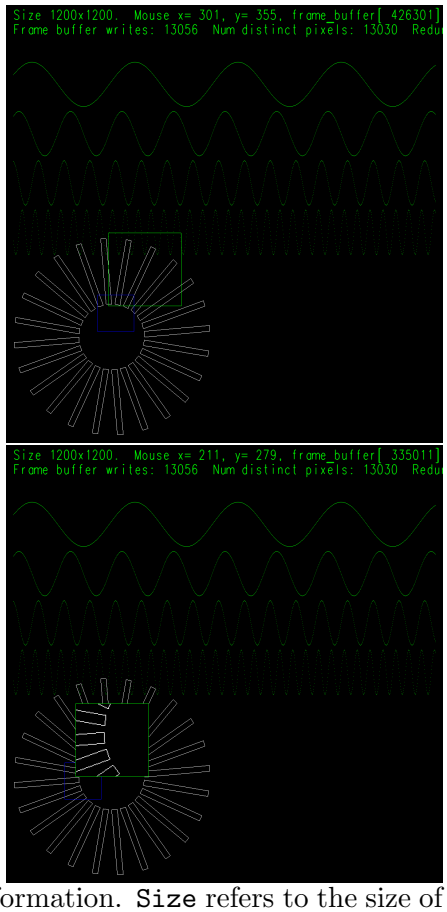
**LSU EE 4702-1** **Homework 1** **Due: 1 September 2023**

*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/2023/hw01.cc.html`*. The solution code is at* `https://www.ece.lsu.edu/koppel/gpup/2023/hw01-sol.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 radially arranged white rectangles. See the screenshot to the upper right. Two boxes should follow the mouse pointer around the window. The mouse pointer should be in the center of the blue box. A green box is nearby. The relative position and size of the green box can be changed using the keyboard, as described further below. After this assignment is correctly solved the green box should show a zoomed version of what is in the blue box. That appears in the screenshot to the lower right.




## 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`.

To change the size of the green box use the arrow keys. The left and right keys change the width, the up and down keys change the height. Pressing shift and an arrow key will move the green box relative to the blue box. It is possible for the green box to completely cover the blue box.

## 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.`.

## Integer Coordinate and Vector Types

This assignment uses integer 2D coordinate type `iCoor` and integer 2D vector type `iVect`. These types will just be used for this assignment, but in other code we will be using similar types, `pCoor` and `pVect` which are for 3D and in which the components are floating-point types.

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;
};
```

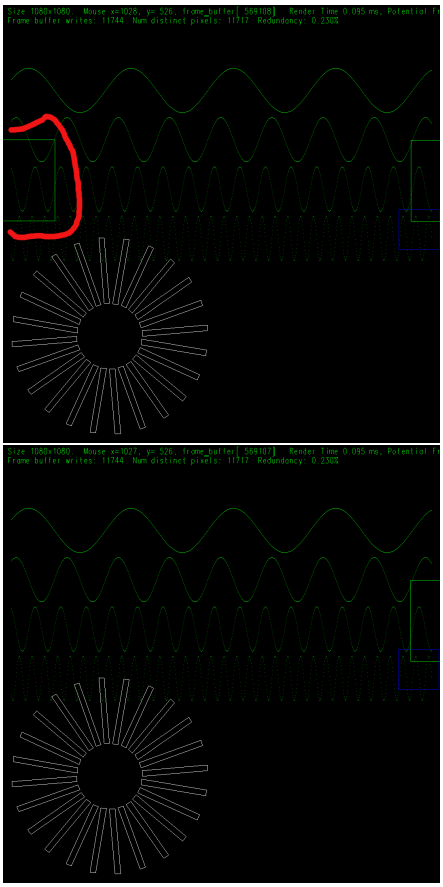
The addition, subtraction, and division operators are overloaded with these types. Here are some examples of how to use them:

```
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.

**Problem 1:** In the unmodified assignment, when the mouse is near the right window edge the green box will wrap around to the left side of the window. See the upper screen-shot to the right, where the wrapped part is circled (almost) in red. The lower screen shot is from code in which this problem is correctly solved but Problem 2 is not started. The green box does not wrap to the left side.



The green box is drawn by calling `aa_rectangle_draw` (`aa` is for axis-aligned). The rectangle is specified by a lower-left coordinate, `p00`, a width, `wd`, and a height, `ht`. The routine draws the rectangle by calling a line drawing routine four times. Modify `aa_rectangle_draw` so that the rectangle it draws does not go beyond a window edge (and so the green box won't wrap).

The routine `aa_rectangle_draw` calls `line_draw` four times. What makes this problem easy is that each line is parallel to an axis. The $x$-axis values range from `p00.x` to `p00.x+wd`. These need to be checked and possibly adjusted to stay within the range 0 to `win_width-1`. Also note that if `p00.x` is less than zero, the left vertical line should not be drawn. Similar reasoning is used to skip drawing the other three lines of the rectangle.

*There is another problem on the next page.*

**Problem 2:** Code near the end of `render_hw01` draws the blue and green rectangles described in the homework introduction. Add code to the routine so that the contents of the frame buffer inside the blue rectangle is copied to the part of the frame buffer inside the green rectangle. A similar problem was asked in 2022 Homework 1, feel free to look at its solution to help get started.

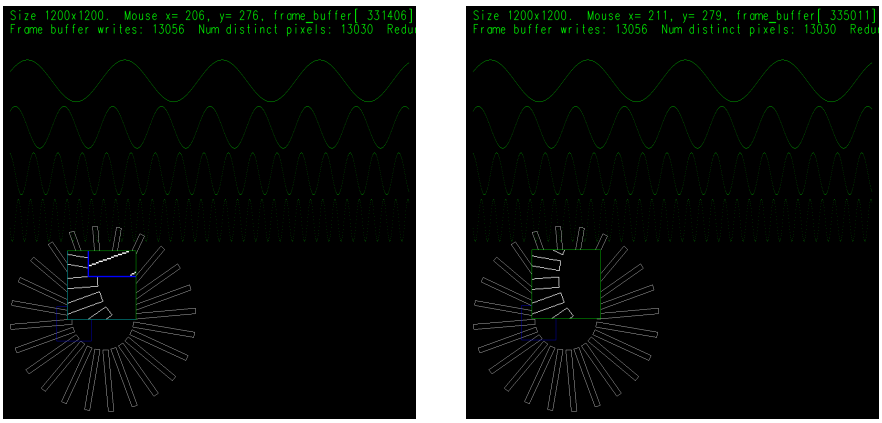
The coordinates of the lower-left of the blue rectangle, called the `mag_from` area in the code, is in variable `mag_from_ll`, a variable of type `iCoor`. (See description in the problem introduction.) The width and height are in `mag_from_diag`.

The coordinates of the lower-left of the green rectangle, called the `mag_to` area in the code, is in variable `mag_from_ll`, and the width and height are in `mag_to_diag`.

In order to solve this problem correctly when the blue and green box overlap, the contents of the blue box will need to be copied to intermediate storage before it is copied to the green box. Use array (actually a vector) `hw01_data.fb_area_dup` for this purpose. The size is `mag_from_diag.x * mag_from_diag.y` elements.

To help with your understanding of how to read and write the frame buffer an the intermediate storage there is code that draws diagonal lines into the green and blue rectangles, and into the intermediate storage. That code is guarded by `show_starter_code`. To actually toggle the visibility of those diagonal lines press `y`.

The screenshot to the lower right was taken from code with the correct solution. Notice that the green box only shows zoomed material and that one cannot see the blue box inside the green box. The shot to the left shows an incorrect solution, one in which intermediate storage was not used or not used properly.




Make sure that your solution works correctly when the green box is moved and re-sized. Use the arrow keys to re-size and shift-arrow to move the green box.