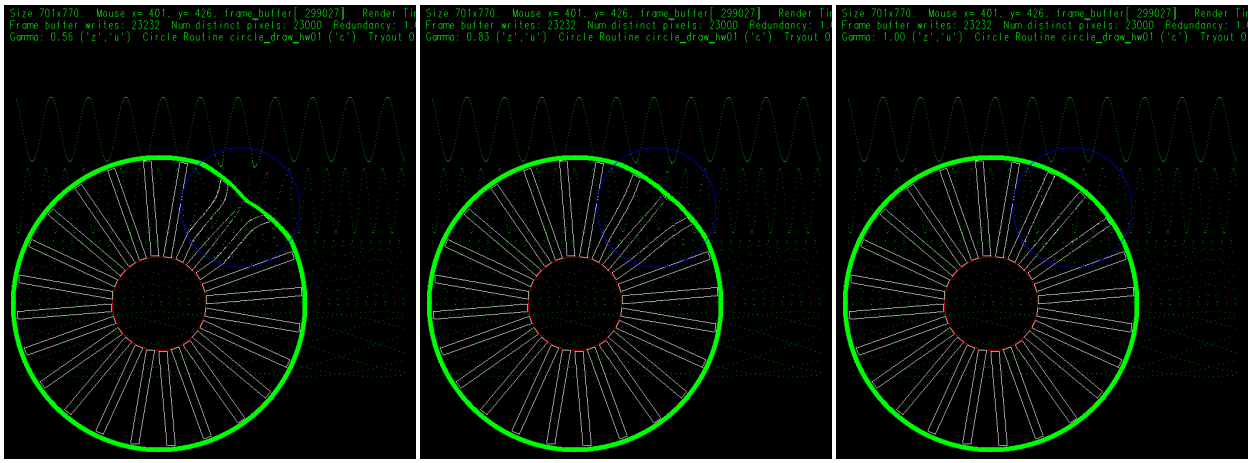
Consider the following ideas:

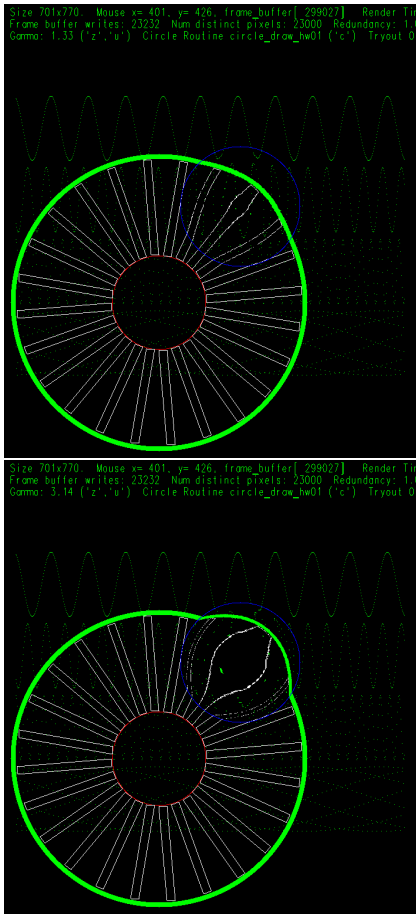
Idea 1: Modify the loop bounds so that `x` and `y` iterate over a *bounding box* that contains the circle. (A bounding box of a circle is the smallest possible box containing the circle. Any smaller and a part of the circle would be outside the box.)

Idea 2: Compute the limits on the `y` loop as a function of `x`. (Note that Idea 2 replaces or modifies the Idea 1 code.)

Idea 3: Have `x` and `y` iterate over only one quadrant of the circle, but have the loop body write four pixels (one write for the quadrant corresponding to `x` and `y`, and the other three writes for the other three quadrants.)

There's another problem on the next page.





Problem 2: A blue circle is drawn near the end of routine `render_hw01` in a call to `circle_draw`. Modify the code after this point so that the portion of the frame buffer inside the blue circle (but not the blue circle itself) is distorted as follows:

Let C denote the center of the circle (in the code the center is in variable `ctr`) and R its radius (`mag_r` in the code). Let $P = \begin{bmatrix} P_x \\ P_y \end{bmatrix}$ denote the coordinates of a pixel inside the circle.

Let $v = C - P = \begin{bmatrix} C_x - P_x \\ C_y - P_y \end{bmatrix}$ denote the vector from C to P. In vector notation $P = C + v$. Let d denote the length of v, which in this case is the distance from C to P. (If you don't know how to compute the distance, look it up.)

For this problem define *distorted distance* $d' = R\left(\frac{d}{R}\right)^{\gamma}$, where γ is a distortion factor. Note that for $\gamma = 1$ we just have $d' = d$. Given a distorted distance we can define an alternative pixel coordinate $P' = C + \frac{d'}{d}v$.

Modify the code in `hw01_render` so that every pixel P in the blue circle is written with the frame buffer contents at P' as defined above. In the code variable `ctr` is the circle center (see how it's used in the call to `circle_draw`). Variable `mag_r` is R, and `gamma` is γ. Variable `P` is P; your code can just use `x` and `y`. In the code variable `v` is v; your code can just use `dx` and `dy`. The code computes d^2, but not d. Using the user interface, the value of `gamma` can be modified by pressing `z` and `u`, and its value is shown on the third line of green text.

Some things to watch out for: most of the variables are declared `int`. To work correctly d' must be computed using floating point operands. Let `xp` and `yp` denote the coordinates of P'. If these are used to compute a frame buffer index, `fb[ yp * win_width + xp]`, make sure that `yp` is an integer, or cast it to an integer otherwise `yp * win_width` won't give the correct result.

Finally, and importantly remember that the pixel being read (at coordinate P') must be from

the frame buffer *before* it was modified by the code that fills the blue circle. So code like `fb[ y * win_width + x] = fb[ int(yp) * win_width + xp];` won't work correctly because the pixel being read, `fb[ int(yp) * win_width + xp];`, might have already been written. So, the area inside the blue circle first needs to be copied to a buffer area and that buffer area needs to be used when writing inside the blue circle. Before writing any pixels in the blue circle first copy them to array `fb_dup`. Array `fb_dup` is of size $(2(R+1))^2$, and so it has enough space for the blue pixels. It's okay of a bounding box around the blue circle is copied into `fb_dup`.

Once the blue circle are is copied to `fb_dup` one can get the distortion effect by writing `fb[ int(yp) * win_width + xp] = fb_dup[ b_y * b_width + b_x]`, where `b_width` is the width of the area copied **(which can't be larger than** $2(R+1)$**)** and `b_y` and `b_x` are relative to a corner of the copied area. See 2023 Homework 1 Problem 2 and its solution for more on how to use such a buffer area (called `fb_area_dup`) in that problem.