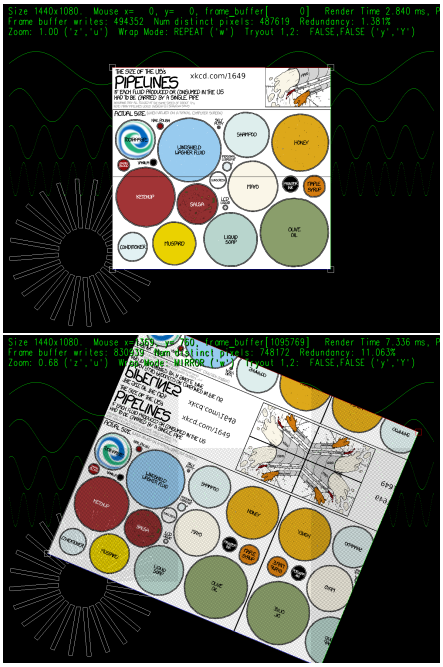
LSU EE 4702-1 **Homework 1** **Due: 14 September 2024**

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/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, radially arranged white rectangles, and an image of xkcd cartoon number 1649 (cropped). See the screenshot to the upper right. When the assignment is complete the image can be rotated and resized by dragging its corners, the image can be zoomed by pressing `z` and `u`, and the *wrap mode* can be changed by pressing `w` so that when more than one image fits in the image box it is either duplicated, mirrored, or just one copy is shown. The screenshot to the lower right shows a rotated, unzoomed, mirror-wrapped version.

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, `tryout1` and `tryout2`, are shown in the green text, pressing `y` toggles `tryout1` (between `true` and `false`) and pressing `Y` toggles `tryout2`. The variables are available in routine `render_hw01`.

Problem-Specific User Interface

Pressing `z` will increase the zoom level for the image, pressing `u` will decrease the zoom level. The zooming won't be seen until Problem 2 is solved correctly.

At each corner of the image box is a small square. To change the size of the image box drag the upper-right or lower-left corner. Both the height and width can be freely changed, down to a minimum size. In the unmodified assignment the image will be scaled to fit in the image box, as long as the image box isn't rotated.

To rotate and scale the image box drag the upper-left or lower-right corner. This will rotate the image and change its size, but its aspect ratio won't change. That is, if it were square before the drag is will be a square after the drag, probably rotated and a different size. In the unmodified code the image will not be rotated.

When the zoom level is less than 1 it should be possible to fit multiple images in the image box. The *wrap mode* indicates what should be displayed (of course, when the assignment is correctly solved). The wrap mode is shown in the green text. Pressing `w` will cycle through wrap modes *repeat*, *mirror*, and *transparent*. See Problem X for a description of what these should do.

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.

Integer Coordinate and Vector Types

This assignment uses both integer and floating point types for coordinates and vectors. (Previous frame buffer assignments just used integer types.) The integer types are `iCoor` for 2D coordinates and `iVect` for 2D vectors. The assignment also uses type `pCoor` for floating-point 3D coordinates (actually 4D, we'll learn in a few weeks) and `pVect` for floating-point 3D vectors.

The `iCoor` and `iVect` classes each have two members, `x` and `y`:

```
class iCoor {
public:
  iCoor( int x, int y ):x(x),y(y){}
  int x, y;
};
class iVect {
public:
  iVect( int x, int y ):x(x),y(y){}
  int x, y;
};
```

The addition, subtraction, and division operators are overloaded with these types. Here are some examples of how to use them:

```
void demo() {
  iCoor c0(3,4);  // Set to x=3, y=4.
  iCoor c1;
  c1.x = 5; c1.y = 7;  // Set to x=5, y=7;

  iCoor c3 = c1;

  // Component-wise subtraction.
  // Equivalent to v1.x=c0.x-c1.x;  v1.y=c0.y-c1.y;
  iVect v1 = c0 - c1;

  // Component-wise addition.
  // Equivalent to c4.x=c1.x+v1.x;  c4.y=c1.y+v1.y;
  iCoor c4 = c1 + v1;

  // Not allowed: Can't add two coordinates.
  iCoor c5 = c4 + c1;
}
```

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 `tryout1` and `tryout2`. 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 1,2:`.

Homework Code Overview

For this assignment the code in routine `hw01_render` will be modified. That code writes the frame buffer with sine waves, a circle of rectangles radiating from a point, and fills the image box (described earlier) with an image read from a file. 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.

The coordinates of the image box are placed in variables `p0`, `p1`, `p2`, and `p3`. The width and height of the image box (assuming it is not rotated) is computed by looking at the `x` and `y` members of the `pCoor` class:

```
// Image Box Vertices (Corners)
//
//   p3 --- p2
//   |       |
//   p0 --- p1
//
const pCoor p0 [[maybe_unused]] = img_box[0];
const pCoor p1 [[maybe_unused]] = img_box[1];
const pCoor p2 [[maybe_unused]] = img_box[2];
const pCoor p3 [[maybe_unused]] = img_box[3];

float box_wd = p1.x - p0.x;
float box_ht = p3.y - p0.y;
```

The units of `box_wd` and `box_ht` are pixels. The width and height of the image are placed in variables `img_wd` and `img_ht` then a scale factor is comptued:

```
const int img_wd = hw01_data.img_px_width;
const int img_ht = hw01_data.img_px_height;

float img_scale_x = img_wd / box_wd;
float img_scale_y = img_ht / box_ht;
```

Variable `img_scale_x` indicates how many image pixels are to fit into one frame buffer pixel, horizontally, `img_scale_y` is set similarly. These variables are set so that the entire image fits in the image box.

A loop nest iterates over the height and width of the image box, and at each iteration is writes one frame buffer pixel (at coordinates `fb_x` and `fb_y` with one image pixel (at coordinates `img_x` and `img_y`. Pay attention to how these sets of coordinates are computed in the code below (in which some parts are omitted):

```
for ( int y=0; y<box_ht; y++ )
  for ( int x=0; x<box_wd; x++ )
    {
      int fb_x = p0.x + x;
      int fb_y = p0.y + y;

      int img_x = x * img_scale_x;
      int img_y = y * img_scale_y;
```

```
      uint32_t img_pixel = img_pixels[ img_idx ];

      fb[ fb_y * win_width + fb_x ] = img_pixel;
    }
```

The code above correctly displays the image so long as the image box is not changed and zoom is left at 1. In the problems that follow the code is to be modified so that the image is correctly displayed when the image box is rotated, scaled, and zoom and wrapped options are changed.

Resources

A good reference for C++ is `https://en.cppreference.com/w/`. See Homework 1 assigned in the past few semesters for similar problems.

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**.

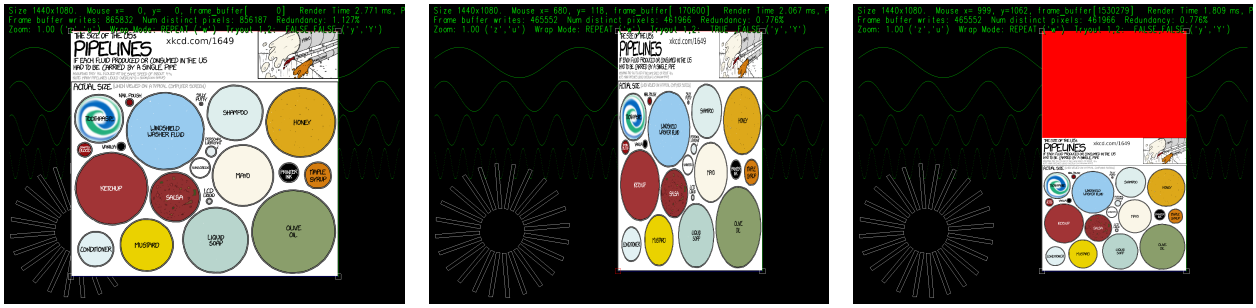
Student Expectations

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: The image box can be resized by dragging the lower-left or upper-right corners. (Dragging the other corners will rotate the image, which we don't want to do in this problem.) When the image box is resized this way the width and height can be changed independently, meaning if the aspect ratio were $w/h = 0.5$, where w denotes the width and h denotes the height) before the drag it, might be $w/h = 0.9$ after the drag. Since we are displaying images that's not a good thing, because circles should not look like ovals. The screenshot to the lower left shows the image before resizing, the one in the middle shows the image after resizing with the aspect

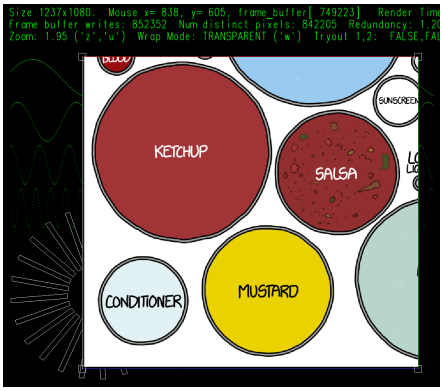
ratio wrong (circles are now ovals), the one on the right shows the image displayed with the correct aspect ratio. The red area will be fixed in Problem 3.



Modify the code in `render_hw01` so that the aspect ratio is preserved. When zoom is 1 (the default) one image should be across with width of the box, regardless of how the image box is resized. For this problem don't worry about unfilled parts of the image box (shown in red in the screenshot).

Hint: This is easy, just one line needs to be changed.

Problem 2: Let z denote the value of variable `zoom`. Modify the code in `render_hw01` so that $\frac{1}{z}$ images are shown horizontally inside the image box. (The size of the image box itself should not be changed, and the image should not be written outside of the image box.) For example, when $z = 2$ the image should appear twice as large. The value of variable `zoom` can be changed by pressing `z`, `Z`, `u`, and `U` (for zoom and unzoom). The upper-case variants change the zoom level by larger amounts. The screenshot to the right shows the image at $z = 1.95$.

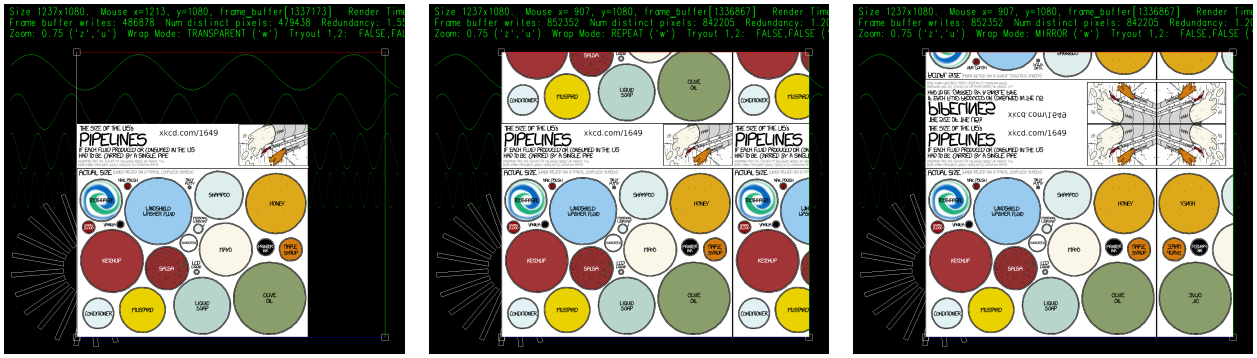


The solution to this problem is also easy. If you're writing a lot of code you're on the wrong track.

There are more problems on the next page.

Problem 3: When the zoom level is less than one, more than one image will fit in the image box. (That can happen too if the zoom level is one or larger but the image box is narrow.) In all cases the lower-left of the image box should hold a copy of the image (or as much as will fit). The value of variable `wrap_mode` indicates what should be displayed in the remaining space.

When `wrap_mode = WM_Transparent` nothing should be written in the remaining parts of the image box. One should be able to see whatever was there before, such as the sine waves. See the screenshot to the lower left. When `wrap_mode = WM_Repeat` the image should be repeated, like a pattern on wallpaper. See the screenshot in the middle. When `wrap_mode = WM_Mirror` the image should be repeated, but reflected as though the copy to the right of the original is its mirror image. The copy above should also be a mirror image. In fact, the mirroring should be done across each line separating copies of the image. See the screenshot to the lower right. (Such mirroring is often done with images intended to give an object a realistic texture, such as a wood grain or of concrete. Mirroring makes the boundaries less visible.)



Modify `render_hw01` so that the image box contents is set for the wrap modes. For your solving convenience in the loop writing the image to the frame buffer there is a `switch` statement with a case for each wrap mode.

The transparent mode is easy, just don't write anything if the image pixel coordinates are out of range. The modulus operator, `%`, should come in handy for the other two wrap modes.

Problem 4: The image box can be rotated and resized by dragging its upper-left or lower-right corner. (Unlike when dragging the other two corners, the aspect ratio will be preserved.) Modify the code in `render_hw01` so that the image is placed in the image box, even after rotation, and is zoomed and wrapped based on `zoom` and `wrap_mode`. See the second screenshot on the first page of this assignment.

In a simple solution, and the one that is sufficient for full credit, the `x` and `y` loop limits and increment are left unchanged. (That is, the `for` statements aren't changed.) As before, set `box_wd` to the distance between points `p0` and `p1` and `box_ht` to the distance between points `p0` and `p3`, but note that when the image box is rotated `box_wd=p1.x-p0.x` does not compute the correct distance between `p0` and `p1`. The only other thing that needs to be done is to change how `fb_x` and `fb_y` are computed. This simple solution will skip over some pixels, the pattern of skipped pixels varies by rotation angle. That can be seen on the screenshot on the first page. Such a solution will get full credit. *The skipped pixels can be avoided by replacing the* `x,y` *loop nest with a loop nests that iterate over frame buffer pixels of the rotated image box. That would be too tedious for a problem 4 on a first assignment.*