Reading Assignment

- Chapter 10, 11
- There will be clicker questions on the reading
- Start on Chapter 12 - next lecture will start on it, but it is long!

Do you have Volume 2 of the text?

A) Yes

B) No

Saving Plots

- You can grab “screenshots” and save them in jpg, png, etc. formats with `tvrd()`, but
  - limited resolution / poor quality
  - not “vectorized”
    - “vectorized” graphics take less memory and have higher resolution, but only work for line drawings

“plot device” / “backend”

- `set_plot,”ps”`: After running this command, your graphics will no longer show up in the plot windows, but will be saved to a `.ps` file
- However, you need to specify the output filename also:
  `device,filename=‘filename.ps’`
- Must close the file too:
  `device,/close`

Postscript

- Postscript has been the *de facto* standard for publication-quality plots for a while
- It is efficient for line plots, but inefficient for images
- It’s somewhat inconvenient to work with, but IDL does it well

Get back to normal...

- To return from postscript plotting to “X-Windows” plotting (the normal windows), do:
  - `set_plot,'x'` or
  - `set_plot,'win'`

SURVEY: How long did Assignment 3 exercises & WDIDs take?

A) < 1 hour

B) ~ 1 hour

C) ~ 2 hours

D) > 2 hours

E) I didn’t do the exercises & WDIDs

SURVEY: How long did Assignment 3 homework take?

A) < 1 hour

B) ~ 1 hour

C) ~ 2 hours

D) > 2 hours

E) I didn’t do / haven’t finished the homework (but if you know how long it will take you, answer one of the others)

SURVEY: Do you like the tutorials?

A) Yes, unequivocally

B) Yes, but they’re too long

C) No, but only because they’re too long

D) No, unequivocally

E) I do not have an opinion.

SURVEY: How much work is this class compared to other astro classes? (if you're not taking any, just don't answer)

A) A lot more

B) A little more

C) About the same

D) A little less

E) A lot less

Color

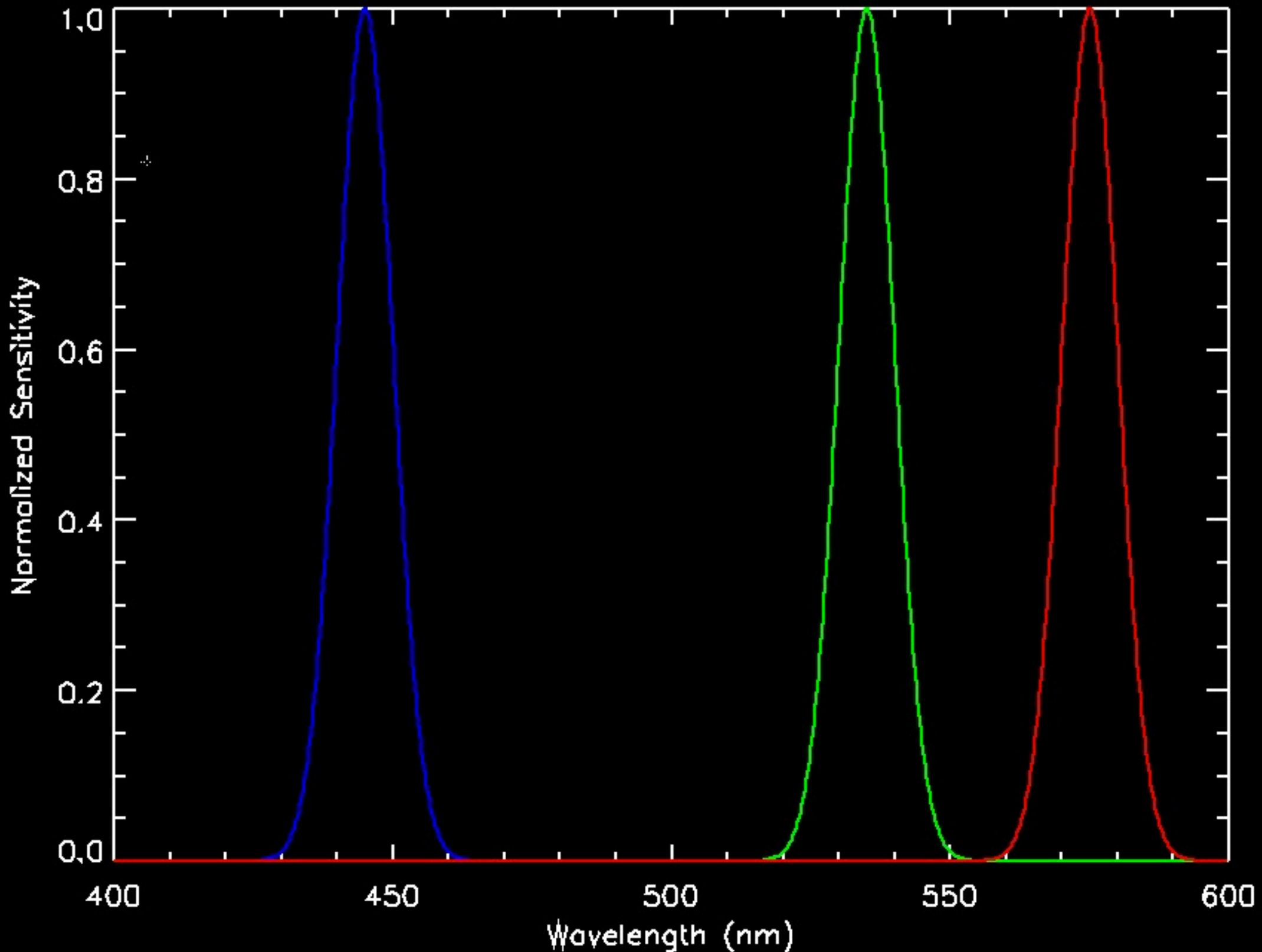
- Your eye can see more than black and white. Why not use it?
- well, it’s a little more work
- but it’s usually worth it

Your Eye

- Your eyes contain two different types of light sensitive cells, called “rods” and “cones” based on the cell shape
  - rods are more sensitive, but don’t distinguish color well
  - cones are less sensitive but sense colors

Cones

- There are three types of cones
  - red are most sensitive to 575nm light
  - green are most sensitive to 535nm light
  - blue are most sensitive to 445nm light


1.0
0.8
0.6
0.4
0.2
0.0
Normalized Sensitivity
400
450
500
550
600
Wavelength (nm)

Faking Colors

- Because your eye is only directly sensitive to 3 colors, your brain uses the relative brightness seen by the different cones to infer color
  - e.g., yellow would be something bright in the green cone and the red cone, but faint in the blue cone

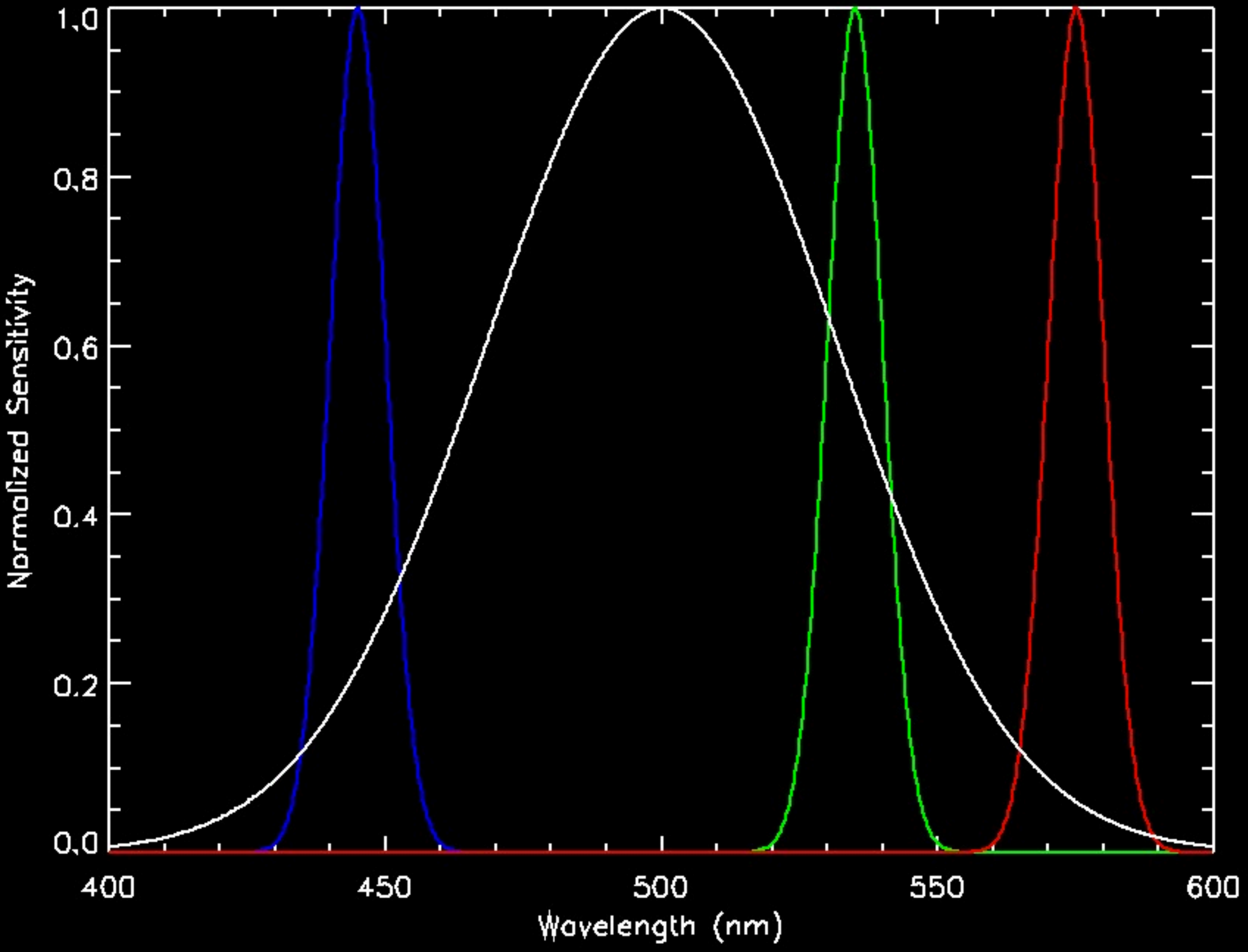
Faking Colors

- This is a nice feature: it means we only need 3 colors of pixels in our TVs / computers to display all colors our brains can interpret
- (white is just all 3 together)

Physiology Sidenote

- rods are not very important in daylight
- but they ARE very important at night!
  They are much more sensitive than cones, but have a much broader response curve
- rods are not very sensitive to red light, though

(note: these are approximate, not accurate!)



How do we make colors?

- Computers follow one of two color schemes (usually):
  - RGB = Red Green Blue
  - CMYK = Cyan Magenta Yellow blacK
- In IDL, we'll use RGB

IDL Color

- IDL actually has a very inconvenient color setup, BUT many people have written “wrapper” codes to get around this
- `color_convert` is one tool to convert between different color schemes

IDL Color

- Each color is identified by a single long integer
- 0000 0000 0000 0000 0000 0000 0000 0000
  - (alpha) BGR
- Each color can have a range varying from 0 to 255 (one byte)

Hex Colors

- It is more convenient to specify such large numbers with hexadecimal than integers
  - Range is 00 to FF

- BUT IDL is the opposite of the internet! (and I don't know which came first...)
  - Internet + python: FF0000, 0000FF
  - IDL: FF0000, 0000FF

Integer Colors

- color = red + green*256L + blue*65536L
  - red, green, blue are all in the range [0,255]

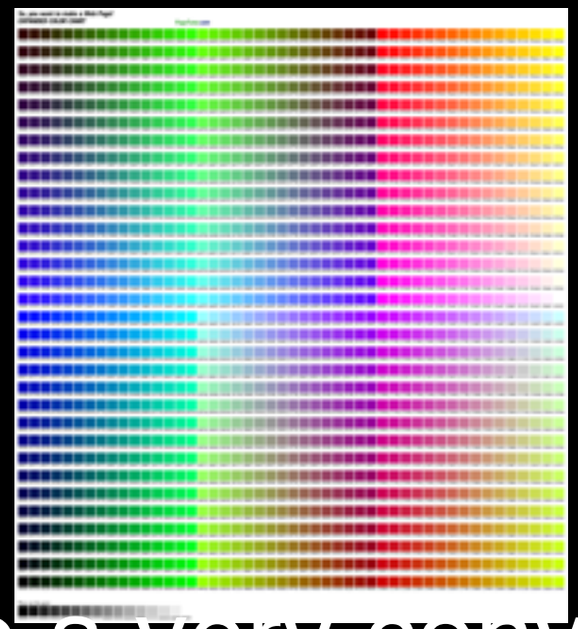
`tvrd` and color

- Since colors are represented by long integers (or a series of bytes), `tvrd` returns a 3D array if you tell it you want color

```
IDL> help,tvrd()
<Expression>    BYTE      = Array[1280, 709]
IDL> help,tvrd(true=1)
<Expression>    BYTE      = Array[3, 1280, 709]
```

("true" stands for "truecolor", a buzzword indicating that there are 16 million colors [3 bytes] available)

Hex Codes



- Hex Codes are a very convenient way to express color (in my opinion)
- If you google “hex color table”, you get lots of pretty-looking hits
- It’s easy to interpret:
  RRGGBB [internet+python] or
  BBGGRR [IDL]

What color is 00FF00?

A)

B)

C)

D)

E) None of the above

What color is FF0000? (in IDL)

A)

B)

C)

D)

E) None of the above

What color is FF00FF?

A)

B)

C)

D)

E) None of the above

What NUMBER is ‘0000FF’x?

A) 0

B) 255

C) 62580

D) 16711680

E) None of the above

Color: IDL v Python

- IDL:
  - `plot,x,color='0000ff'x`
  - `plot,x,color=255`
- python:
  - `plot(x,color='#ff0000')`
  - `plot(x,color='r')`
  - `plot(x,color=(1.0,0,0))`
    - "vector" of "all red, no green, no blue"

Color: 

- python allows “see-through” colors (i.e., they show the background color partially)
- Can use `alpha`, which says how opaque something is (1=opaque, 0=invisible)
- Or the color vector: `color=(R,G,B,α)`
- See http://matplotlib.org/examples/pylab_examples/color_demo.html

SCRIPTING

- Getting into stuff that qualifies as “pretty neat”
- also, standard programming practice (it’s great to learn on the interactive command line, but most languages don’t have one!)

What’s a script?

- Not a procedure or a function (those are coming soon)
- A script is a series of commands, just like you’d use them interactively
- You could copy and paste the contents of a script at the IDL prompt, and it would work

Why script?

- It’s more convenient to write a single command than copy and paste the whole contents of a file
- My script example will be `script.pro`
  - the `.pro` extension is important

How to call script

- `@script.pro` is the special syntax for calling a script
  - `@script` is also OK if your file is called `script.pro`
- If your script contains an error in it (e.g., a syntax error), it will do all of the lines that are OK from the start up until that error

Example Use

- Physical constants
  - You could save them all in a `.sav` file - that's actually a pretty good idea
  - OR you could include their definitions in a script
    - better because you can specify units & include comments

Sample Script

- `physicalconstants.pro`
  - No risk of typos!

```
1 kb=1.38e-23 ; Boltzmann's Constant [joules / K]
2 cs=2.997925e8 ; speed of light [ meters / s]
3 hh=6.626e-34 ; [joule sec] Planck's constant
```

Line Numbers

comments with units!

two-character constants are easier to search for than 1-character constants

Another example

- Remember reading all those files? And then having to re-do all those lines because you messed up a little bit?

- If you have a file you read in often - say, a spectrum you're going to use frequently - might be nicer to script opening it

```
; read the example spectrum from assignment 3
; first the flux array
openr,lun,'/home/shared/astr2600/data/assignment3/lif2bflux.dat',/get_lun

nn = 0L         ; first # is known to be long
readu,lun,nn ; read first 4 bytes of file into long(n)
nn = swap_endian(nn) ; swap the endianness [repeated for all vars]

yy = fltarr(nn)
readu, lun, yy  ; read the rest of the file into yy
yy = swap_endian(yy)

free_lun,lun ; close file

; read the wavelength
openr,lun,'/home/shared/astr2600/data/assignment3/lif2bwave.dat',/get_lun

nn2 = 0L         ; first # is known to be long
readu,lun,nn2 ; read first 4 bytes of file into long(n)
nn2 = swap_endian(nn2)
print,"First file has ",nn," entries, second file has ",nn2," entries"

xx = fltarr(nn)
readu, lun, xx  ; read the rest of the file into yy
xx = swap_endian(xx)

free_lun,lun ; close file

plot,xx,yy
```

`read` in a script

- The `read` command accepts interactive input:
  - `read,input,prompt="Input number!"`
- Can be useful if you want a script that you can re-use with different files

Scripts know what's defined

- If you have a script that *uses* a variable, you can define that variable outside of the script:

- ```
  filename = 'spectrum.dat'
  @plot_spectrum ; this is a script that plots
                 ; the spectrum in the filename specified
  ```
  ```

Line Continuation Again

- This is about where it shows up in the book...
- Line continuation can be used interactively, in scripts, or in procedures
  - it’s most useful in scripts & procedures

Compound Statements

- Not covered in the text (at least, not that I've found yet...)
- You can do multiple things on one line:
  - `x = 5 & y=7 & z=x*y`
- The `&` character does this

Compound Statements

- Generally to be avoided
- It is almost always more clear to do things on multiple lines
- BUT, if you want to do a lot on the command prompt (and this is a bad idea in general but I do it sometimes), it can be useful

A warning about scripts

- Because scripts know what variables are defined, they may behave differently in a “fresh” IDL session
- *ALWAYS* test your scripts in a fresh session before turning them in!
  - `.reset_session` can be used for this

Readability

- The hardest concept to get across, but the most important
- Whoever reads your code should EASILY understand what it does!
- Scripts, procedures, and functions should NOT read like whuduzitdo's!!

Readability!!

- Comments are great
  - Inane comments are... not so great
    - Hilarious comments can preserve your sanity
      - but use sparingly, lest you find yourself writing a book!
- Comment OFTEN! But not always!

Which comment(s) are extraneous?

A

B

C

D

E

```
1  ; Draw an eyeball
2  ; Make a "t" array for parametric eqns
3  tt = findgen(1000)/999. * 2 * !pi
4  ; Build a circle:
5  ; x^2 + y^2 = r^2, so
6  ; x = r cos(t), y=r sin(t)
7  rr = 0.25 ; radius of circle
8  xx = cos(tt)*rr ; cosine(t) * r
9  yy = sin(tt)*rr ; sine(t) * r
10
11 xo1 = 1.5
12 yo1 = 2.0
13
14 xo2 = 2.5
15 yo2 = 2.0
16
17 plot,xx+xo1,yy+yo1,xrange=[1,3],yrange=[1,3]
18 oplot,xx+xo2,yy+yo2
```

Which lines need more explanation?

A

```
1 ; Draw an eyeball
2 ; Make a "t" array for parametric eqns
3 tt = findgen(1000)/999. * 2 * !pi
4 ; Build a circle:
5 ; x^2 + y^2 = r^2, so
6 ; x = r cos(t), y=r sin(t)
```

B

```
7 rr = 0.25 ; radius of circle
8 xx = cos(tt)*rr ; cosine(t) * r
9 yy = sin(tt)*rr ; sine(t) * r
```

C

```
10
11 xo1 = 1.5
12 yo1 = 2.0
13
14 xo2 = 2.5
15 yo2 = 2.0
16
```

D

```
17 plot,xx+xo1,yy+yo1,xrange=[1,3],yrange=[1,3]
18 oplot,xx+xo2,yy+yo2
```

E

Readability!!!

- When to comment:
  - Any time you use a mathematical formula

```
; The Planck function
B = 2*h * c^2 / lambda^5 * (exp(h*c/(lambda*k*t)) - 1)^(-1)
```

  - Any time it’s not 100% clear what code is going to do
  - Whenever you are trying to do something and you know how to say what you’re trying to do, but not how to do it

Readability!!!!

- Example:

```
44 ; Take the derivative of the planck function
45 ; I think that means I need dx, but I can't
46 ; remember what else right now
47 ; OH YEAH! I need to use the shift function!
```

Readability!!!!!

- Use good variable names
  - Good:
    `nbins ; short for number of bins`
  - Bad:
    `nb`
- Hard to show by examples - we'll try to feed back with comments on your code

Readability!!!!!!

- SPACING!
- Use spaces wisely!
  - Use blank lines to separate “conceptual” blocks within a script / procedure
  - Use indentation inside blocks of code [this will come later, but is *part of the syntax* in python!]

Comments cont'd

- You should comment whenever you use a formula you found somewhere else
- Even in this book: say "Formula from ch 5, page 56" or something like that
- it will be MUCH easier to figure out how you did things when you want to do them again!

Comments again!

- If you look at your old code, you might realize it’s not well enough commented
- That’s a great time to... add comments!!

Comments at last

- If you change your code, make sure you update the corresponding comments!
- out of date comments are awful and extra confusing!

Variable Names

- Don’t use single-letter variable names
- When you’re writing by hand, this can save a lot of time (& hand cramping)
- When you’re using a computer, it COSTS time: it is MUCH HARDER to search for single-letter variables than words

A last note on readability

- IDL does not, technically, have a fixed “convention” for how you should do all the readability things (naming, spacing, etc)
- python DOES: it’s called “PEP8” http://www.python.org/dev/peps/pep-0008/

Chapter 11

- What's the difference between a "script" and a "program"?
  - programs have the `end` keyword at the end
  - you call programs with
    `.r program.pro`
    instead of
    `@script.pro`
  - programs do not change interactive variables & do not know about them

“compiling” programs

- IDL doesn’t “compile” programs in the same way that, e.g., C or FORTRAN would
  - they create an “executable” file that can then be called as a unix command
- In IDL, “compile” means “make accessible as a command”, but only within IDL

Compile

- You compile programs with... wait for it... the `.compile` command
- IDL will check for syntax errors at this point

`.run`

- After you've compiled code, you can run it with the `.run` command
- `.run` will compile the code anyway, though, so `.compile` is not necessary

Programs vs Scripts

- `.run`ning or `.compile`ing checks for syntax error before the code runs
  - scripts try to run each line, without checking for errors
- Also, programs allow you to use flow control.

Programs > Scripts

- Flow Control is a BIG DEAL. There's a whole chapter on it, chapter 12. That chapter is long. And important.
- Most "fundamentals of programming" courses will start with flow control around chapter 2 - it really is fundamental

Programs & stahp

- You can include `stop` statements in your programs
- This allows you to halt the code mid-execution, do something (check the status of some variable, for example), then... `.continue` to continue
  - `.c` is short for `.continue`

`stop step continue`

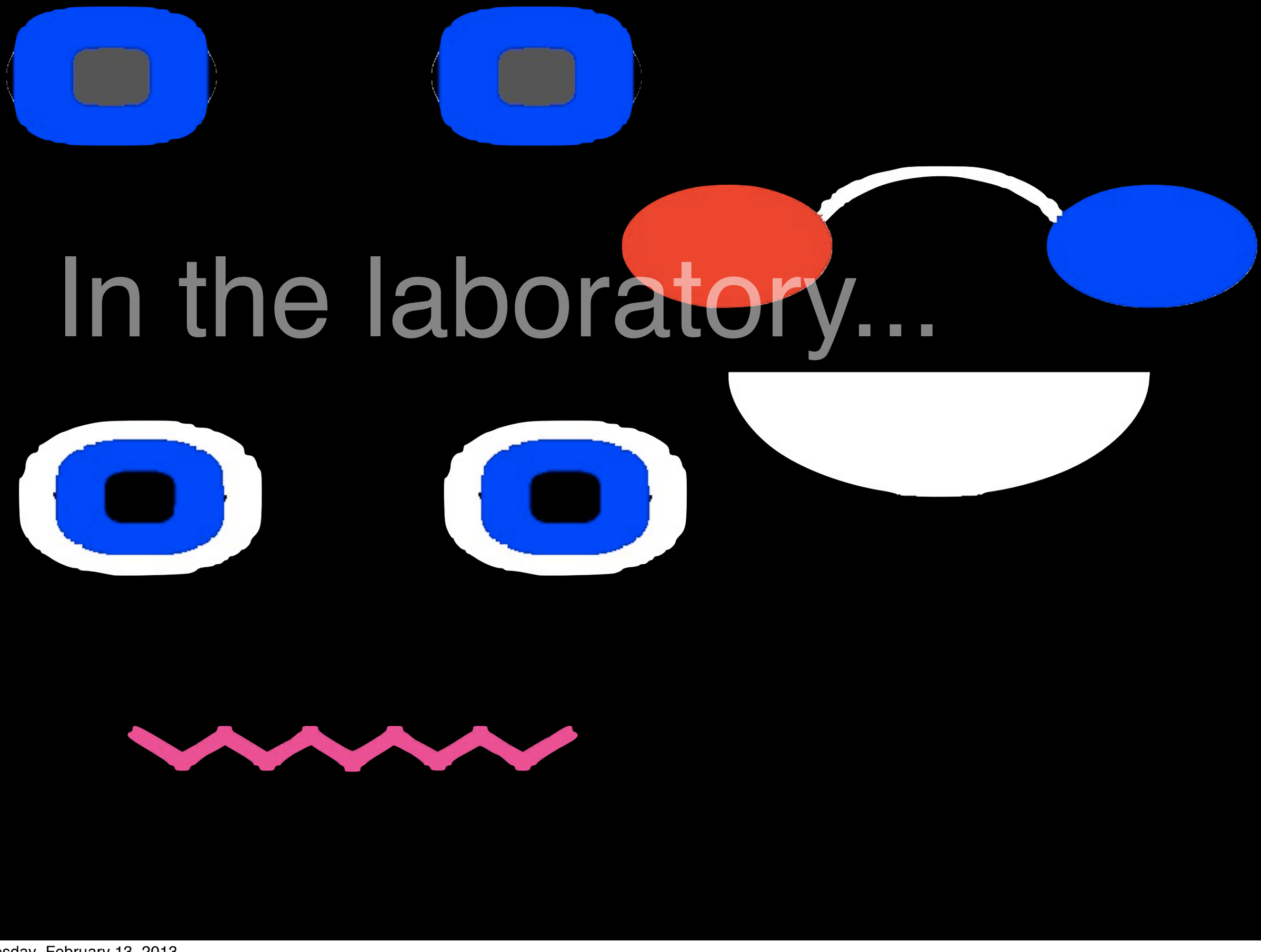
- You can use the `.step` command to step through code line-by-line
  - useful for debugging!
- you will automatically drop into "`stop`" mode whenever there's a crash in your code (if it's not a syntax error)

Python “scripts”

- In python, there’s not really a distinction between a “script” and a “program”
- Instead of `.r`, you use `%run`
  - e.g.: `%run myscript.py`
- python scripts don’t know about interactive variables by default
  - Can change this: `%run -i myscript.py` if you want it to know about variables

Python `import`

- Python has a concept IDL lacks: `importing`
- In IDL, you `.compile` a code, then have access to all functions defined in that program
- In python, you `import numpy`, then you can access its functions:
  - `numpy.linspace`
  - `numpy.sin`
- We'll revisit the concept of "namespaces"


In the laboratory...