Contents

Version 0.5

Fixed the equation for Y. (It was missing the acceleration term).
Simplified the equation for VX since we don't have acceleration in X.

Added the section on converting polar to rectangular.

Typo fix in constraints wall location is at 12 inches.

V0.4: Needed 0.5 in the Y equation

V0.5: Edge cases: See **Note** at the end of the constraints section. The constraints themselves have been updated to match. At the edge or on the wall is on the table.

Physics & Dimensions

All of the variables that store the numbers in this section are to be stored in **doubles** unless otherwise indicated.

Dimensions

- The table is 24 inches wide (slightly larger than the old standard 22 inch table)
- X=0 is the centerline of the table in width
- The table is 48 inches deep
- Y=0 is the lower end of the table (towards the player / bottom of the screen)
- The table is angled at 7 degrees, giving a g force value of -47.052534795 inches per second per second in Y
- There is no acceleration in X
- Our simulation will use a time step of 1/64 of a second

Stored numerical data

A ball on the launcher (just been read in) needs:

- X
- Y
- Theta (angle of launch; 0 is pointed along the increasing X axis)
- Force (magnitude of launch velocity)

A ball in motion needs:

- X
- Y
- VX
- VY

Equations of motion

When updating the position of a ball, compute the variables in this order (position changes before velocity changes).

- New x = old x + (velocity in x * time interval)
- New y = old y + (velocity in y * time interval) + 0.5 *acceleration in y * time interval * time interval
- New velocity in x = old velocity in x ~~+ (acceleration in x * time interval)~~
- New velocity in y = old velocity in y + (acceleration in y * time interval)

Since VX doesn't change, you don't need to update it in your code.

Constraints

Do **not** check constraints in the same function that does basic motion. Make each the three constraint checks its **own** function. This lets you add walls one at a time. This lets you test motion before you implement any walls.

If the ball triggers a constraint condition, the program will print a message with appropriate details of what happened. It will fix the ball if need be.

- A magnitude (+/-) that is greater than 12 in X is invalid (side walls)
- Y > 48 is invalid (top wall)
- Y < 0 means the ball is off the table and out of play

Note: +/-12.0 exactly in X is still on the table. Y= 0.0 and Y = 48.0 are both still on the table. This avoids edge case issues in reflection.

Walls

When a ball goes past a wall, it must be fixed so that it reflects off the wall and is back on the table. Create a function for each wall that only deals in the reflection off that wall.

For example, here is how to deal with the ball having Y greater than 48:

- The new position is given as 48 less the amount it was above 48. (A ball one inch above the wall is now one inch below the wall.)
- The new VY is the negation of VY. (Hitting the wall changes direction of the ball.)
- Then multiply both VY and VX by 0.95 (hitting the wall robs energy)

Use similar mechanics for the side walls.

A ball sent to a corner will trigger two constraints. Check X constraints before checking Y constraints.

Leaving the table

The ball "drains" when Y is less than 0. This ball is out of play. You will need to provide a function that detects this and returns a Boolean. It's not a constraint, per see, but a condition that we need to be able to detect.

~~Update the status of the ball to be off the table. (Details for this are handled in the main sequence section).~~

Converting Polar to Rectangular

After a ball is read in, it needs to have the data converted to the form we will use.

We will make use of the C math library to help us do the conversion. To use it, we #include the appropriate header file to get the function declarations:

```
#include <math.h>
```

We also need to link in the math library, so we have to change our makefile:

```
pmov1: pmov1.o bits.o output.o physics.o
        gcc -g -o $@ $^  -lm
```

The above makefile entry is for a prototype that tests movement code. Note the -lm. Dash lower case L means that the linker should bring in a library and "m" is the short name for the math library. Once we have the C math library onboard, we have access to the trig functions sin() and cos() . These take an angle in radians and return the appropriate trigonometric value for that angle.

There is a problem with math.h in that gcc won't provide us the value of **`M_PI`** to us even though it is in the header file. I have extracted the correct value from math.h and placed it in the file `constants.h` which is available in Piazza. Multiply degrees by pi and divide by 180.0 to get radians.

What we've been calling "force" is really the magnitude of the initial velocity. It is directly related to force, but we have simplified the conversion away. For our purposes we multiply the "force" by sin of the angle to get VY and we multiply the "force" by the cos of the angle to get VX. *When making this conversion on the numbers in the array, be sure to use temporary values if your array has 4 slots. The theta value will get overwritten by VX and the force value will get overwritten by VY. Compute both before updating the array.* It might be good to have a function that does the conversion, and that should probably live with the physics functions for lack of a better place.

Separation of concerns

All of the code mandated here needs to be in a single file. Think of it as “all physics happens here.” Anything that knows the equations of motion or the sizes belongs in this file. As much as possible, “nothing but physics happens here.”

Magic Numbers

All “magic” numbers in the dimensions section must be #define symbols with useful names. They do not go in a header file since there are restricted to being used in only one C code file.