EEN118 LAB FOUR

In this lab you will be performing a simulation of a physical system, shooting a projectile from a cannon and working out where it will land. Although this is not a very complicated physical system, it does introduce the fundamental principles of computer simulation, which is a very important field. As the last step, you will turn the simulation into a little interactive game about blowing things up.

1. *The Calculation.*

If a projectile is fired or thrown upwards at a known velocity, its entire trajectory can be accurately computed. You probably remember the physics formula

$$h = v\,t - \tfrac{1}{2}\,g\,t^2$$

(h is the projectile's height at any given time, v is the initial upwards velocity, t is the time elapsed since it was fired or thrown, and g is the "acceleration due to gravity", an approximate constant anywhere on the surface of the earth). If you measure h in feet, t in seconds, and v in feet per second, g is about 32.174.

Obviously this is going to be a very important formula for this program. One of the most important aspects of good programming is to make separate functions for all the important things.

So make a good start: write a clear and simple function that takes values for v and t as parameters, and calculates and returns the corresponding value of h. Think about what *types* the parameters and other constants should have for an accurate calculation.

Test your function thoroughly. It may be difficult to think of good values for testing. If you remember calculus you can work things out exactly, but velocities between 100 and 200 feet per second should keep the cannon ball in the air for a few seconds.

The exact formula for how long the ball stays in the air, when there is no atmosphere to slow it down is $(2\,v\,/\,g)$ seconds.

2. *Tabulation.*

It would be a lot easier to do the testing (and the results would be a lot more useful) if your program could automatically print the height at a number of different times. For instance, if you have a total flight time of 12 seconds, you might want to see what the height is at each of 0, 1, 2, 3, ..., 11, and 12 seconds.

```
after 0 seconds, height is 0 feet
after 1 seconds, height is 184 feet
after 2 seconds, height is 336 feet
after 3 seconds, height is 456 feet
```
etc.

Think of the various examples you have seen of programs controlling repetition, and you will probably find this part easy.

Don't modify the function you wrote for part 1 (unless there is something wrong with it). You have tested it and ensured that it is perfect. If you change it, all that testing was for nothing. Instead, write another function that makes use of the first one.

3. *Interaction.*

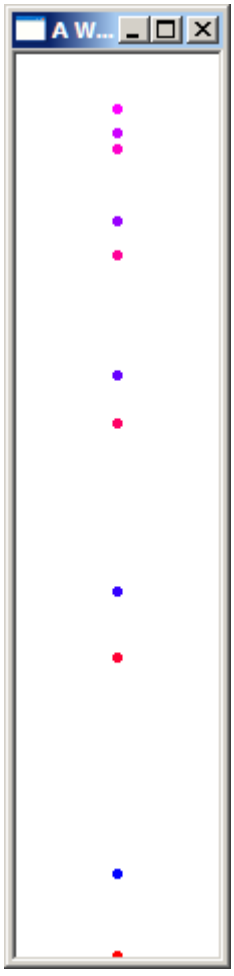
There are functions in the library called `read_int`(), `read_double`(), `read_float`(), `read_string`(), etc. When called, they wait until the user has typed something of the appropriate type, then return that value as their result. A program could rudely ask someone their age like this:

```
print("How old are you? ");
const int age = read_int();
print("That is old. in ten years you will be ");
print(age+10);
```

(The system only notices input typed when the black "dos shell" window is active.)

Make your program ask the user to enter the initial velocity for the cannon ball. It should then calculate the total flight time and, using your solution to part 2, print the height above ground at a reasonable number of times throughout the flight.

4. *Visual Representation.*



Now make your program plot those altitudes as vertical dots in a reasonably sized window, perhaps one dot for each second of flight. *If* you make the colour of the dots gradually change as time progresses, the results are much easier to interpret.

If you use reasonable velocities, you will be able to have a scale of one pixel per foot, which is quite convenient.

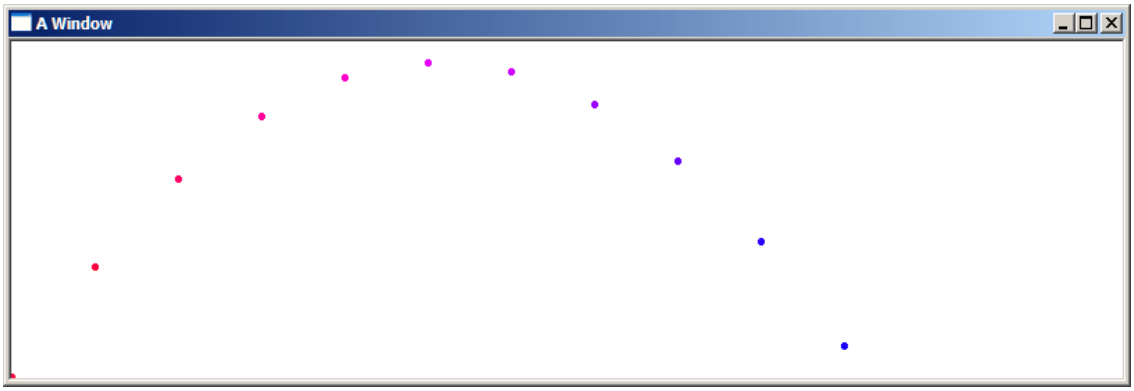
To control the colours in this way, you could make use of a handy library function: `set_pen_color(R, G, B)`. The three parameters, R, G, and B are numeric values anywhere between 0.0 and 1.0. They specify the redness, greenness, and blueness of a colour.

`set_pen_color(1.0, 0.0, 0.0)` is pure red
`set_pen_color(0.0, 0.0, 1.0)` is pure blue
`set_pen_color(0.7, 0.0, 0.3)` is a blueish red
`set_pen_color(0.3, 0.0, 0.7)` is a reddish blue
`set_pen_color(0.0, 0.0, 0.0)` is black
`set_pen_color(1.0, 1.0, 1.0)` is white, and so on.

5. *First Flight.*

Even with the dots gradually changing colour as time increases, that vertical picture doesn't give a very good representation of a trajectory. Using height as the vertical axis at one pixel per foot works out quite nicely. Letting time provide the horizontal axis also makes a lot of sense. You'll need to scale the time axis somehow, as one pixel per second is useless. Even if the cannonball is fired upwards at the speed of sound, it only spends about 63 seconds in flight.

Make it happen.



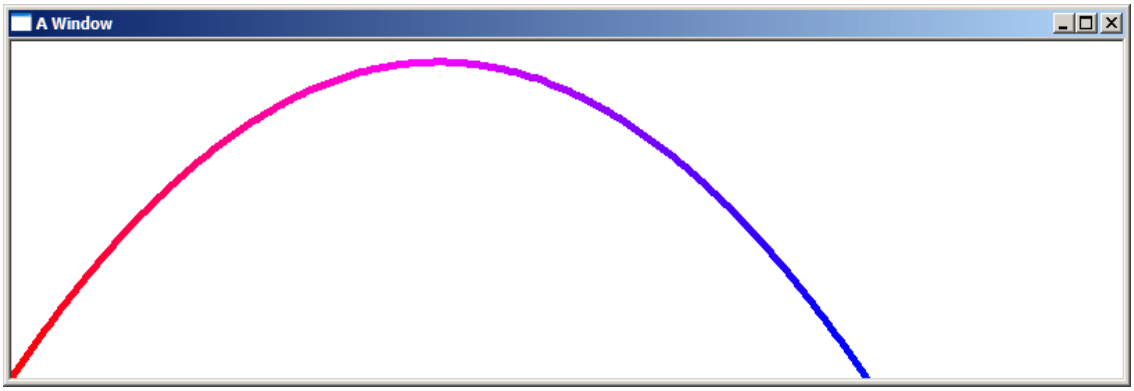
None of these steps should require big changes to your program, nor should they be making your program big or hard to follow.

If your program is getting out of hand, take a step back and think carefully about what every part of it is doing. If you have a lot of parts that seem very similar do what you can to clean things up. A little effort at clarity and structure in the beginning really pays off in the end.

6. *A Nice Arc.*

It seems absolutely trivial to convert the program to draw lines between the points. It is almost trivial, but there is one little problem that you'll have to deal with. The problem will become obvious when you try it. Try it now.

Once you have got the points joined by lines, you notice that one point per second isn't enough. See what you can do to produce a nice smooth accurate trajectory.



7. *Take the Battle to the Enemy.*

Sensible artillerymen do not fire their cannons straight up. The always set a cannon at an angle so that the cannonball will travel horizontally as well as vertically. If the cannon is set at angle α (0° = vertical, 90° = horizontal), the position equations are slightly altered:

$$h = v\,t\cos(\alpha) - \tfrac{1}{2}\,g\,t^2$$
$$d = v\,t\sin(\alpha)$$

Now, d represents the horizontal distance travelled. Remember that computers use radians instead of degrees; there are π radians to 180°.

Modify your program so that it now takes two inputs from the user before plotting the path. The first is still the initial velocity, the second should be the angle that the cannon is set at. Be kind to your user: accept the input angle in degrees, and have the program make the conversion.

In part 6, the vertical axis really did represent vertical height, maybe even at a very reasonable scale of 1 pixel = 1 foot. But the horizontal axis indicated time, which gives an unrealistic view. After this step, the horizontal axis really will indicate the horizontal position of the cannonball, perhaps again at a scale of 1 pixel = 1 foot. Just like a window onto nature.

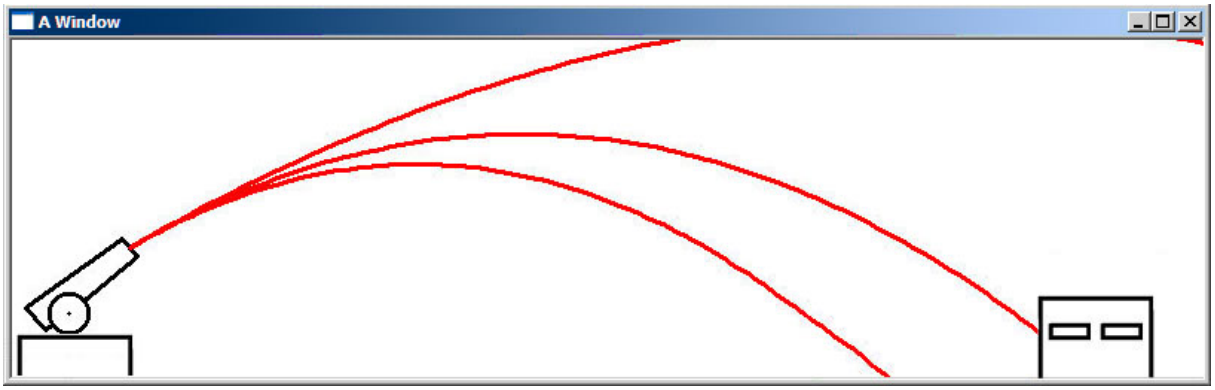
When the cannon-ball finally hits the ground, your program should print its final horizontal (x) position, and its total time in flight.

8. *The Game.*

Draw a simple representation of a cannon set at the right angle at one edge of the window, and a simple representation of the Enemy HQ at the other. If the cannonball hits the ground inside Enemy HQ, the user wins. Perhaps you could draw a little congratulatory explosion. Don't get obsessed by graphical perfection. It is only a game.

Use your cannon drawing function from lab 3. Making flexible, re-usable software is very important in the professional world, so you might as well get used to it.

Put the cannon on a built-up platform so we've got the advantage of height, and Draw the enemy HQ as an ugly squat bunker with a flat roof and two slits for windows. They are the enemy after all, their æsthetic sense is very poor.



9. *Make it Interesting.*

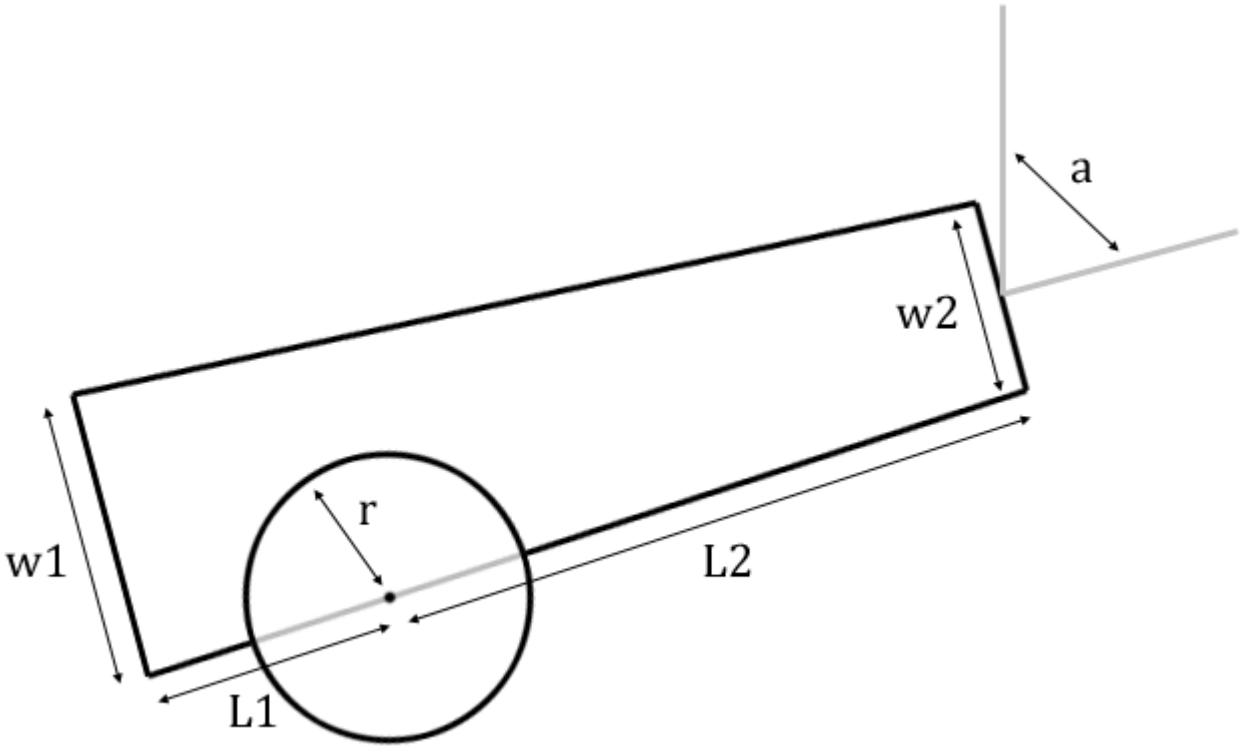
Give the user a few chances to pick the right velocity and angle, three to five tries seems reasonable. Make the HQ appear at a random position each time a new game is started, so the user can't just learn the right numbers. Perhaps even make it work in *real time*, so that a flight of five seconds really takes five seconds.

10. *OPTIONAL, For Extra Credit.*

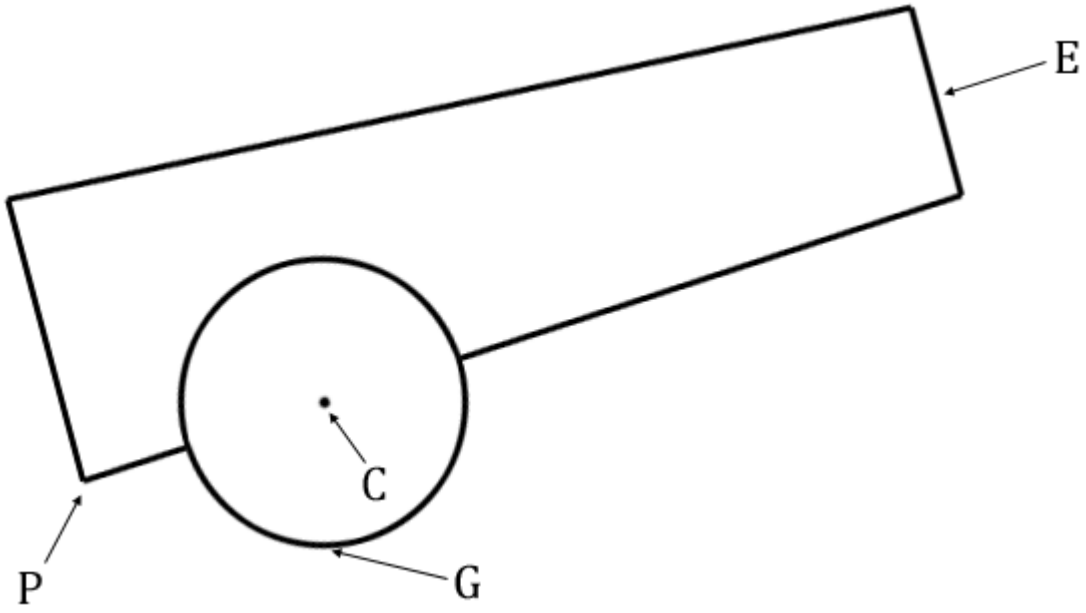
This was a popular game on Macintosh computers in the mid-1980s. Sell your version of it to Nintendo and become a millionaire. My cut is only 15%.

That's the END of the lab. The rest is just if you need a bit of help with the shape:

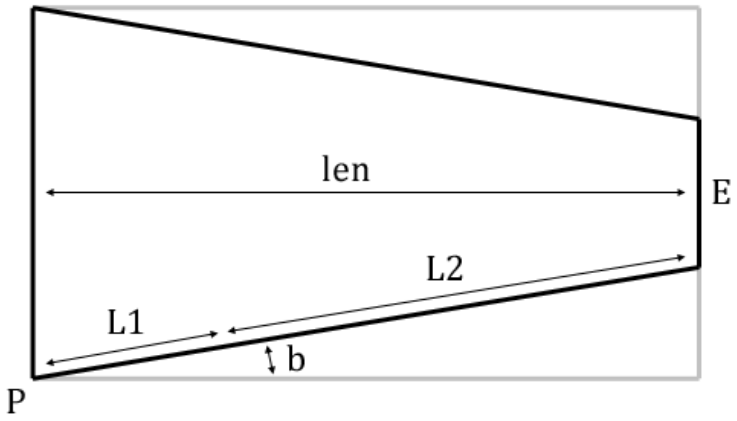
This is the cannon sitting on a wheel. The wheel's radius is r.



a is the aiming angle.
L1 is the distance from the back of the cannon to the wheel's axle, L2 is the rest of the length.
Like all cannons it is wider at the back (w1) than the front (w2).



G is the point on the ground where the wheel rests. Its coordinates are (xg, yg).
C is the exact position of the axle, its coordinates are (xc, yc).
P is the easiest point to start drawing the body of the cannon from, coordinates (xp, yp).
E is the point where the ball pops out when it is fired, coordinates (xe, ye).



This is a simplified picture of the body of the cannon shown with its “bounding box”. The point is to illustrate the difference between the real length of the cannon (len) and the sum L1+L2.

The angle shown as b is also helpful when drawing the shape. When the cannon is aimed at angle a, the heading for the bottom line is (a-b). Don’t forget that all angles are computed in radians.

```
xc = xg
yc = yg - r

b = asin((w1-w2)/2/(L1+L2))

xp = xc - L1 * sin(a-b)
yp = yc + L1 * cos(a-b)

len = (L1+L2) * cos(b)
```

Finally, to find the point E, we need two extra values:

d is the distance between points P and E

g is the angle from point P to point E if the cannon lies flat as in the third diagram.

```
d = sqrt(len*len + w1*w1/4)
g = asin(w1/2/d)

xe = xp + d * sin(a-g)
ye = yp - d * cos(a-g)
```