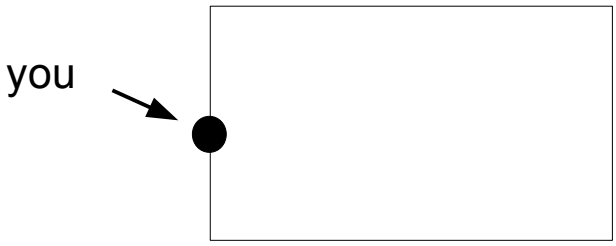
Pygame Part 2: The Electrical Rays Project

This set of worksheets is special: You won't be learning new Python words or pygame modules or functions or objects.

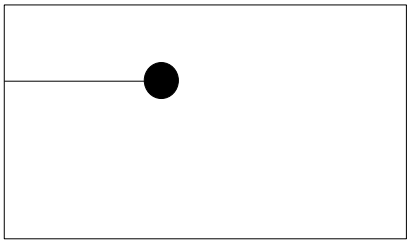
This set of worksheets has only one goal: to work on a project. I've provided hints to move you along. But for this project to be fun, it should be challenging. Therefore I try not to give you too much information. If you do need help, just ask.

A Popular 80s Game

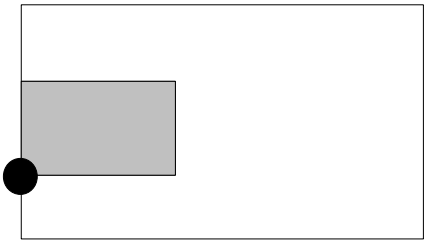
Long time ago ... (as in about 20 year ago) ... there was a popular video game that goes like this: In the screen, you are a dot and you can move along the boundary of the screen.



You can also move vertically or horizontally out of the boundary. By moving out and then back to the boundary, you can draw rectangles:

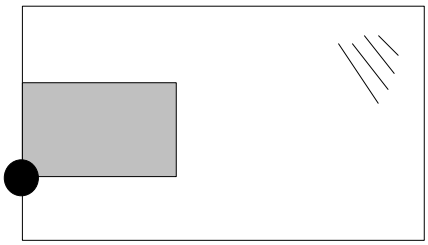


You move right, out of the boundary ...



You then move down and then left to capture a rectangular area.

Well ... obviously this game is not too challenging ... in fact it's pretty dumb. That's because there is no “adversary” – an enemy – or obstacles. That's because I haven't told you that while you are capturing areas a bunch of lines are bouncing in the un-captured area of the screen. The bunch of lines are spaced out neatly like this:



When the evil electrical signals hit you, you lose a life. You have altogether three lives. Your score is based on the total area you have captured.

The animation of the bunch of lines have also been used for screen savers. (It's also one of the first few animation programs I wrote when I was in high school.)

In this set of worksheets, you will learn to write a program to animate the evil electrical signals.

We will not be completing the whole game. But in the next set of worksheet, I will show you how to get keyboard inputs for a pygame program. This is different from `input` and `raw_input`. So after the next set of worksheets, with perseverance, you might be able to complete the game on your own.

A Single Line

Actually you already know how to animate a single line – refer to the previous set of worksheets. This is the program:

```
import pygame
pygame.init()

WIDTH, HEIGHT = 640, 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

violet = pygame.Color("violet")
black = (0,0,0)

def move(d, v, m):
    d = d + v
    if d > m:
        d = m
        v = -v
    elif d < 0:
        d = 0
        v = -v
    return d, v

x, y = 50, 60
xspeed, yspeed = 2, 1

X, Y = 600, 300
Xspeed, Yspeed = 1, 1

while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    x, xspeed = move(x, xspeed, WIDTH - 1)
    y, yspeed = move(y, yspeed, HEIGHT - 1)
    X, Xspeed = move(X, Xspeed, WIDTH - 1)
    Y, Yspeed = move(Y, Yspeed, HEIGHT - 1)

    surface.fill(black)
    pygame.draw.line(surface, violet, (x,y), (X,Y))
    pygame.display.flip()
```

Error Messages

You know that you're supposed to run the pygame program by double clicking on the icon of the file. Now for your non-pygame programs, when you have an error, the Python shell will show you an error message. The error message is usually helpful for figuring out the problem in your program.

Exercise. Run this program and read the error message

```
print x
```

Does the error message help you figure out what's wrong with your program?

Exercise. Run this program and read the error message

```
z = 0
print x / z
```

Does the error message help you figure out what's wrong with your program?

You have already seen functions. In this example, I will give you a function that does not have any input or output. Read the code for the function carefully.

```
def foo():
    x = 1
    z = 0
    print x / z
    return

foo()
print "back from foo ..."
```

Note that even when there is no input, you must still have the parentheses, i.e. `()` with nothing in between. Notice also that since there is no output, you don't see any value or expression after the word `return`.

Save the program as t.py. Run the program and you get this error message:

```
Traceback (most recent call last):
  File "C:/Documents and Settings/Yihsiang Liow/Deskto
p/t.py", line 7, in -toplevel-
    foo()
  File "C:/Documents and Settings/Yihsiang Liow/Deskto
p/t.py", line 4, in foo
    print x / z
ZeroDivisionError: integer division or modulo by zero
>>>
```

Of course you see that the last two lines tell you the statement

```
print x / z
```

gives a "`ZeroDivisionError: integer division or modulo by zero`" hints to you that you are dividing x by 0, i.e. z is 0.

But notice that there is more information. It tells you how you got to this statement. Look for the words "toplevel" and "foo". "toplevel" refers to the part of your program at the highest level – the code not inside any block.

```
def foo():
    x = 1
    z = 0
    print x / z
    return

foo()
print "back from foo ..."
```

The "foo" tells you that from the "toplevel" you entered a function block called "foo".

Not only that ... Look at the error message again. It tells you that you are going to foo at line 7 of t.py. Look at your program. At line 7 you have

```
foo()
```

And that the error occurs in line 4 of t.py which is the line

```
print x / z
```

So let me make a summary of what Python is trying to tell you.

Python is saying that you have a division-by-zero error at line 4 of t.py which is the statement

```
print x / z
```

and you arrive there from line 7 of t.py which is

```
foo()
```

Exercise. In the same folder (for instance on your Desktop) create the following files:

```
# this file is a.py

def f(x):
    print "in f ..."
    z = 0
    print X / z   # note: uppercase X
```

Now write the following program, save it as b.py and run it inside IDLE:

```
import a
a.f(5)
```

You should get an error message similar to the following:

```
>>>
in f ...
Traceback (most recent call last):
  File "C:\Documents and Settings\Yihsiang
Liow\Desktop\b.py", line 2, in ?
    a.f(5)
  File "C:\Documents and Settings\Yihsiang
Liow\Desktop\a.py", line 6, in f
    print X / z   # note: uppercase X
NameError: global name 'X' is not defined
>>>
```

Try to interpret the error message. What is the error? Where is the statement causing the error? How did you arrive at this error?

As you can see error messages are extremely important: It's Python way of helping you find and correct your error(s).

Debugging Messages for Pygame Programs

Now the problem with a Python program that uses the pygame package is that you lose the error message because the window that Python uses to print error message closes immediately when there is an error since at that point the program ends its execution.

Write this progra and save it. Run this program by double-clicking on the file's icon.

```
import pygame
pygame.init()

WIDTH, HEIGHT = 640, 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

print x / 0
```

You might be able to see the error message for a split second before closing.

Of course you can run this in IDLE. You do get the error message. The problem is that you will need to use the Task Manager to close pygame's window.

Here's a way to get around this. You can tell Pygame not to print the error messages to that black window (the console window), but rather to a ***file***. In fact there are two kinds of output to the console window. You see when you do a print statement, the stuff that you print is actually sent to a file. The file is associated with the console window. When Python prints error messages, that is also sent to a file and this file is also associated with the console window.

But ....

You can tell Python to associate these two files to files ***on your hard drive***. So if Python prints an error message or if you print information using the print statement, the output is sent to two files so that you can read them later. The file you print to is called stdout; the one Python uses for error messages is called stderr. This is how you tell Python to store the messages to files. I'm going to call the files

stdout.txt and stderr.txt respectively.

Run this program:

```
import sys, pygame
pygame.init()

sys.stdout = file("stdout.txt", "w")
sys.stderr = file("stderr.txt", "w")

WIDTH, HEIGHT = 640, 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

print "hello world"
print x / 0
```

(If you don't recall about files, you might want to refer to your previous worksheets.)

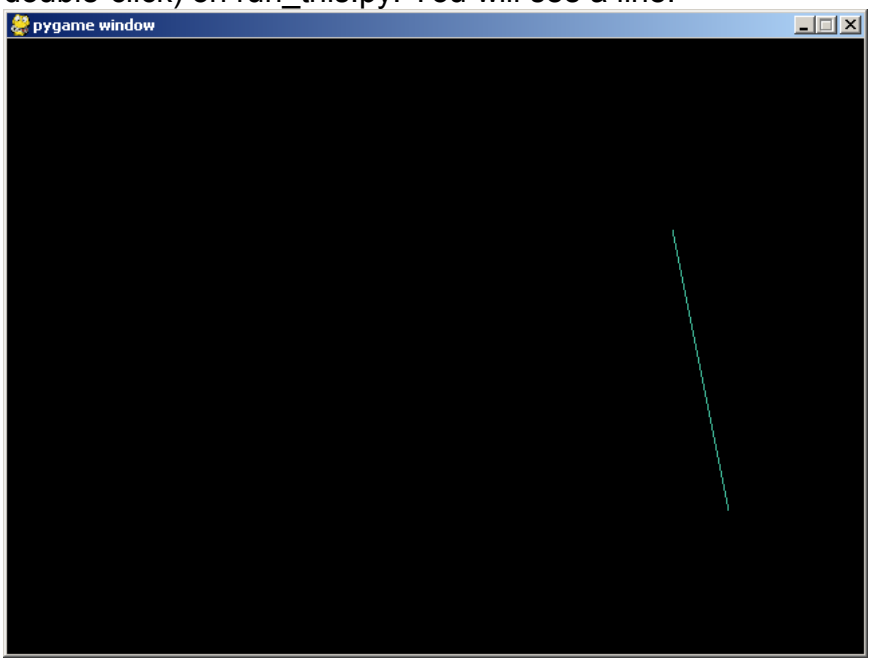
Run the program. Once the program stops, look for the files stdout.txt and stderr.txt.

You have a print statement in the program which should be sent to stdout.txt. The statement `print x / 0` should generate a division-by-zero which should be printed to stderr.txt.

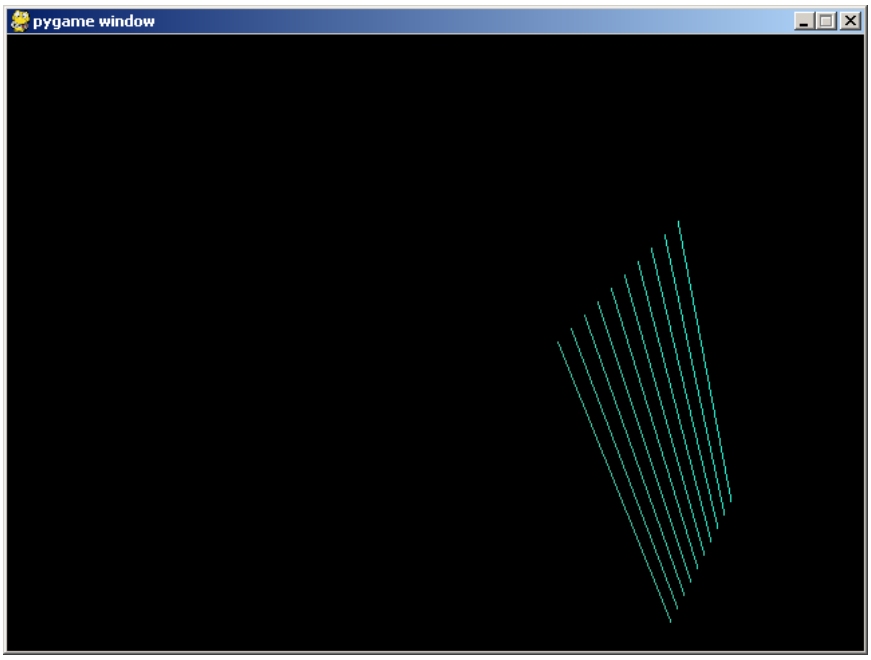
By the way note that the file variables sys.stdout and sys.stderr are found in the sys module.

The Electrical Rays Program

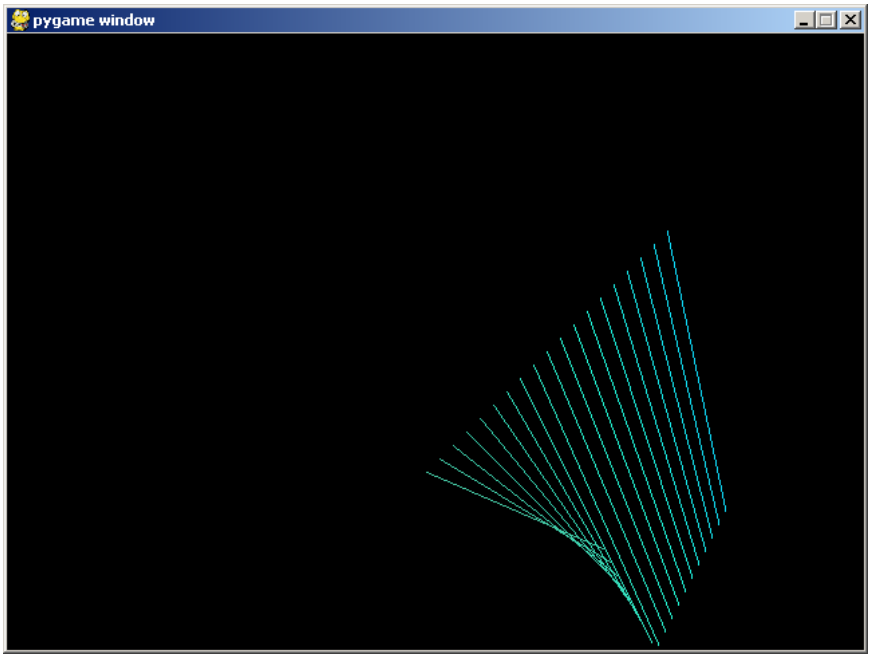
The first thing to do is to go to our web site https://bilbo.ccis.edu/plone/Members/yliow/ccpc/fall06 and download pygame_rays.zip. Unzip the file and run (i.e. double-click) on run_this.py. You will see a line:



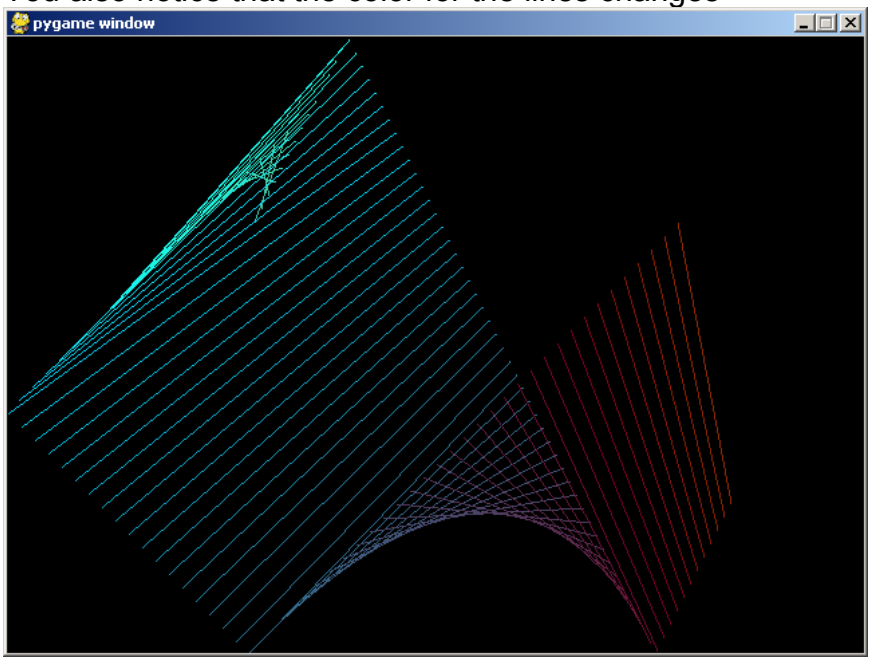
and more lines will follow:



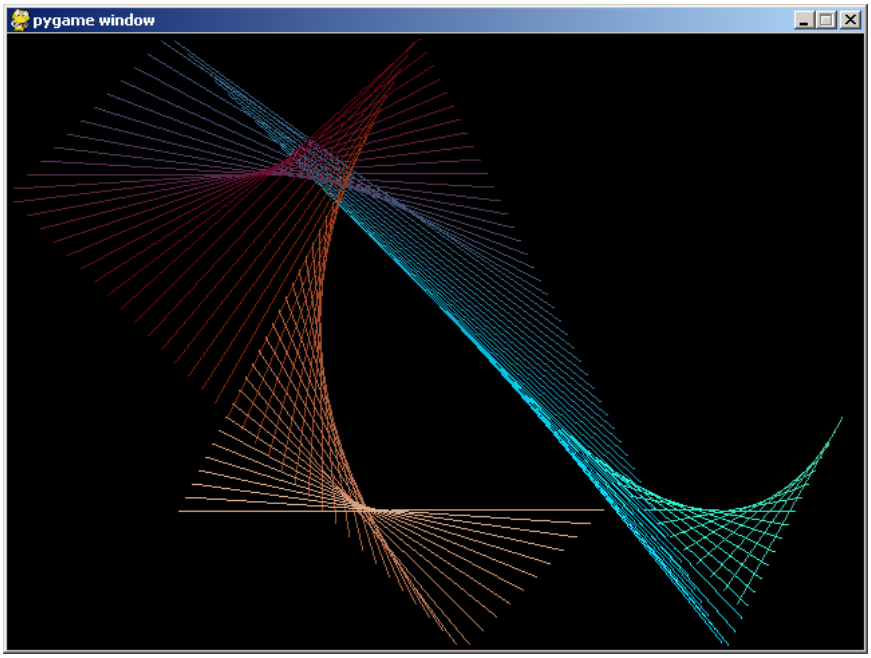
The lines bounce within the screen:



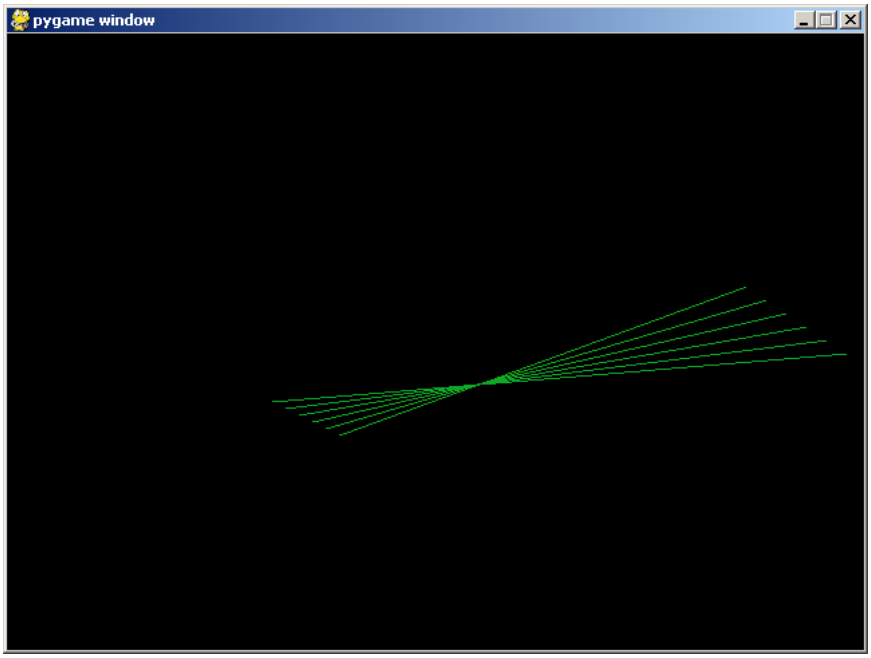
You also notice that the color for the lines changes



(Your worksheets are in black-and-white so you won't see the colors.) You also notice that after a while no new lines are created:



You also notice that the next time you run the program, the lines begin at a new position



Analyzing the Program

First of all, whenever you are given the description of the program to write (AKA requirements), try to see if you can build a series of programs that move towards the real target. Do not try to build what is needed in one step. Of course this applies only if you don't see how to write the program quickly.

What would you begin with? Of course a first step would be a build a program with only one line. That's the reason why I did the bouncing line program in the previous set of worksheets.

What's the next thing to do? Sometimes there are several different paths to reach the target program. It's not a science. You just have to make a reasonable decision.

For instance you might want to modify your bouncing line program so that it starts at a different position each time you run the program. Or maybe you want to modify the program to have two lines and then think about starting them at a random position on the screen.

Or maybe you can to begin with randomizing the colors.

So there are many different ways to proceed.

Occasionally you might find that you made a wrong decision. In that case you have to plan a new strategy.

Let's try to modify the program so that you get two lines. Let's worry about randomizing the starting positions and colors later. So our strategy is this:

- Improve the program to have two bouncing lines.
- Improve the program to have any number of lines.
- Improve the program to have the lines come out at a random place.
- Improve the program to have gradually changing colors.
- ... and we're done!

Among software engineers we would say that we're planning 4 iterations of the program.

Iteration 1: Two lines

You think ... “There are two lines. Bouncing two lines is no big deal. BUT ... it's not just two lines ... the first line follows the second. Hmmm ...”

Let's think... Imagine you're Python. In the game loop, you move line 1 (or rather you move the end points of line 1), you clear the screen with blank, you draw line 1. That's the logic of the program for the single bouncing line.

Now obviously the second line doesn't show ... yet.

But if line 2 does show, it behaves like line 1 a small time period earlier.

Whenever you don't see the logic of a loop, you do something called “unrolling the loop”.

```
move line 1 (first run of the body of the loop)
clear screen
draw line 1

move line 1 (second run of the body of the loop)
clear screen
draw line 1

move line 1 (third run of the body of the loop)
clear screen
draw line 1
```

And now we “release” line 2 ...

```
move line 1 and line 2 (fourth run)
clear screen
draw line 1 and line 2

move line 1 and line 2 (fifth run)
clear screen
draw line 1 and line 2
```

Note that line 1 and line 2 behave in the same way except that line 2 starts later than line 1. Let's see everything together:

```
move line 1
clear screen
draw line 1

move line 1
clear screen
draw line 1

move line 1
clear screen
draw line 1

move line 1 and line 2
clear screen
draw line 1 and line 2

move line 1 and line 2
clear screen
draw line 1 and line 2
```

In this case I'm releasing the second line after three execution of the loop. So I need to keep count on how many times I've been running the body of the game loop. Furthermore note that sometimes the execution of the body can look like this:

```
move line 1
clear screen
draw line 1
```

or

```
move line 1 and line 2
clear screen
draw line 1 and line 2
```

And which version to use depends on my count. AHA! So the game loop looks like this:

```
count = 0
while 1:

    if count < 3:
        move line 1
    else:
```

```
            move line 1 and line 2

        clear screen

        if count < 3
            draw  line 1
        else
            draw line 1 and line 2

        count = count + 1
```

You can also write this: Note that “move line 1” appears twice. Same for “draw line 1” Clear it up!

```
    count = 0

    while 1:

        move line 1
        if count >= 3:
            move line 2

        clear screen

        draw line 1
        if count >= 3:
            draw line 2

        count = count + 1
```

Of course you need to initialize the end points of the lines (their positions and their speeds). Since line 2 follows line 1, the positions and speeds of line 2 should be the same as line 1.

You note that also the `count` variable count the number of times the body of the while-loop has been execute, technically speaking we only need to keep the count up to 3. So this is what we have:

```
    initialize the positions and speeds of the endpoints
    of line 1 and line 2

    count = 0
```

```
while 1:

        move line 1
        if count >= 3:
                move line 2

        clear screen

        draw line 1
        if count >= 3:
                line 2

        if count < 3:
                count = count + 1
```

Now ... it's your turn ... get your hacking juice running!

Note that if the lines are too close to each other you might need to release the second line when count reaches a bigger number; I recommend 10.

The answer is on the next page ... DON'T PEEK WITHOUT SOME DD (DUE DILIGENCE)!!!

Answer to Iteration 1

Here's my version. You might want to compare it with yours.

```
import pygame, sys
WIDTH, HEIGHT = 640, 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

#sys.stdout = file("stdout.txt", "w")
#sys.stderr = file("stderr.txt", "w")

violet = pygame.Color("violet")
black = (0,0,0)

def move(d, v, m):
    d = d + v
    if d > m:
        d = m
        v = -v
    elif d < 0:
        d = 0
        v = -v
    return d, v

# End points for line 1
x, y = 50, 60
xspeed, yspeed = 2, 1
X, Y = 10, 200
Xspeed, Yspeed = 1, -2

# End points for line 2
a, b = 50, 60
aspeed, bspeed = 2, 1
A, B = 10, 200
Aspeed, Bspeed = 1, -2

count = 0

while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    # Move line 1
    x, xspeed = move(x, xspeed, WIDTH - 1)
    y, yspeed = move(y, yspeed, HEIGHT - 1)
    X, Xspeed = move(X, Xspeed, WIDTH - 1)
```

```
    Y, Yspeed = move(Y, Yspeed, HEIGHT - 1)

    # Move line 2 is count reaches 3
    if count >= 10:
        a, aspeed = move(a, aspeed, WIDTH - 1)
        b, bspeed = move(b, bspeed, HEIGHT - 1)
        A, Aspeed = move(A, Aspeed, WIDTH - 1)
        B, Bspeed = move(B, Bspeed, HEIGHT - 1)

    surface.fill(black)

    # Draw line 1
    pygame.draw.line(surface, violet, \
                     (x,y), (X,Y))

    # Draw line 2 if count reaches 3
    if count >= 10:
        pygame.draw.line(surface, violet, \
                         (a,b), (A,B))

    pygame.display.flip()

    if count < 10:
        count = count + 1
```

Note that I'm using a,b to keep track of the positions of one of the end points of the second line and A,B for the second. The speed (or rather velocity) is given by aspeed,bspeed and Aspeed, Bspeed.

I've commented out the statements

```
#sys.stdout = file("stdout.txt", "w")
#sys.stderr = file("stderr.txt", "w")
```

Since my program is working and I don't need them anymore. I don't want to waste compute time on useless statements.

I've included comments to help you read the program.

Controlling Frame Rate

The following section contains a technique for controlling the "speed" of a program that applies to many types of games.

If you run Iteration #1 from the previous section you notice that the animation might not be smooth. The jerkiness is sometimes due to the fact that your Python program is sharing the computer with other programs.

The other problem is that if you run your program on a faster machine the lines will move faster – maybe too fast to be interesting.

To create a smoother animation, you need to run the body of the game loop in a regular way. In other ways you want to redraw the surface at a constant rate. This is called the **frame rate**.

This is how you do it. Suppose you want the program to run body of the game loop 60 times per second, all you need to do is to measure the amount of time needed to run the body, and add delay to match up to the correct time. Earlier on, you have seen that the `time` module has the `clock` function to measure time. Pygame has a similar function called `get_ticks()` which is in the `time` module in the `pygame` package. Time is measured in milliseconds.

The general picture is this:

```
FRAME_RATE = 1000.0 / 60

while 1:
    starttime = pygame.time.get_ticks()

    # do some animation

    endtime = pygame.time.get_ticks()
    totaltime = endtime - starttime
    timeleft = int(FRAME_RATE - totaltime)
    if timeleft > 0:
        pygame.time.delay(timeleft)
```

I measure the time before and after my animation code and

put them into `starttime` and `endtime`. The variable `totaltime` would then give me the total amount of time to execute the animation.

The final thing to do is to delay the program. The total time allocated for each execution of the body of the game loop is in `FRAME_RATE`. We want to execute 60 steps in 1 second, i.e. 1000 milliseconds. That's why `FRAME_RATE` is set to `1000.0 / 60`.

So the time we need to delay is `FRAME_RATE - totaltime`. We keep this in the variable `timeleft`. Now the `pygame.time.delay` function in pygame accepts an integer. That's why we use `int(FRAME_RATE - totaltime)`.

Exercise. Remember `int` function? If you don't try this exercise. First guess what output you will get when you run this program:

```
x = 1.2
y = 1.4
z = 1.6
a = int(x)
b = int(y)
c = int(z)
print a, b, c
```

Now run the program.

Finally we call `pygame.time.delay(timeleft)` if `timeleft` is greater than 0.

Now use this idea to force Iteration 1 to run at a constant frame rate of 60 frames per second.

The answer is on the next page.

Iteration 1 with Constant Frame Rate

Here you go ...

```
import pygame, sys
WIDTH, HEIGHT = 640, 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

sys.stdout = file("stdout.txt", "w")
sys.stderr = file("stderr.txt", "w")

violet = pygame.Color("violet")
black = (0,0,0)

def move(d, v, m):
    d = d + v
    if d > m:
        d = m
        v = -v
    elif d < 0:
        d = 0
        v = -v
    return d, v

# End points for line 1
x, y = 50, 60
xspeed, yspeed = 2, 1
X, Y = 10, 200
Xspeed, Yspeed = 1, -2

# End points for line 2
a, b = 50, 60
aspeed, bspeed = 2, 1
A, B = 10, 200
Aspeed, Bspeed = 1, -2

FRAME_RATE = 1000.0 / 60

count = 0

while 1:

    starttime = pygame.time.get_ticks()

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

```
    # Move line 1
    x, xspeed = move(x, xspeed, WIDTH - 1)
    y, yspeed = move(y, yspeed, HEIGHT - 1)
    X, Xspeed = move(X, Xspeed, WIDTH - 1)
    Y, Yspeed = move(Y, Yspeed, HEIGHT - 1)

    # Move line 2 is count reaches 3
    if count >= 10:
        a, aspeed = move(a, aspeed, WIDTH - 1)
        b, bspeed = move(b, bspeed, HEIGHT - 1)
        A, Aspeed = move(A, Aspeed, WIDTH - 1)
        B, Bspeed = move(B, Bspeed, HEIGHT - 1)

    surface.fill(black)

    # Draw line 1
    pygame.draw.line(surface, violet, \
                     (x,y), (X,Y))

    # Draw line 2 if count reaches 3
    if count >= 10:
        pygame.draw.line(surface, violet, \
                         (a,b), (A,B))

    pygame.display.flip()

    if count < 10:
        count = count + 1

    endtime = pygame.time.get_ticks()
    totaltime = starttime - endtime
    timeleft = int(FRAME_RATE - totaltime)
    if timeleft > 0:
        pygame.time.delay(timeleft)
```

On to Iteration 2!!!

Iteration 2

Now we need to extend the program to handle many more lines. The version I demo'd during the meeting has a maximum of ***100 lines***.

It's ***not*** a good idea to manually programmed all the 100 lines like what we did for the second line!!!

Remember the for-loop? We will use the for-loop to repeat the initialization, moving and drawing of the 100 lines. Note that we have to keep the lines somewhere. We will use lists.

Now make sure you make a copy of Iteration 1. Work on the copy.

Instead of trying to create a program for 100 lines, start by changing your current program so that it works with lists.

It's possible to have a list for the x value of the positions of one end point, a list for the y value of the positions of the same end point, another list for the x value of the positions of the second end point, a list for the ....

This will make your program harder to read. Another way would be to create a list of “lines”.

Now think about it. What makes up a line in our program? Look at line 1of Iteration1. Line 1 is made up of x, y, xspeed, yspeed, X, Y, Xspeed, Yspeed where (x,y) and (X,Y) denote two endpoints and (xspeed, yspeed) and (Xspeed, Yspeed) denote their speeds. There are 8 integer values. We can use a list of 8 integer values to denote a moving line.

Again: ***A moving line (for us) is a list of 8 integers***.

Let me just give you an extra example for practice. Here's an example of a list of list of integers:

```
bios = [[5.7,200], [6.1,130], [5.5,180]]
```

Suppose each item in x represents the height (in ft) and weight (in lbs) of a person. So we have three people who are being measured. Here's a for-loop that runs across the list

```
for bio in bios:
     print bio
```

Now try this and make sure you understand why the program produces the output:

```
for bio in bios:
     height = bio[0]
     weight = bio[1]
     print height, "ft and ", weight, "lbs"
```

Now think at the high level: Suppose you have a list of 100 lines. You need to know how to process (move and draw) say the first 5 of them. Recall that if `bios` is a list, then `bios[0:5]` (or `bios[:5]`) will give you the list consisting of the first 5 items from `bio`. Continuing the above example try this:

```
for bio in bios[:2]:
     height = bio[0]
     weight = bio[1]
     print height, "ft and ", weight, "lbs"
```

Of course you need to know how to add things into the list. Remember `append`? Here's how you create a list of 10 bio data with the same height of 5.5 and weight 200:

```
bios = []
for i in in range(10):
     bio = [5.5, 200]
     bios.append(bios)

print bios
```

Make sure you run the above examples.

I think I will want two iterations within Iteration 2: Iteration 2A will work just like Iteration 1 but with a list of two lines while Iteration 2B will be work with a list of 100 lines.

Let's get started with Iteration 2A.

Now let's hit Iteration 2. We start with the code from Iteration 1. First here is the code to create two lines:

```
# End points for line 1
```

```
x, y = 50, 60
xspeed, yspeed = 2, 1
X, Y = 10, 200
Xspeed, Yspeed = 1, -2

# End points for line 2
a, b = 50, 60
aspeed, bspeed = 2, 1
A, B = 10, 200
Aspeed, Bspeed = 1, -2
```

The numbers x, y, xspeed, yspeed, X, Y, Xspeed, Yspeed go into a list [x, y, xspeed, yspeed, X, Y, Xspeed, Yspeed] to describe the first line.

Modify this piece of code so that it creates a list of lines. Don't forget that a line is a list of 8 numbers. Here's a template for you to complete:

```
lines = []
for i in range(2):
    line = _______________________________
    lines.append(line)
```

That takes care of initializing the lines.

Notice that I need to know how many lines in the list of lines to move and then draw. So I need to remember that in a variable. Let's call this variable `num_lines`. Of course initially there is 1 line to process. Add this to the above code when you create `lines`. (Later when `count` reaches 10, `num_lines` will become 2. Right?)

Now let's go into the game loop. You have to modify the part that move the lines. You only need to move the lines in `lines[:num_lines]`. Here's the code you have to complete:

```
for line in lines[:num_lines]:
    # Move line
    # (Use line[0], line[1], ... instead
    # of x, y, ...)
```

And for your reference, here's the code for moving line 1 from Iteration 1:

```
    # Move line 1
    x, xspeed = move(x, xspeed, WIDTH - 1)
    y, yspeed = move(y, yspeed, HEIGHT - 1)
    X, Xspeed = move(X, Xspeed, WIDTH - 1)
    Y, Yspeed = move(Y, Yspeed, HEIGHT - 1)
```

You have to do the same for drawing lines of course. This is the code you have to complete:

```
for line in lines[:num_lines]:
    # Draw line
    # (Use line[0], line[1], line[2], ...
    # instead of x, y, X, Y.)
```

For your reference this is the code for drawing line 1

```
    pygame.draw.line(surface, violet, \
                     (x,y), (X,Y))
```

There's one thing extra you have to do that's not done in Iteration 1. And that is to update `num_lines`. You need to set `num_lines` to 2 when `count` reaches 10.

If you follow the above you should be able to rewrite Iteration 1 with lists. Now ... hit the code ...

Solution to Iteration 2A

I have to use a smaller font size because some lines were too long.

```
import pygame, sys
pygame.init()

WIDTH, HEIGHT = 640, 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

sys.stdout = file("stdout.txt", "w")
sys.stderr = file("stderr.txt", "w")

violet = pygame.Color("violet")
black = (0,0,0)

def move(d, v, m):
    d = d + v
    if d > m:
        d = m
        v = -v
    elif d < 0:
        d = 0
        v = -v
    return d, v

lines = []
for i in range(2):
    line = [50, 60, 2, 1, 10, 200, 1, -1]
    lines.append(line)

FRAME_RATE = 1000.0 / 60

count = 0
num_lines = 1

while 1:

    starttime = pygame.time.get_ticks()

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    # Move lines in lines[:num_lines]
    for line in lines[:num_lines]:
        line[0], line[2] = move(line[0], line[2], WIDTH - 1)
        line[1], line[3] = move(line[1], line[3], HEIGHT - 1)
        line[4], line[6] = move(line[4], line[6], WIDTH - 1)
        line[5], line[7] = move(line[5], line[7], HEIGHT - 1)

    surface.fill(black)

    # Draw lines in lines[:num_lines]
    for line in lines[:num_lines]:
        pygame.draw.line(surface, violet, (line[0],line[1]), \
                         (line[4],line[5]))

    pygame.display.flip()

    if count < 10:
        count = count + 1
```

```
    if count >= 10:
        num_lines = 2

    endtime = pygame.time.get_ticks()
    totaltime = starttime - endtime
    timeleft = int(FRAME_RATE - totaltime)
    if timeleft > 0:
        pygame.time.delay(timeleft)
```

Iteration 2B

If you get Iteration 2A to work correctly, then 2B is easy.

Instead of two lines, create 100 lines in the variable `lines`. `num_lines` is still initially set to `1`.

The moving and drawing part is already taken care of ... believe it or not. The only thing left is the control of releasing the lines.

There are two variables involved in releasing lines: `count` and `num_lines`.

We want to increase num_lines by 1 after 10 execution of the body of the game loop. To put it simply, when I unroll the loop this you should see this values for `count` and `num_lines` at the top of the body of the game loop:

```
count = 0, num_lines = 1
count = 1, num_lines = 1
count = 2, num_lines = 1
count = 3, num_lines = 1
count = 4, num_lines = 1
count = 5, num_lines = 1
count = 6, num_lines = 1
count = 7, num_lines = 1
count = 8, num_lines = 1
count = 9, num_lines = 1
count = 0, num_lines = 2
count = 1, num_lines = 2
count = 2, num_lines = 2
count = 3, num_lines = 2
count = 4, num_lines = 2
count = 5, num_lines = 2
count = 6, num_lines = 2
count = 7, num_lines = 2
count = 8, num_lines = 2
count = 9, num_lines = 2
count = 0, num_lines = 3
count = 1, num_lines = 3
```

etc. Previously `count` is like a timer between line 1 and line 2. Now `count` has to be a timer between line 2 and line 3,

line 3 and line 4, etc.

The structure of the program should look like this:

```
create lines
num_lines = 1
count = 0

while 1:
     move things in lines[:num_lines]
     clear surface
     draw things in lines[:num_lines]
     flip surface
     update count and num_lines
     insert time delay
```

Now focus your on "`update count and num_lines`".

Think about when should `count` be incremented (i.e when do you execute `count = count + 1`)?

```
increment count
if count is 10:
     increment num_lines
     set count to 0
```

Think about the logic and make sure you understand it.

There is a detail. Notice that the above logic will keep increasing the number of lines to be set "alive". You only want to do that when `num_lines` is less than 100. We also do not increment count when `num_lines` has already reached 100.

Solution to Iteration 2B

```
import pygame, sys
pygame.init()

WIDTH, HEIGHT = 640, 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

sys.stdout = file("stdout.txt", "w")
sys.stderr = file("stderr.txt", "w")

violet = pygame.Color("violet")
black = (0,0,0)

def move(d, v, m):
    d = d + v
    if d > m:
        d = m
        v = -v
    elif d < 0:
        d = 0
        v = -v
    return d, v

lines = []
for i in range(100):
    line = [50, 60, 2, 1, 10, 200, 1, -1]
    lines.append(line)

FRAME_RATE = 1000.0 / 60

count = 0
num_lines = 1

while 1:

    starttime = pygame.time.get_ticks()

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    # Move lines in lines[:num_lines]
    for line in lines[:num_lines]:
        line[0], line[2] = move(line[0], line[2], WIDTH - 1)
        line[1], line[3] = move(line[1], line[3], HEIGHT - 1)
        line[4], line[6] = move(line[4], line[6], WIDTH - 1)
        line[5], line[7] = move(line[5], line[7], HEIGHT - 1)

    surface.fill(black)

    # Draw lines in lines[:num_lines]
    for line in lines[:num_lines]:
        pygame.draw.line(surface, violet, (line[0],line[1]),
(line[4],line[5]))

    pygame.display.flip()

    if count < 10 and num_lines < 100:
        count = count + 1

    if count >= 10:
        if num_lines < 100:
            num_lines = num_lines + 1
            count = 0

    endtime = pygame.time.get_ticks()
```

```
    totaltime = starttime - endtime
    timeleft = int(FRAME_RATE - totaltime)
    if timeleft > 0:
        pygame.time.delay(timeleft)
```

Iteration 3

This is easy. Modify the program so that each time you start the program, the lines are released from a random place. Not only that, the speeds of the end points should also be random ... of course the speeds shouldn't be too large.

Go ahead ... and do it!!!

By the way you should allow negative speeds. So for instance you might want to choose randomly a value from -3, -2, -1, 0, 1, 2, 3 for xspeed. This is easy: it's just `random.randrange(-3, 4)`. Right? But note that if `xspeed` is 0, then `x` will not change. So you probably want to randomly select from -3, -2, -1, 1, 2, 3 (i.e. ***do not use 0***). There are many ways of doing that ...

Exercise. This method has to do with selecting a “sign” and then a positive number between 1 and 3 (inclusive). Try running this a couple of times:

```
import random
print random.randrange(-1, 2, 2)
```

Now try this a couple of times:

```
import random
print random.randrange(-1, 2, 2) * \
      random.randrange(1, 4)
```

Exercise. Try this a couple of times:

```
import random
names = ["Halo", "Warcraft", "Unreal"]
print random.choice(names)
```

If you don't “get it”, go to Python's documentation, look for the random module documentation and look for the `choice` function. Why should the `choice` function help?

Iteration 4

Now make the program change the color of the lines gradually. I leave it to you complete. Run my program several times, think about how you want to vary the color, and code it. Ask questions if you need help.

Two hints are given below. Use them only when you're stuck.

Hint: Here's a hint: Each line is not just a list of 8 numbers. It should be a list of 8 numbers and a color. Don't forget that a color is just a tuple of 3 numbers (R, G, B) where R, G, and B are between 0 and 255 (inclusive).

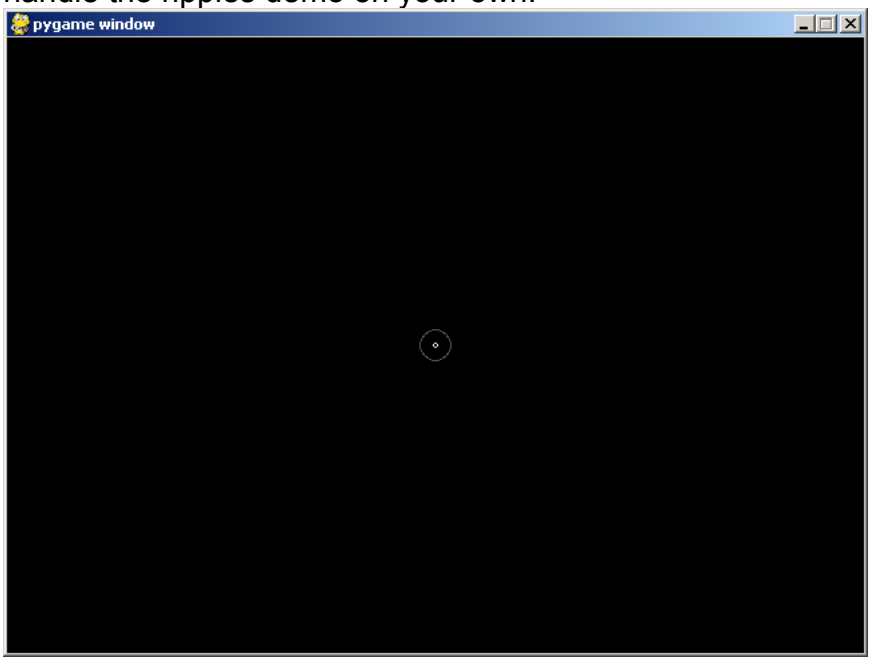
Hint: Run the program and observe the colors of the lines. Notice that the color changes gradually from a line to the next. Also, note that the color of a line does ***not*** change. Two colors are very close if their R, G, B values are close. So you need to choose starting values for R, G, B. Each time you create a line to be placed into `lines` you need to vary R, G, and B slightly. Of course you need to make sure that R, G and B stay within 0 to 255. HEY! That's exactly like change the x value of an endpoint of a line! This means that you can probably use the `move` function in the code. So in the creation of `lines`, your code should look something like this:

```
compute x, y, xspeed, ...
R = some random value between 0 and 255
Rspeed = some small random value (say
         between -3 and 3, excluding 0)
Same for G, Gspeed, B, Bspeed

lines = []
for i in range(100):
     compute new R, Rspeed using move
     compute new G, Gspeed using move
     compute new B, Bspeed using move
     color = (R, G, B)
     line = [x, y, ..., color]
     lines.append(line)
```

Ripples Project and Looking Ahead

Once you're done with this project, you should be able to handle the ripples demo on your own.



Next week we'll talk about rects, images (actually from the bouncing alien program you already know how to blit them), and the keyboard. I'll also talk briefly about collision detection, i.e. when do two “things” appear to collide on the screen? Finally I will show you how to play sound or music.