CS450/550 — Program 2 — Due January 30, 1996

In this assignment, you will write a program that "bounces" a circular ball around the screen. The assignment will give you some practice with a problem that is closely related to the problem of line clipping.

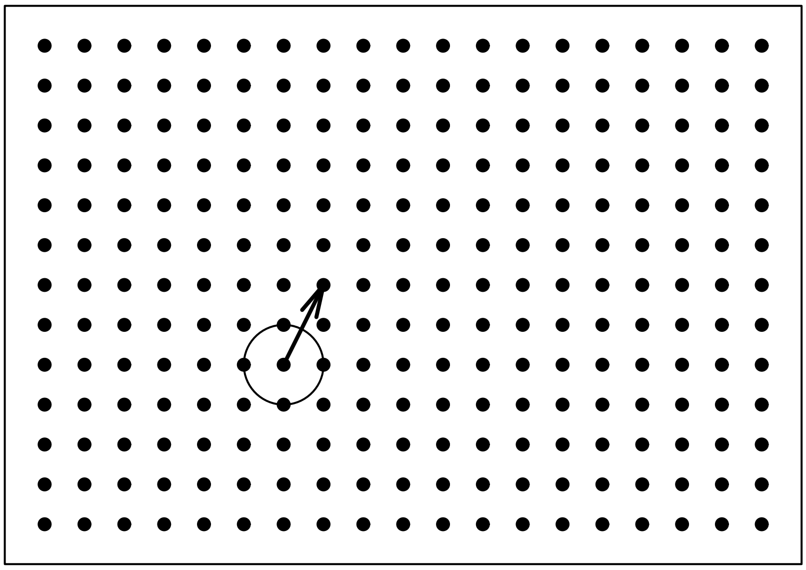

Figure 1: A ball moving in direction $(1, 2)$

Consider Figure 1. A circle of radius 1 is shown, along with a "velocity vector" with components $(1, 2)$. Your main program will create the circle and the velocity vector and then enter a main loop. In the loop, you will put the program to sleep for 0.5 seconds, then erase the circle, move it by an amount specified by the velocity vector (e.g., +1 in the X direction and +2 in the Y direction, in this case), and then redraw the circle. This will make the circle appear to move slowly across the screen. (The velocity vector should not be drawn.)

When the circle reaches the edge of the window, you will need to make it "bounce", as shown in Figure 2. Notice that the center of the circle cannot actually reach the edge of the window. It must stop at a distance r away from the edge, where r is the radius. Notice also that the velocity vector must change when the circle "bounces". In this case, it becomes $(1, -2)$. In general, if we bounce off a vertical wall, the sign of the x coordinate will change. If we bounce off a horizontal wall, the sign of the y coordinate will change.

Things can get a bit more complicated as shown in Figure 3. If the velocity vector would take the circle beyond the edge of the wall, it must bounce at the point where it would exactly hit the wall, and then continue for the appropriate amount of distance. In this case, the velocity vector wants to move 4 units in the y direction. After 2 units, the circle hits the wall and bounces. It continues for 2 more units. This essentially involves clipping the velocity vector against the top of the window, moving the circle by the clipped amount, switching the direction of the velocity vector, computing how much of the vector is "remaining", and moving by that amount.

Finally, things can get extremely complicated in corners as shown in Figure 4. The velocity vector $(-5, 5)$ can potentially be clipped against both the top and the left side. The left side turns out to be the shorter clip, so it is the one we use. The circle moves diagonally one square, and the new remaining velocity vector is $(4, 4)$. It can still be clipped against the top to become $(2, 2)$. The remaining vector is $(2, -2)$.

After bouncing, we need to resume using the full length velocity vector, although it will now be $(5, -5)$.

Notice that we don't actually clip against the edges of the window. Instead, we must take into account the radius of the circle too.

One final complication: I would like your bouncing circle to leave a "trail" that shows where its center point has been (see Figure 5.)

To implement this, I suggest that you define a new subclass of `circle_t` called `ball_t`. A `ball_t` should have appropriate constructors, and it should have a data field for the velocity vector.

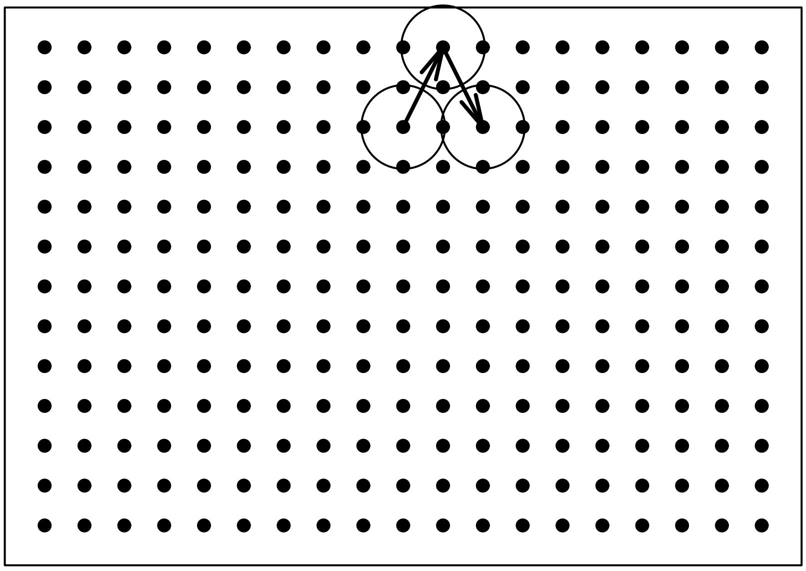

Figure 2: A simple "bounce"

I suggest that you define a member function called `step`, which makes the ball take one step (which may include a complex bounce) and lays down the trail.

You will find the following functions useful:

- **prim_t::erase(scene_t * s)** This will erase a primitive object (including polygons and circles). It works by drawing the object using the background color.
- `win_view_t::givleft()` This returns the left edge of the window in world coordinates. Analogous functions are defined: `givright()`, `givtop()`, and `givbottom()`.
- `prim_t::translate(point_t & p)` This will translate a primitive by the amount specified by point `p`.
- `micro_sleep(unsigned int usec)` This will make your process pause for `usec` number of microseconds. It is defined in `xglib.h`.

Your program should initialize the circle with its center at (200,200). It should prompt the user for the radius and initial velocity vector. Each time through the loop, your program should check whether the user has pressed the mouse and exit if this is true. There is a function `mouse_t::check_button_down(point_t &)` that will check whether the mouse button has been clicked. If it has, it will return an integer indicating which button was clicked, and it will set the X and Y coordinates of the `point_t` argument to be the world coordinates of the mouse position. If no button has been clicked, it will return 0. It does not "block" your process.

Hint: The quality of the display may be improved by calling `flush()` after each time you draw the circle. This forces all X events to be processed by the X server, so the circle will be completely drawn.

For Students in CS550:

Extend the `scene_t` class to support a list of bouncing balls. To do this, you should define a `copy` function for your `ball_t` class. Look at the `polygon_t` class to see how this is done. Add a member function to `scene_t` so that it will make each of its objects take a step.

Add code to your main loop to check whether the user has pressed the mouse button inside one of the balls. If so, erase that ball from the screen and delete it from the `scene_t`. (I call this "popping" the balls.) You will probably want to write a `scene_t::operator -= (prim_t &)` function to handle modifying the scene.

What to turn in:

For CS450, you should be able to email a `shar` file containing your source file and a `README` file that tells how to run it. For CS550, you will need to email a uuencoded tar file that should include a `README` file also.

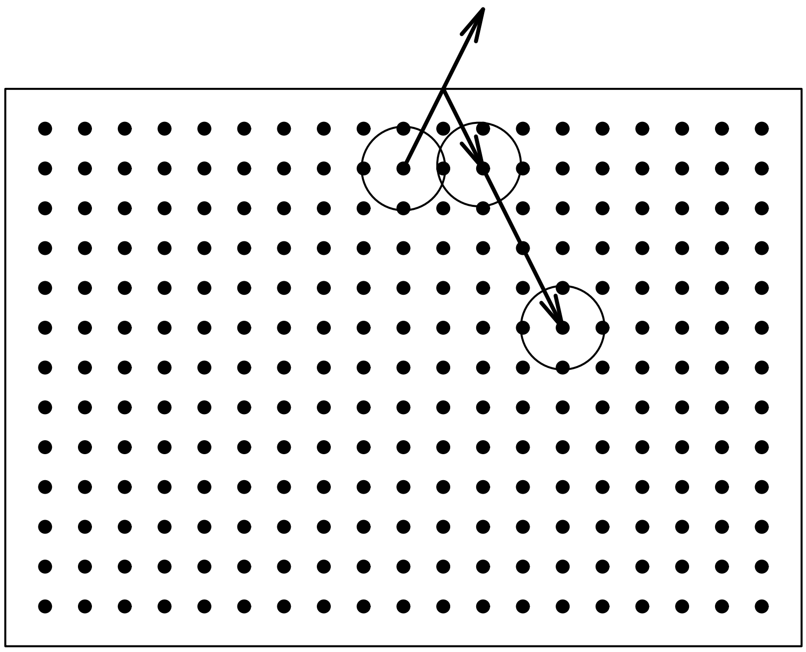

Figure 3: A more complex bounce.

The files necessary for this assignment can be obtained from the web page or by copying from `~tgd/public_html/p2.tar`
.

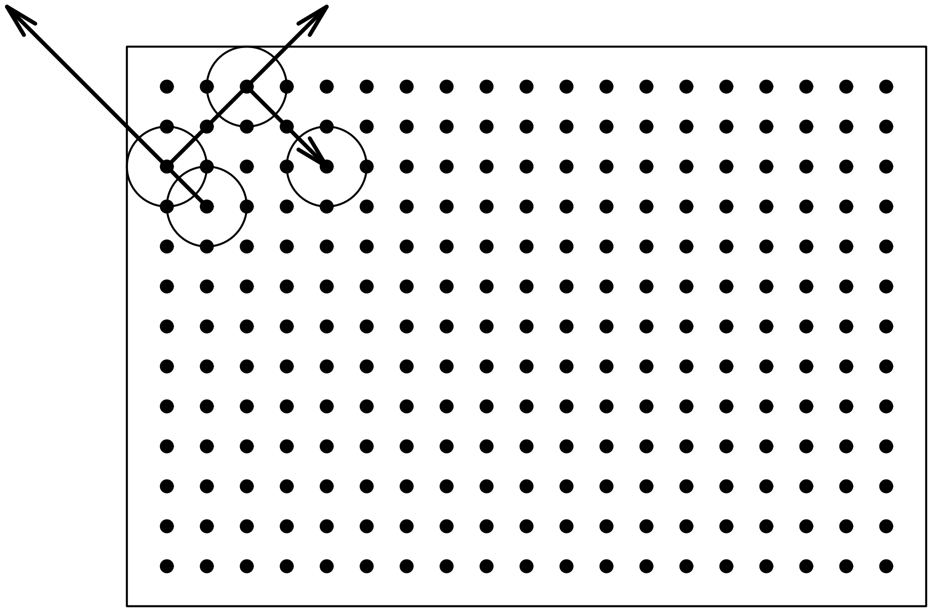

Figure 4: A corner bounce.

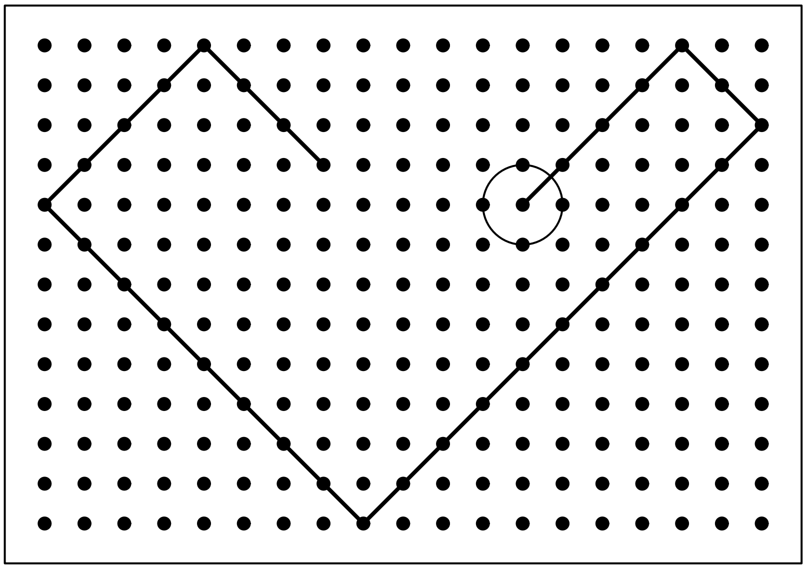

Figure 5: Leaving a trail.