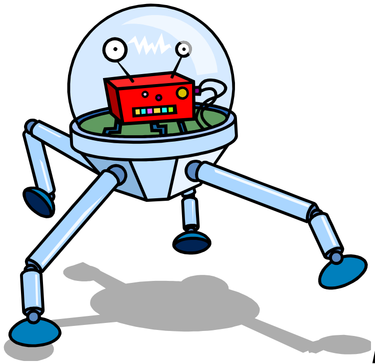
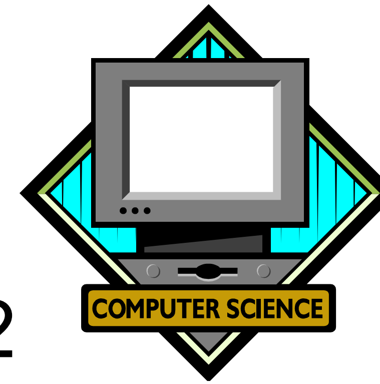
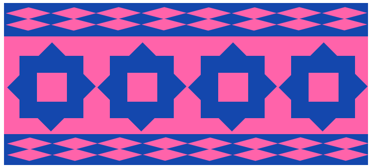
Caltech/LEAD Summer 2012 Computer Science

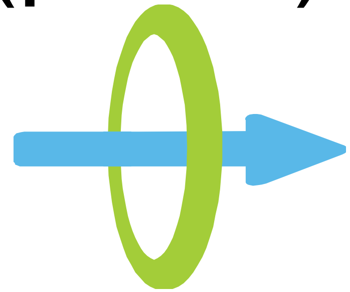




Lecture 11: July 18, 2012

Graphics and Graphical User Interfaces (part 1)





This lecture

- Graphics
- Graphical user interfaces (GUIs)
- Event handling and event loops

What does "graphics" mean?

- Graphics is (loosely speaking) the process by which you create visual images on a computer screen
- Graphics also involves the process of *interacting* with these visual images in a meaningful fashion

Text-based programming

- Much of the programming we've done so far has been *text-based*
- Examples: games that work from the terminal (Mastermind), programs that read and write text files, etc.
- Text-based programming has a simple notion of...

User interface

- A *user interface* refers to how you (the user) interact with a program
- Text-based programs tend to have very simple user interfaces
- Programs that create and use graphics can have much more complex user interfaces

Text-based user interfaces

- A text-based program typically has one of two kinds of user interface:
  - *batch mode*
  - *interactive mode*

Batch mode

- A *batch mode* program is run without any user involvement
  - Example: compute and print π to 1000 decimal places
  - The computer computes a while and then prints out `3.1415926...` and finally halts
  - Once the program has started, the program's user just waits for the answer

Interactive mode

- An *interactive mode* program is run with the help of the user
  - Example: Mastermind game
  - You enter a guess
  - The computer tells you how good it was
  - You enter another guess
  - etc. until you guess correctly
- You and the computer "take turns"

Graphical user interfaces

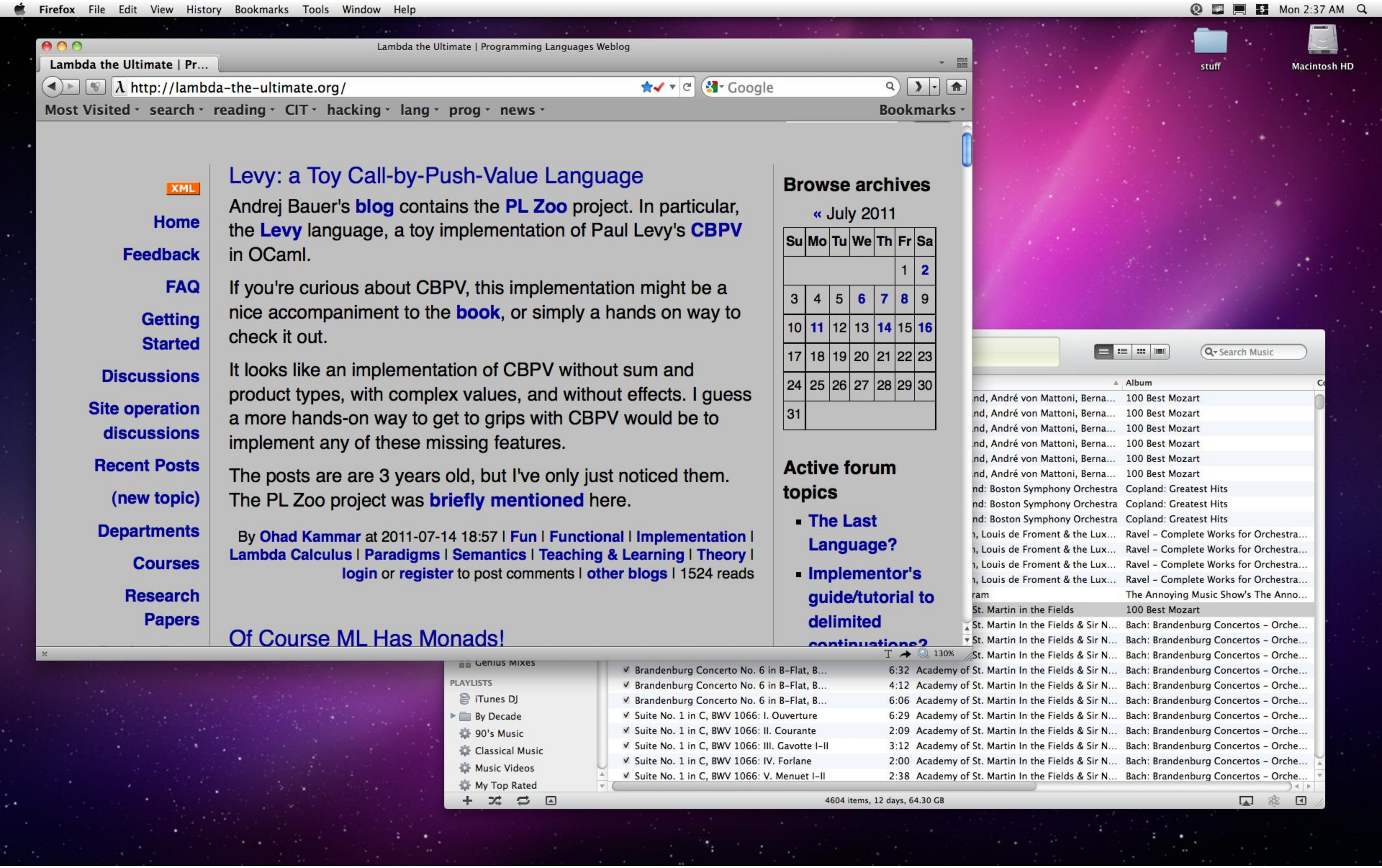
- Programs that do graphics usually don't fit into either of these categories
- Instead, they have a *graphical user interface* (GUI) which users interact with directly

Graphical user interfaces

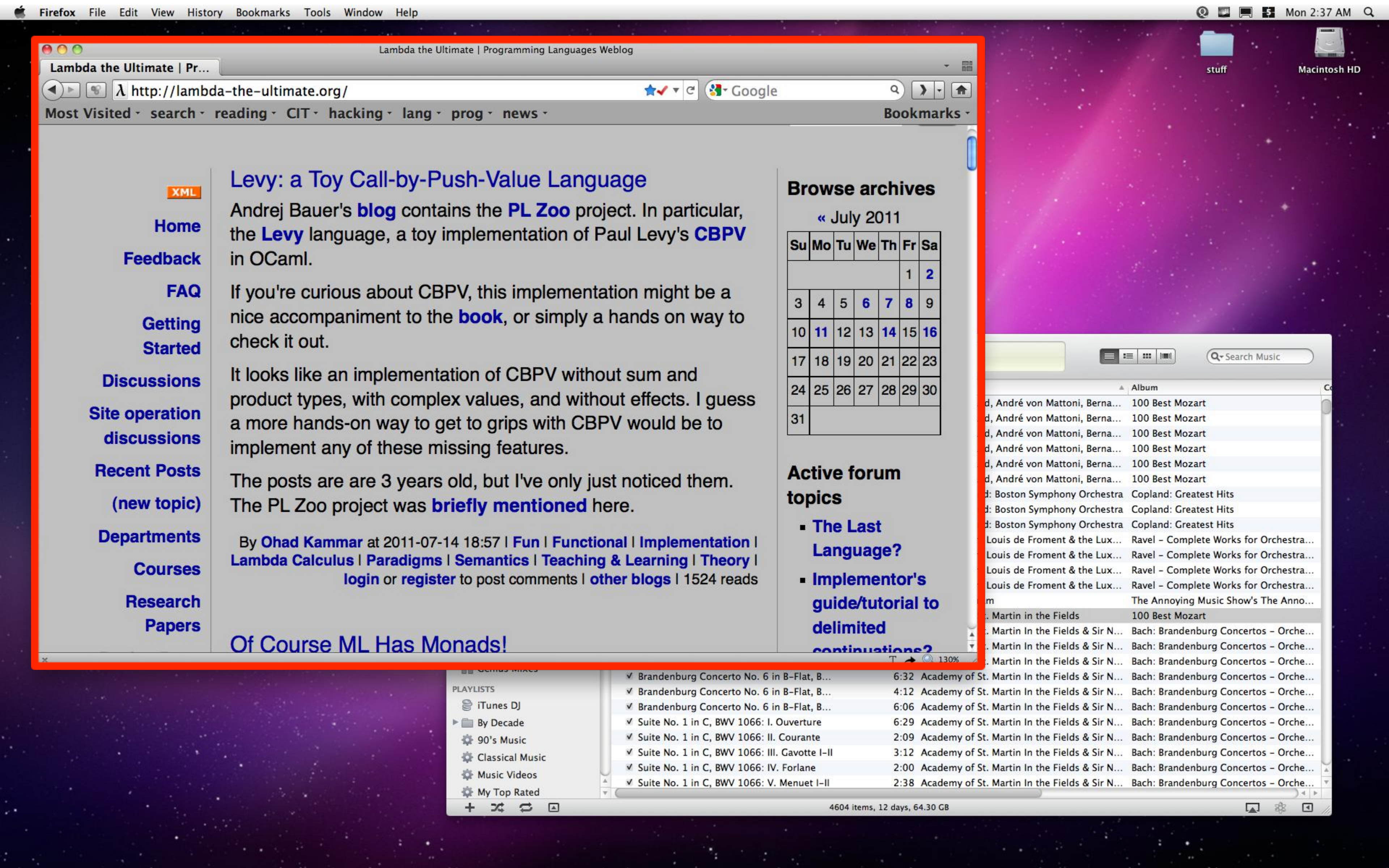
- The program provides various visual entities (called *widgets*) that you can interact with
  - buttons, menus, sliders, scrollbars, etc.
  - drawing surfaces for drawing
- The program also displays output visually
  - images, animations, etc.

Example

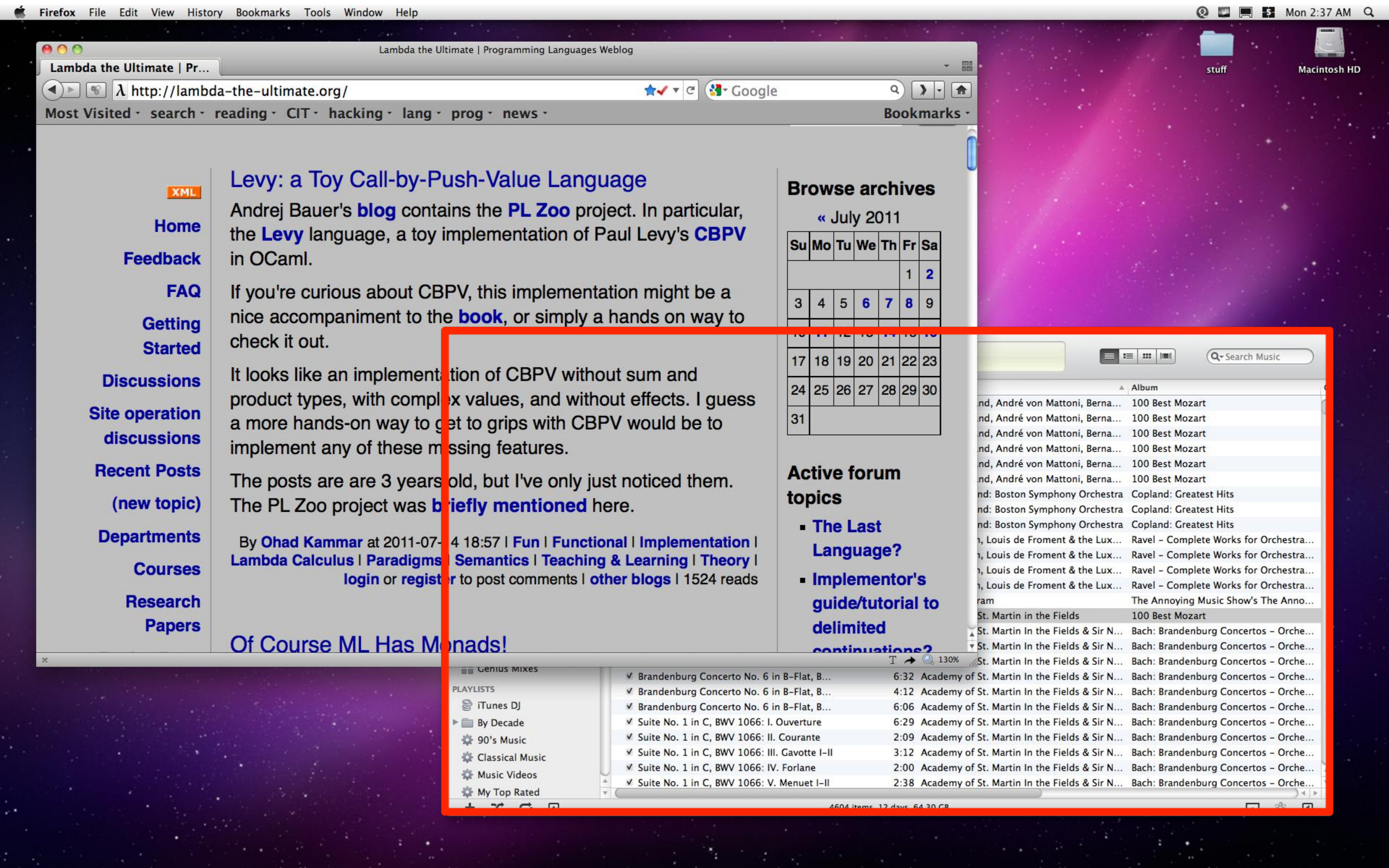
- The "desktop" of a computer running Mac OS X


Firefox File Edit View History Bookmarks Tools Window Help
Mon 2:37 AM
stuff
Macintosh HD
Lambda the Ultimate | Programming Languages Weblog
Lambda the Ultimate | Pr...
http://lambda-the-ultimate.org/
Google
Most Visited search reading CIT hacking lang prog news
Bookmarks
XML
Home
Feedback
FAQ
Getting Started
Discussions
Site operation discussions
Recent Posts
(new topic)
Departments
Courses
Research Papers
Levy: a Toy Call-by-Push-Value Language
Andrej Bauer's blog contains the PL Zoo project. In particular, the Levy language, a toy implementation of Paul Levy's CBPV in OCaml.
If you're curious about CBPV, this implementation might be a nice accompaniment to the book, or simply a hands on way to check it out.
It looks like an implementation of CBPV without sum and product types, with complex values, and without effects. I guess a more hands-on way to get to grips with CBPV would be to implement any of these missing features.
The posts are are 3 years old, but I've only just noticed them. The PL Zoo project was briefly mentioned here.
By Ohad Kammar at 2011-07-14 18:57 | Fun | Functional | Implementation | Lambda Calculus | Paradigms | Semantics | Teaching & Learning | Theory | login or register to post comments | other blogs | 1524 reads
Of Course ML Has Monads!
Browse archives
« July 2011
Su Mo Tu We Th Fr Sa
1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
Active forum topics
The Last Language?
Implementor's guide/tutorial to delimited continuations?
Search Music
Album
100 Best Mozart
Copland: Greatest Hits
Ravel - Complete Works for Orchestra...
The Annoying Music Show's The Anno...
Bach: Brandenburg Concertos - Orche...
Genius Mixes
PLAYLISTS
iTunes DJ
By Decade
90's Music
Classical Music
Music Videos
My Top Rated
Brandenburg Concerto No. 6 in B-Flat, B... 6:32 Academy of St. Martin In the Fields & Sir N...
Brandenburg Concerto No. 6 in B-Flat, B... 4:12 Academy of St. Martin In the Fields & Sir N...
Brandenburg Concerto No. 6 in B-Flat, B... 6:06 Academy of St. Martin In the Fields & Sir N...
Suite No. 1 in C, BWV 1066: I. Ouverture 6:29 Academy of St. Martin In the Fields & Sir N...
Suite No. 1 in C, BWV 1066: II. Courante 2:09 Academy of St. Martin In the Fields & Sir N...
Suite No. 1 in C, BWV 1066: III. Gavotte I-II 3:12 Academy of St. Martin In the Fields & Sir N...
Suite No. 1 in C, BWV 1066: IV. Forlane 2:00 Academy of St. Martin In the Fields & Sir N...
Suite No. 1 in C, BWV 1066: V. Menuet I-II 2:38 Academy of St. Martin In the Fields & Sir N...
4604 items, 12 days, 64.30 GB

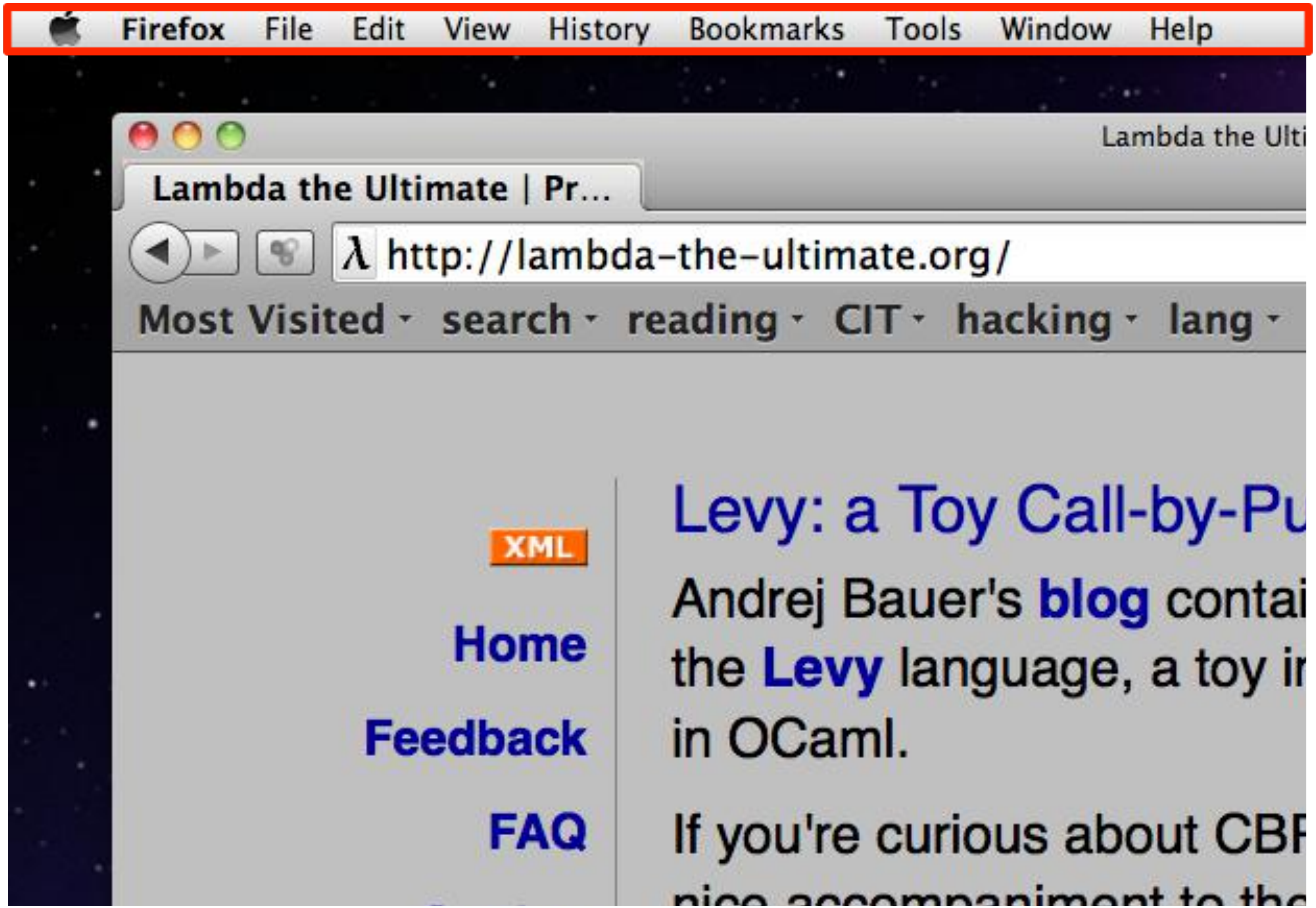
Application (program)



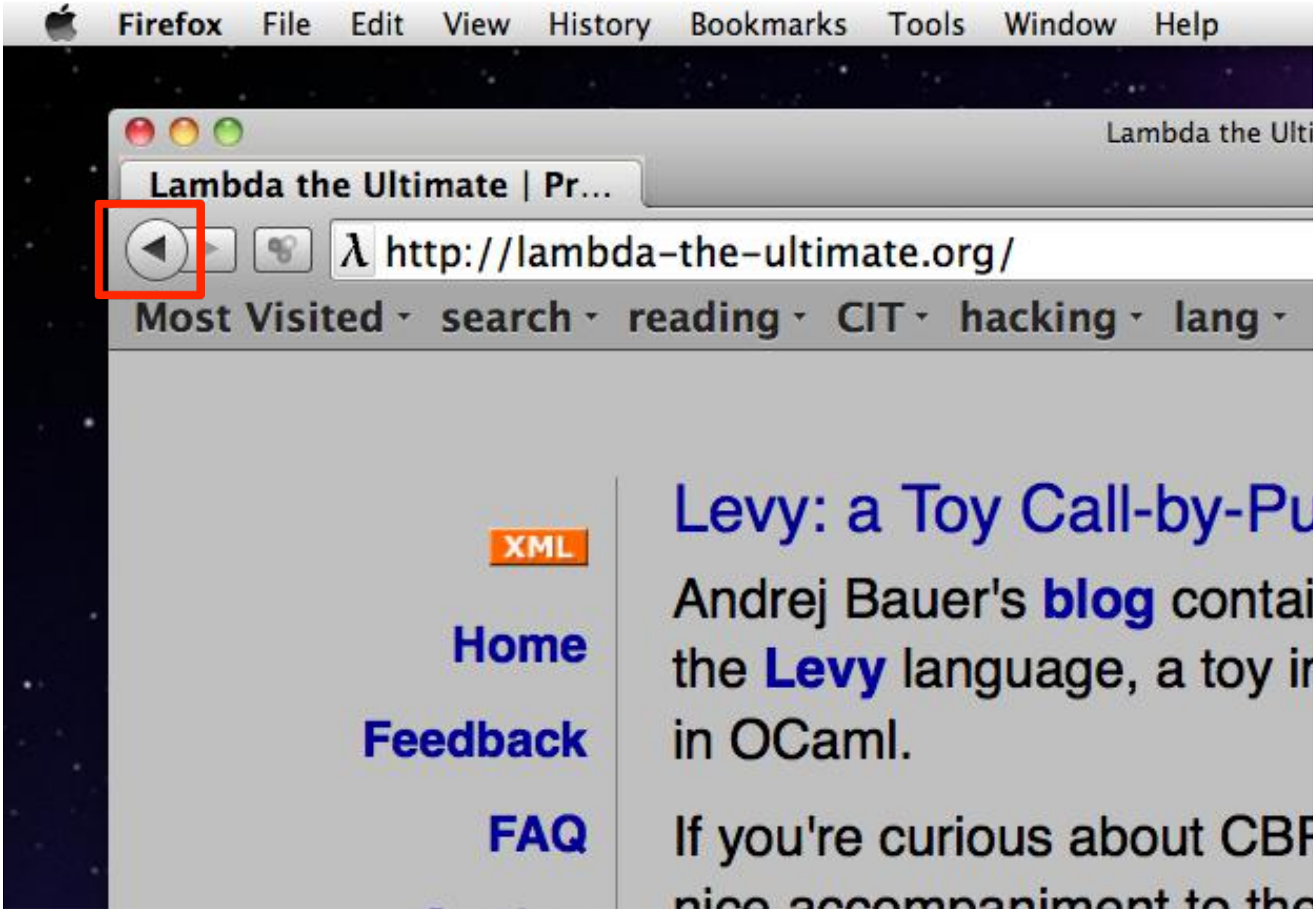
Application (program)



Menu


Firefox File Edit View History Bookmarks Tools Window Help
Lambda the Ulti
Lambda the Ultimate | Pr...
http://lambda-the-ultimate.org/
Most Visited
search
reading
CIT
hacking
lang
XML
Home
Feedback
FAQ
Levy: a Toy Call-by-Pu
Andrej Bauer's blog contai
the Levy language, a toy in
in OCaml.
If you're curious about CBP

Button


Firefox File Edit View History Bookmarks Tools Window Help
Lambda the Ulti
Lambda the Ultimate | Pr...
http://lambda-the-ultimate.org/
Most Visited search reading CIT hacking lang
XML
Home
Feedback
FAQ
Levy: a Toy Call-by-Pu
Andrej Bauer's blog contai
the Levy language, a toy i
in OCaml.
If you're curious about CBP

Graphical interfaces

- Graphical interfaces are so common today, we don't even notice them
  - unless they go wrong ;-)
- But their interfaces are very complex
- They are designed to make it feel "natural" for you to interact with the program

Event loop

- Typically, program will wait for the user (you) to activate one of its widgets
  - push a button, select a menu item, draw on a drawing surface etc.
- Then it will do something in response
- Then it will go back to waiting for you to do something again
- This process is called an *event loop*
  - your actions are the "events" the program is waiting for

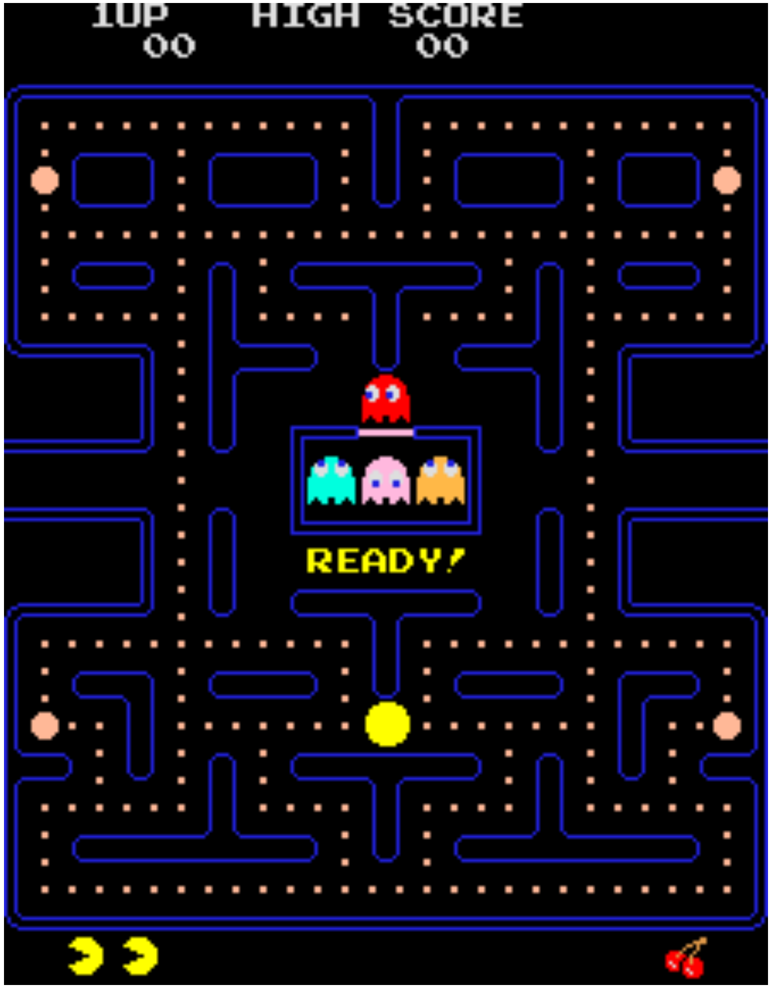
Programming GUIs

- Programming graphical user interfaces (GUIs) is somewhat laborious
  - (some people find it boring)
- The programmer must anticipate every reasonable thing the user might want to do with the program
  - then provide a graphical object (widget) to allow that to happen
- Good news: graphics programming is about more than this!

2-D and 3-D graphics

- Aside from graphical user interfaces, there are two other broad categories of graphics programming:
- 2-D (two-dimensional) graphics: drawing pictures (or animations) on a two-dimensional surface
- 3-D (three-dimensional) graphics: drawing pictures (or animations) to resemble three-dimensional objects

2-D graphics example



3-D graphics example

Today

- We will only be dealing with 2D graphics today
  - With some user interface thrown in for good measure
- We'll continue and expand our examples in later lectures

Python and graphics

- Many kinds of graphics libraries are available in Python
  - 2-D, 3-D, GUI, etc. (many of each)
- However, none are built-in
  - all require that you add the libraries to the basic Python installation
- Today, we'll look at the most commonly-used graphics library in Python

Turtle graphics

- Many of the labs use a graphics system called "turtle graphics"
- This is a simplified graphics system used mainly for teaching programming concepts
- It's also lots of fun!
- Turtle graphics are described in the writeups for labs 3-5
  - (not covered here)

Turtle graphics

- Internally, the Python turtle graphics module uses the same graphics library (`Tkinter`) we're going to describe now
- However, `Tkinter` can do much more, including GUIs and arbitrary drawing
- This will be enough graphics capability for the rest of this course

Tkinter

- The most common graphics library in Python is called **`Tkinter`**
  - perhaps because all the sensible, meaningful, pronounceable names were taken?
- It's a Python **`inter`**face to a system called "**`Tk`**" (which stands for "graphics **`T`**ool**`k`**it") written in a different language

Tkinter

- Tkinter provides a number of tools for writing programs that use graphics:
  - many GUI widgets
    - buttons, menus, labels, scrollbars etc.
  - a canvas widget on which arbitrary drawings can be created
    - using lines, circles, rectangles, ovals, images, text, etc.
  - ways to capture user interaction
    - key presses, mouse clicks, etc.

Tkinter

- We will concentrate on the canvas widget and drawing simple 2-D pictures
- We'll also show how to get a program to respond to actions (key presses) on the canvas

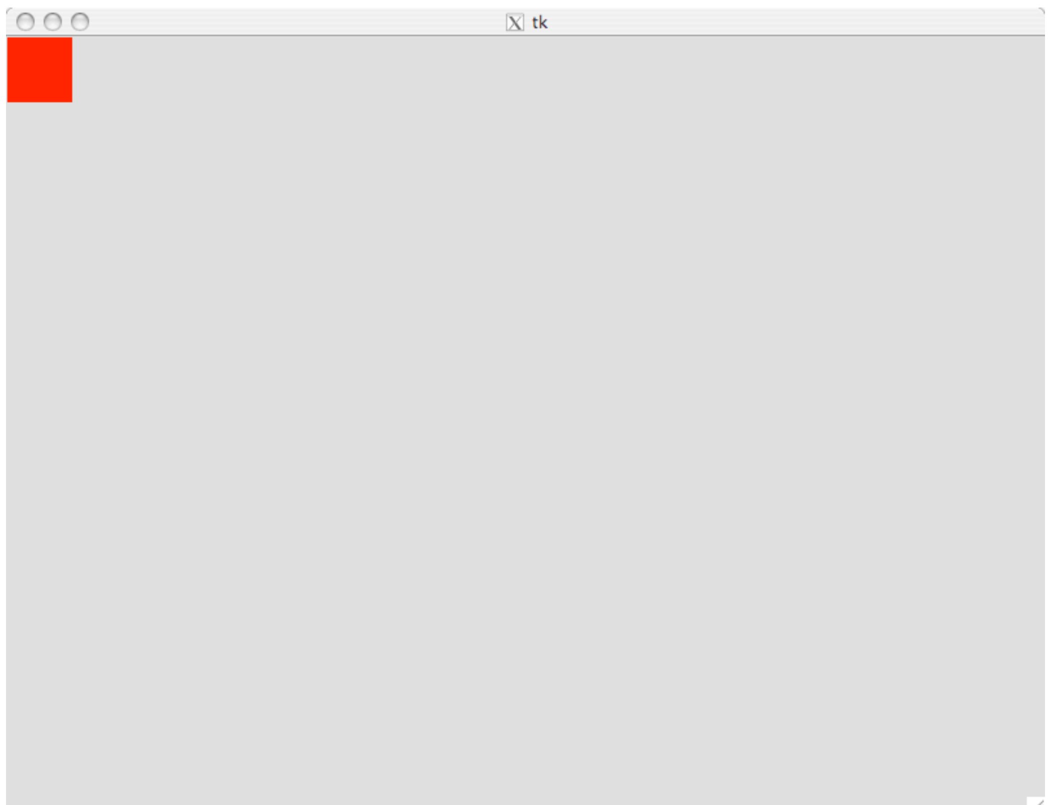
Simple Tkinter program

- In file **tkinter1.py**:

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
r = c.create_rectangle(0, 0, 50, 50, \
            fill='red', outline='red')
raw_input("Press <return> to quit")
```

Simple `Tkinter` program

- This program brings up a window with a red rectangle drawn in one corner:



Simple `Tkinter` program

- To understand how this works, we first have to understand
  1. pixel coordinates
  2. windows
  3. Python keyword arguments

Pixel coordinates

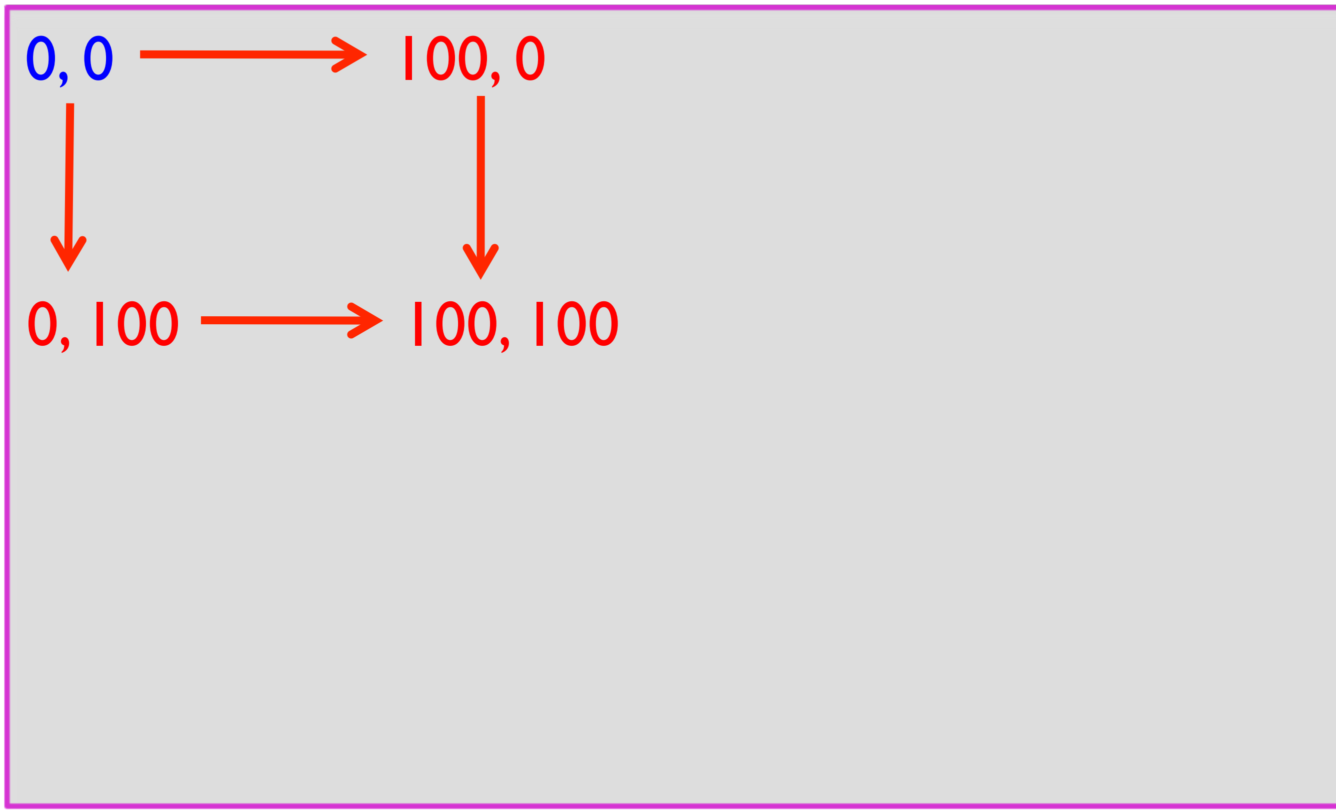
- To the computer, the entire screen is a 2-dimensional grid of tiny colored boxes called *pixels*
- Most computer screens have large numbers of pixels
  - *e.g.* 1440x900 pixels on this computer
  - or about 1.3 million pixels
- With millions of possible colors per pixel

Pixel coordinates

- The pixel in the upper left-hand corner is pixel (0, 0) (called the *origin*)
- The first pixel in the pair represents the horizontal dimension
  - so (100, 0) would be to the right of (0, 0)
- The second pixel in the pair represents the vertical dimension
  - so (0, 100) would be below (0, 0)

Pixel coordinates

- Visually:



Windows and pixels

- Most computers run multiple programs at one time, each in a separate *window*
- Pixel coordinates can be
  - absolute
  - relative to a particular window
- Absolute coordinates means the upper left-hand corner of the *monitor* is (0, 0)
- Relative means the upper left-hand corner of a *particular window* is (0, 0)
- Almost always use relative coordinates

Keyword arguments

- Last time, talked about dictionaries
  - store key/value pairs in a single data structure
  - keys are usually strings
- Python also allows functions to get key/value pairs as arguments to functions
  - as long as the key is a string
- All the key/value pairs are put into a dictionary before the function sees them

Keyword arguments

```
# Keyword argument example:
def foo(x, y, **kw):
    print x, y
    print kw

>>> foo(1, 2, width=100, height=200)
1 2
{ 'width' : 100, 'height' : 200 }
```

Keyword arguments

- In the definition of `foo`:

```
def foo(x, y, **kw):
```

- the `**kw` means that all the keyword arguments will be put into a dictionary called `kw`
- the `**kw` has to come at the *end* of the argument list
  - for boring technical reasons

Keyword arguments

- When calling the function `foo`:

```
foo(1, 2, width=100, height=200)
```

- the keyword arguments are `width` and `height`
- Inside the function `foo`, they get put into the `kw` dictionary, which becomes:

```
{ width: 100, height: 200 }
```

Keyword arguments

- Keyword arguments are useful in functions where you want to be able to specify arguments by name
- `Tkinter` uses keyword arguments a lot
- Usually the meaning is intuitive
  - *e.g.* `height` means height in pixels, `width` means width in pixels

Back to the example

- We had these lines:

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
```

- Let's see what they mean...

Back to the example

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
```

- This line means: import all names from the `Tkinter` module
- Use `from Tkinter import *` form because writing `Tkinter.<name>` for every name would be very tedious to write and to read

Back to the example

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
```

- This line means: create the *root window* of the program
- This is the window in which all the other graphical components of the application will be placed
- It is a Python object, so has methods

Back to the example

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
```

- This line calls the `geometry` method on the root object
- This line means: set the size of the root window to be **800** pixels wide (horizontal dimension) by **600** pixels deep (vertical dimension)

Back to the example

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
```

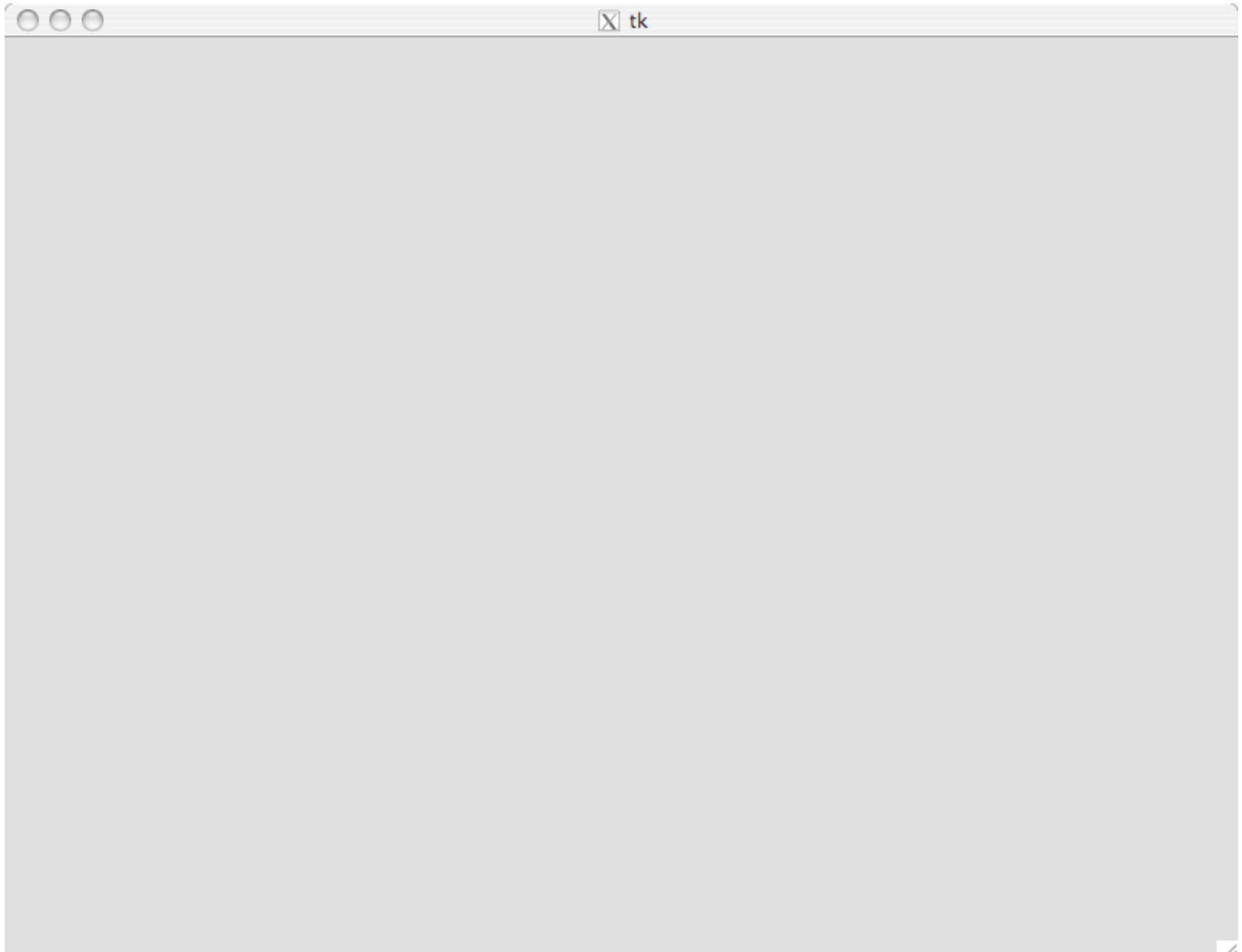
- You could run this as a whole program
- If you did, a blank window of size 800 by 600 would appear on the screen and then go away almost immediately
- Need a way to make the screen stay up!

Back to the example

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
raw_input('Press <return> to quit.')
```

- If this is the whole program, you get a blank window of size 800x600 which stays up until you press the return key in the terminal window

Result of the simple example



Result of the simple example

- Much as we love blank windows, we want to do more than this!
- The root window is basically just a container in which we can put other things
- We will put a drawing surface called a canvas inside it
  - canvas: analogy to painter's canvas

Adding a canvas

- Let's add two new lines to the example:

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
raw_input('Press <return> to quit.')
```

Adding a canvas

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
```

- This creates a new canvas object called `c`
- Its *parent* is the `root` object
  - it will be located entirely inside that object on screen
- Its dimensions will be 800x600 pixels
  - note keyword arguments: `width`, `height`

Adding a canvas

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
```

- A canvas is a Python object too, so it has methods
- The `pack` method positions the canvas inside its parent (the `root` object)
- Since they are both the same size, the canvas completely covers the `root` object

Adding a canvas

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
```

- Without this line, the canvas will never show up on the screen!
  - So don't leave it out!

Drawing

- Now we've created
  - the root window
  - the canvas
- It's time to do some actual drawing on the canvas

Drawing

```
# ... as before ...
c = Canvas(root, width=800, height=600)
c.pack()
r = c.create_rectangle(0, 0, 50, 50, \
          fill='red', outline='red')
```

- This creates a rectangle `r` on the canvas `c`
- `create_rectangle` is a method of the canvas object `c`

Drawing

```
# ... as before ...
c = Canvas(root, width=800, height=600)
c.pack()
r = c.create_rectangle(0, 0, 50, 50, \
          fill='red', outline='red')
```

- The first four arguments: `0`, `0`, `50`, `50` mean:
  - rectangle's upper left-hand corner is at location (0, 0)
  - rectangle's lower right-hand corner is at location (50, 50)
  - so it's actually a square of dimensions 50x50 pixels

Drawing

```
# ... as before ...
c = Canvas(root, width=800, height=600)
c.pack()
r = c.create_rectangle(0, 0, 50, 50, \
              fill='red', outline='red')
```

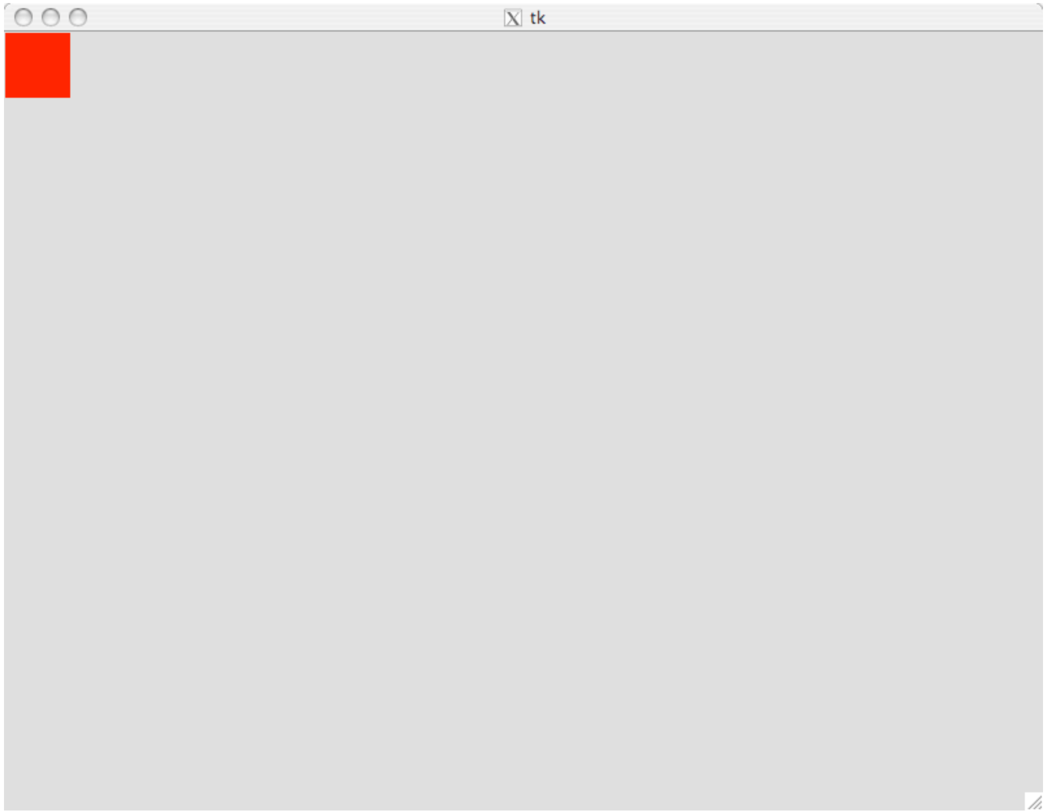
- The `fill` is the color inside the square
  - set it to be `'red'` because we like red!
- The outline is the color of the edges of the square
  - set to be `'red'` to make the entire square red

Drawing

```
# ... as before ...
r = c.create_rectangle(0, 0, 50, 50, \
          fill='red', outline='red')
raw_input('Press <return> to quit.')
```

- The **`raw_input`** line again makes the image visible until we hit the return key on the terminal
- This is a *very* crude way of interacting with a graphical program!
  - We'll see better ways soon

Result



- This is boring
- Let's add more stuff!

Drawing more

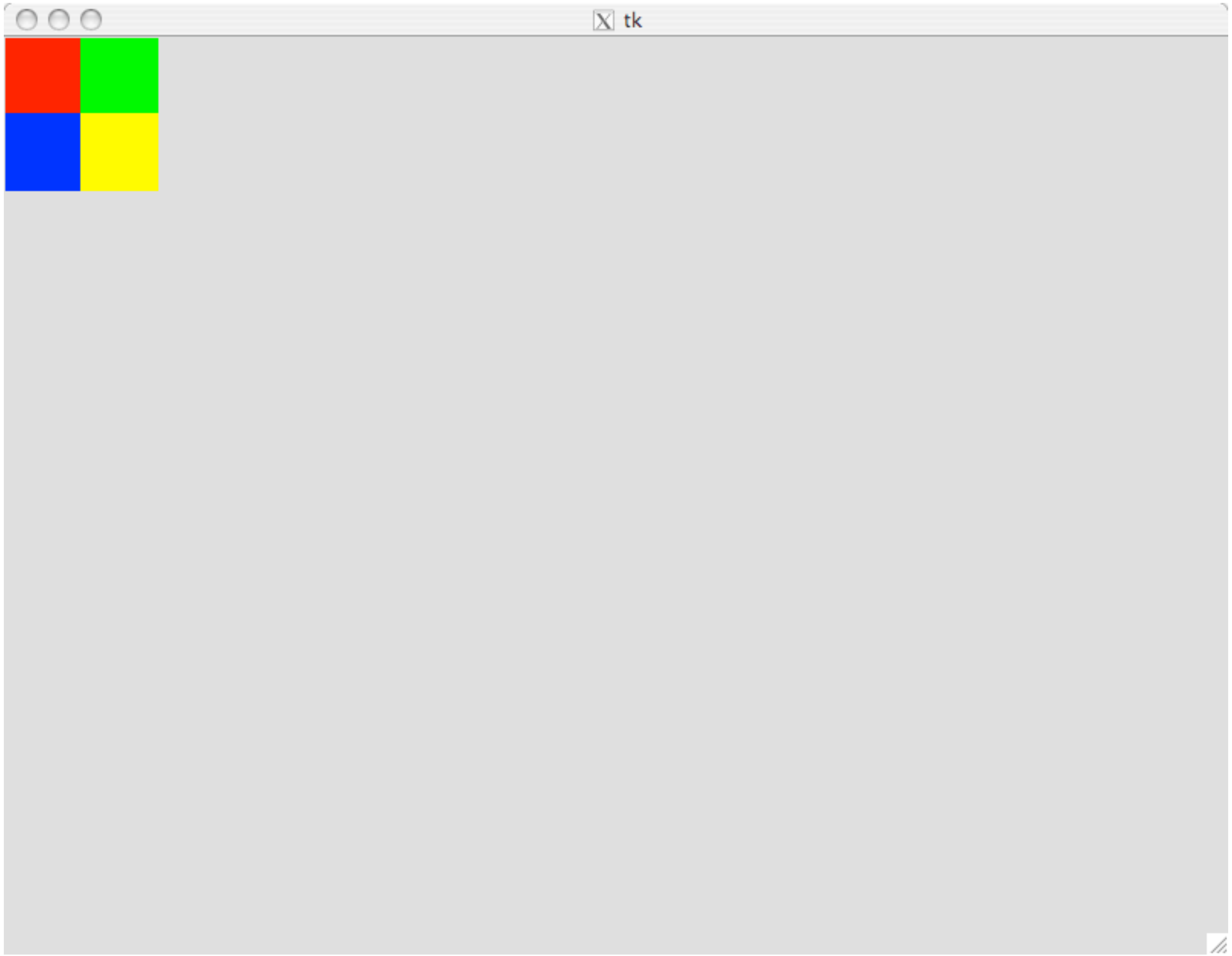
```
# ... as before ...
r = c.create_rectangle(0, 0, 50, 50, \
          fill='red', outline='red')
# ... add extra lines here ...
raw_input('Press <return> to quit.')
```

- We'll put extra lines between the `create_rectangle` line and the `raw_input` line
  - won't re-type those lines to save space on slides

Drawing more

```
# ... previous stuff ...
r2 = c.create_rectangle(0, 50, 50, 100, \
  fill='blue', outline='blue')
r3 = c.create_rectangle(50, 0, 100, 50, \
   fill='green', outline='green')
r4 = c.create_rectangle(50, 50, 100, 100,\
   fill='yellow', outline='yellow')
raw_input('Press <return> to quit.')
```

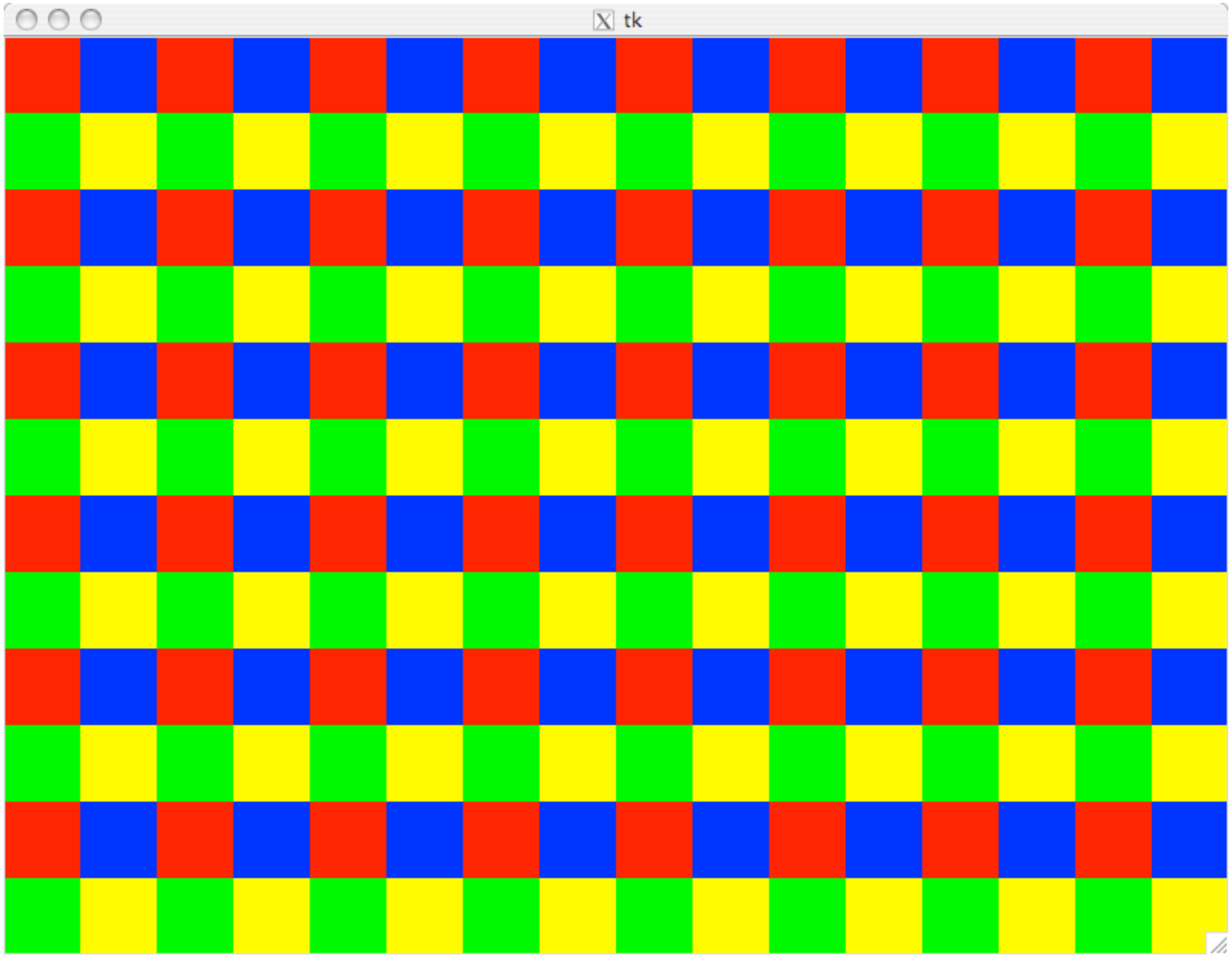
Result



Drawing more

- With only a few simple additions, we can generate more complicated images...

Drawing more



Moving on

- Now we know how to draw colored squares
- We will now create some different graphical objects

Starting point

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
```

- This part of the code will stay the same

Lines

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
line1 = c.create_line(0, 0, 800, 600,
            fill='blue', width=3)
line2 = c.create_line(800, 0, 0, 600,
            fill='red', width=6)
raw_input('Press <return> to quit')
```

create_line

- `create_line` is a method on canvas objects
- It creates one or more connected lines with particular properties
- Arguments:

```
create_line(x1, y1, x2, y2, ...,
  option1=value1, ...)
```

create_line

```
create_line(x1, y1, x2, y2, ...,
  option1=value1, ...)
```

- x1 and y1 are the initial point (where the line begins)

create_line

```
create_line(x1, y1, x2, y2, ...,
  option1=value1, ...)
```

- `x2` and `y2` are the next point
- A line is drawn on the canvas between coordinates `(x1, y1)` and coordinates `(x2, y2)`

create_line

`create_line(x1, y1, x2, y2, x3, y3, ..., option1=value1, ...)`

- There may or may not be more points
- Here, `x3` and `y3` are the next point
- A line is drawn on the canvas between coordinates `(x2, y2)` and coordinates `(x3, y3)`
- A single `create_line` method call can create a series of connected lines

Python notes

- `create_line` is an unusual function
- It can take an *arbitrary* number of arguments
  - the `x1`, `y1`, `x2`, `y2`, `x3`, `y3` etc. arguments
  - (We haven't seen how to do this yet!)
  - Must have at least `x1`, `y1`, `x2`, `y2` arguments (this makes one line)
  - If any more `x`, `y` pairs, they define the endpoints of subsequent lines

Python notes

- `create_line` can also take an arbitrary number of *keyword arguments*
  - the `option=value` arguments

- Examples:
  - `fill='blue'` (color of the line)
  - `width=3` (width of the line in pixels)

Back to the example

```
line1 = c.create_line(0, 0, 800, 600,
          fill='blue', width=3)
```

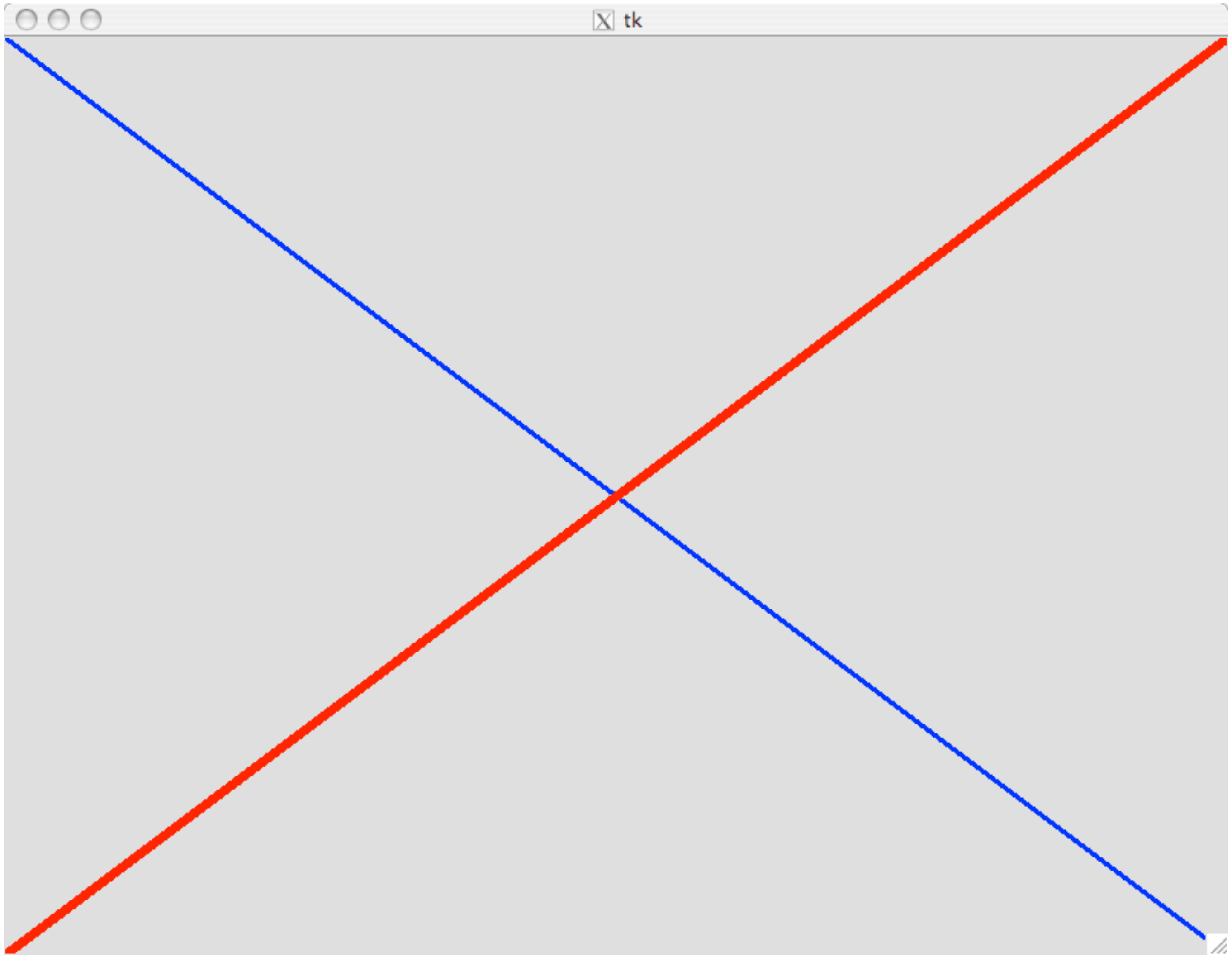
- This means:
  - create a line between coordinates `(0, 0)` and `(800, 600)`
  - (the upper-left corner and the lower-right corner)
  - This line should be `blue`
  - The width of the line should be `3` pixels
  - The resulting line should be named `line1`

Back to the example

```
line2 = c.create_line(800, 0, 0, 600,
          fill='red', width=6)
```

- This means:
  - create a line between coordinates `(800, 0)` and `(0, 600)`
  - (the upper-right corner and the lower-left corner)
  - This line should be `red`
  - The width of the line should be `6` pixels
  - The resulting line should be named `line2`

Result



Drawing more lines

- Let's try different line drawing commands:

```
line1 = c.create_line(100, 100, 400, 100,
                      100, 400, 400, 400,
                      fill='blue',
                      width=3)
```

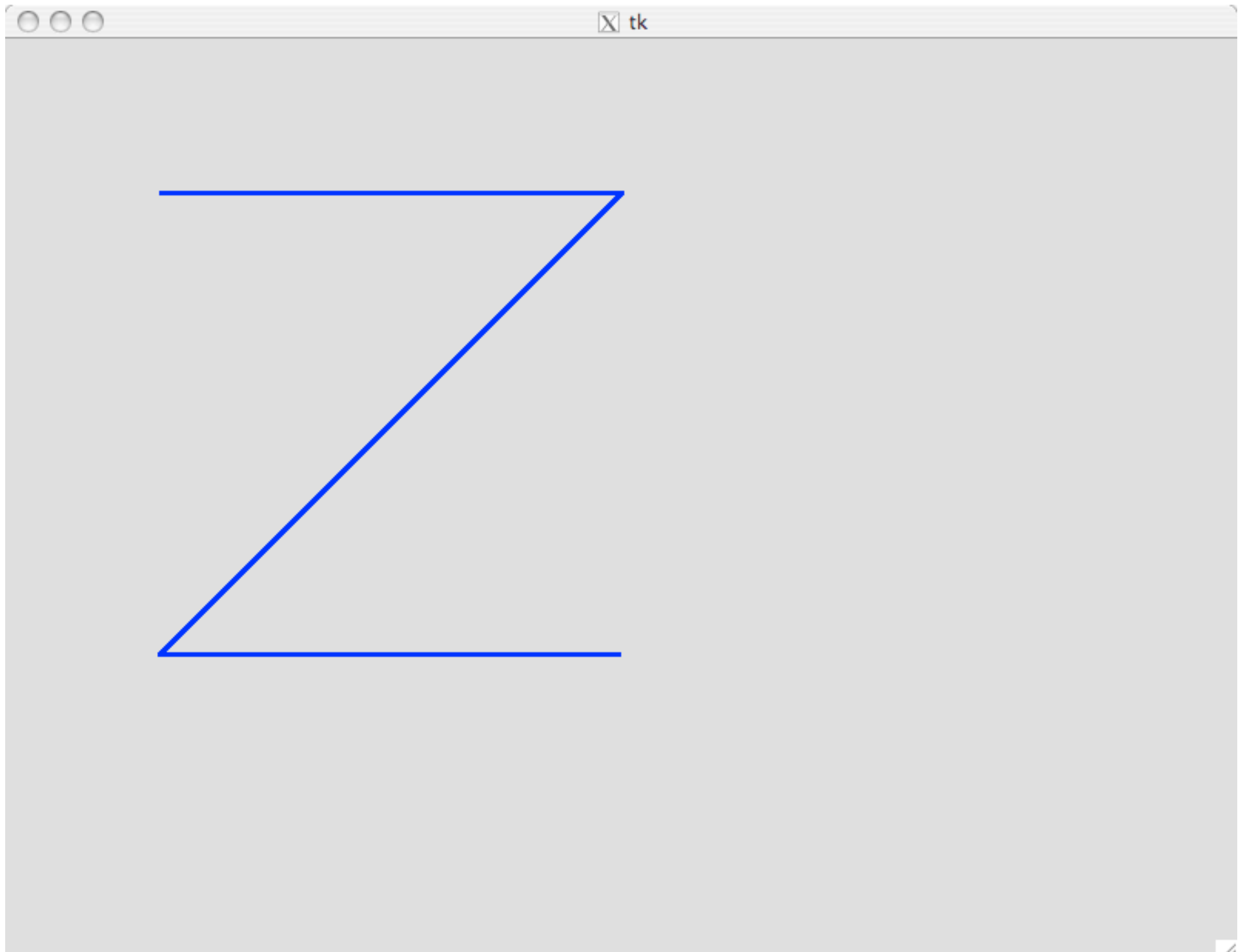
- This command creates 3 lines joined end-to-end
- All colored blue with a width of 3 pixels

Drawing more lines

```
line1 = c.create_line(100, 100, 400, 100,
                      100, 400, 400, 400,
                      fill='blue',
                      width=3)
```

- First line: `(100, 100)` to `(400, 100)` (horizontal)
- Second line: `(400, 100)` to `(100, 400)` (diagonal)
- Third line: `(100, 400)` to `(400, 400)` (horizontal)
- Gives a zig-zag 'Z' pattern

Result



Beyond lines

- Lines are only one of many things we can draw on Tkinter canvases
  - (Saw rectangular boxes earlier)
- Many other things can be drawn on canvases
  - polygons, arc, text, etc.
- Commands are similar to what we've already seen
- For fun, we'll look at one more example: ovals

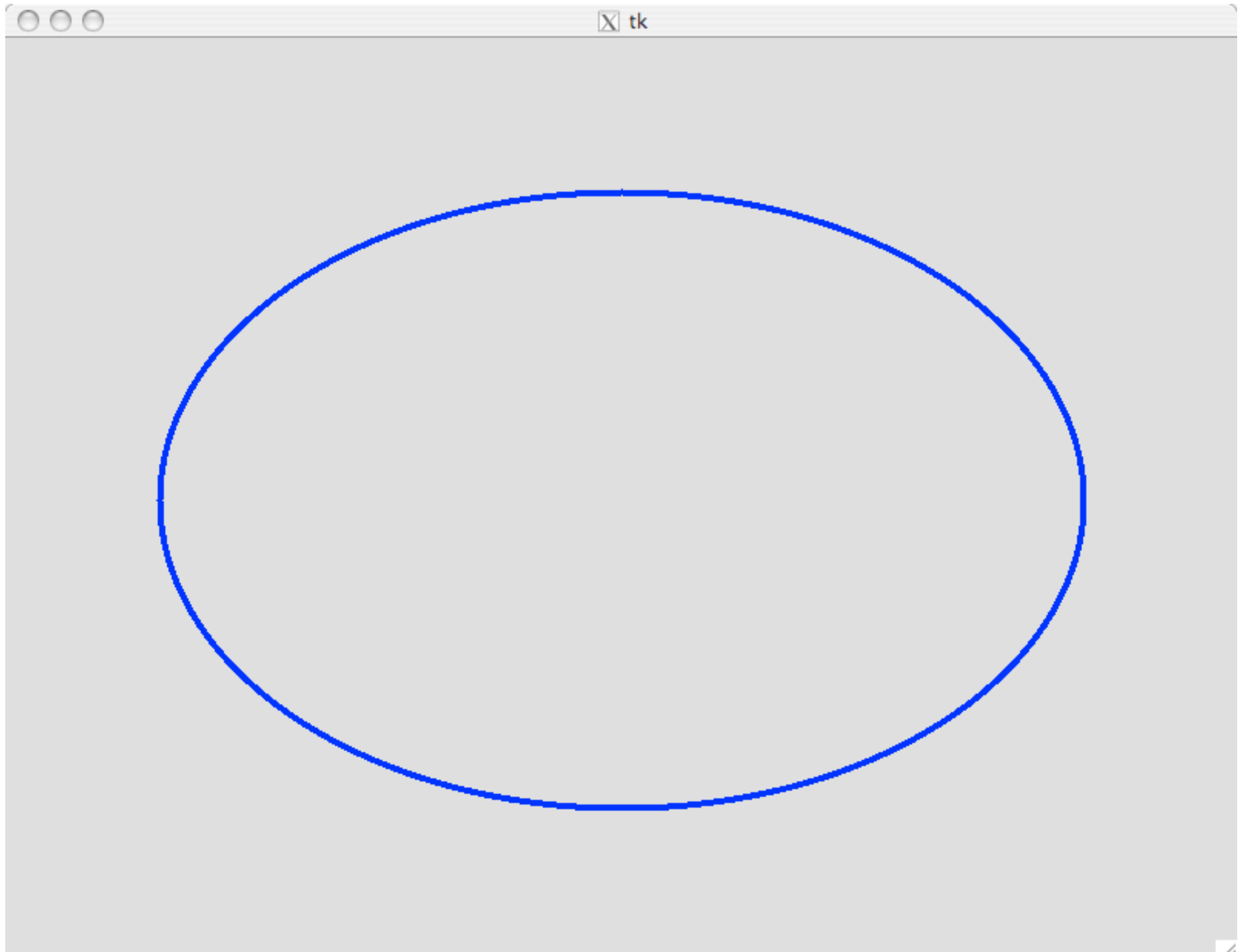
Ovals

- An oval is an elliptical shape which fits neatly inside a rectangle
  - edges touch the outer edges of the rectangle
- If the rectangle is a square, the corresponding oval is a circle
- Commands to draw ovals in `Tkinter` are very similar to rectangle-drawing commands

Oval example 1

```
oval = c.create_oval(100, 100, 700, 500,
                     outline='blue',
                     width=4)
```

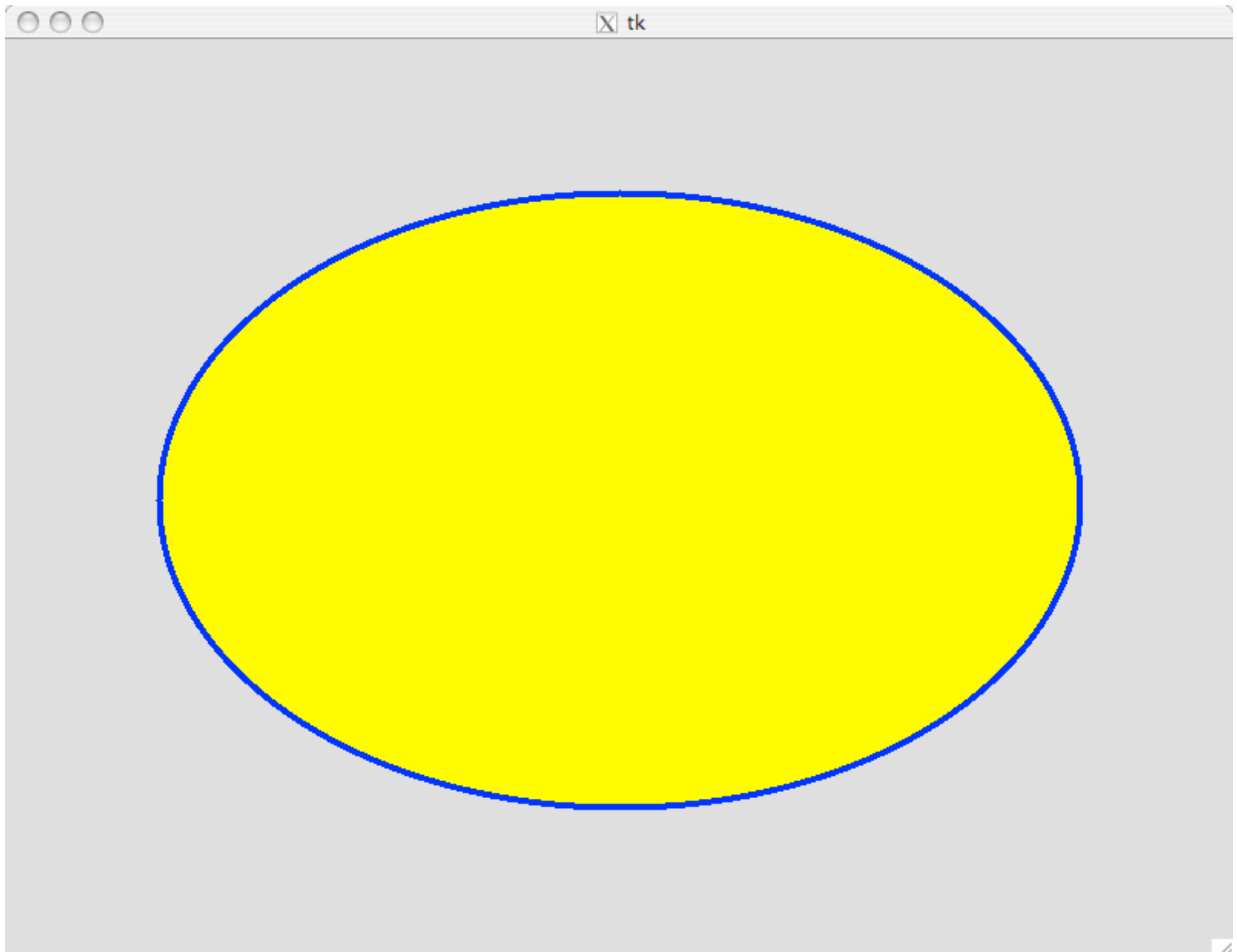
Oval example 1



Oval example 2

```
oval = c.create_oval(100, 100, 700, 500,
                     outline='blue',
                     fill='yellow',
                     width=4)
```

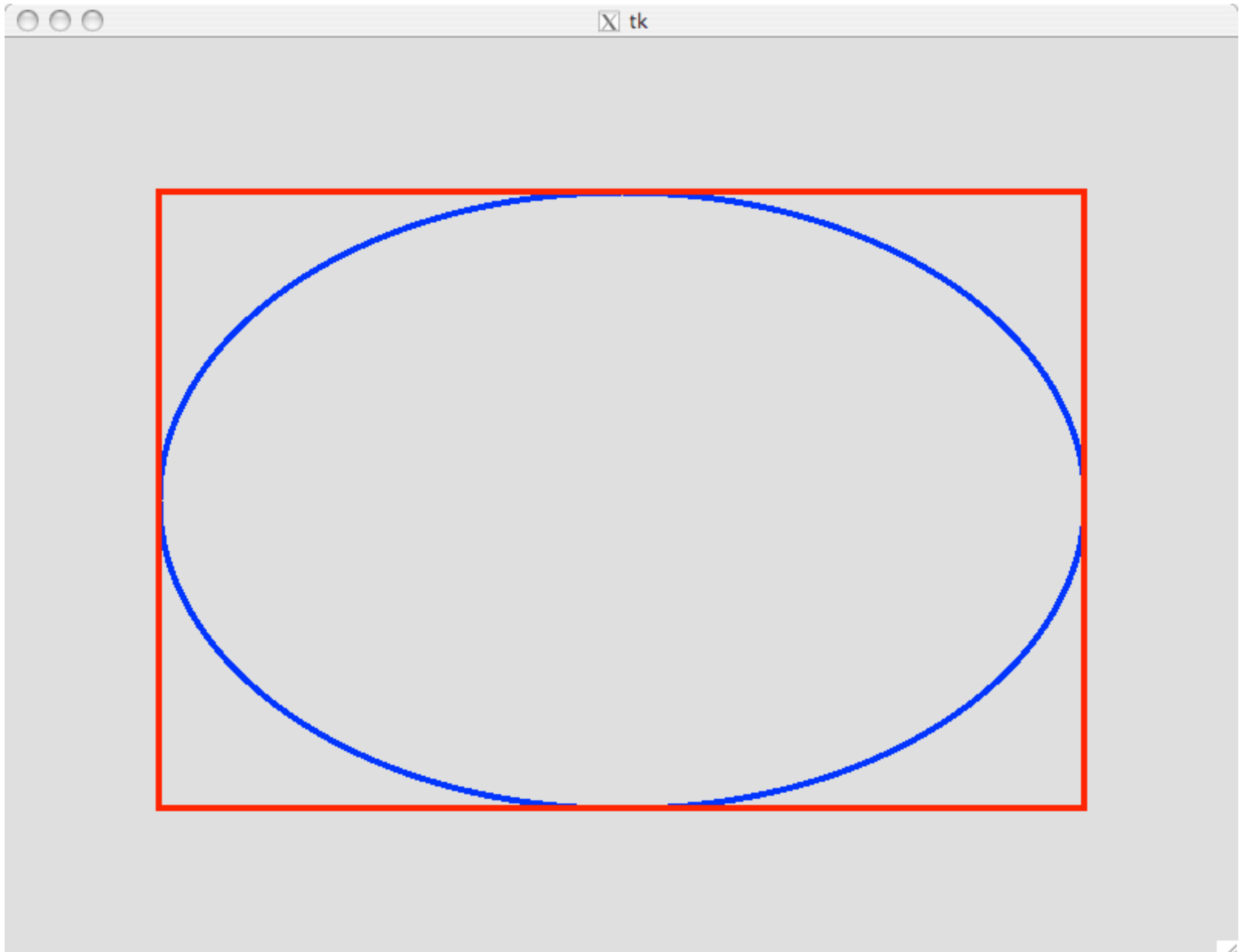
Oval example 2



Oval example 3

```
oval = c.create_oval(100, 100, 700, 500,
                     outline='blue',
                     width=4)
rect = c.create_rectangle(100, 100,
                          700, 500,
                          outline='red',
                          width=4)
```

Oval example 3

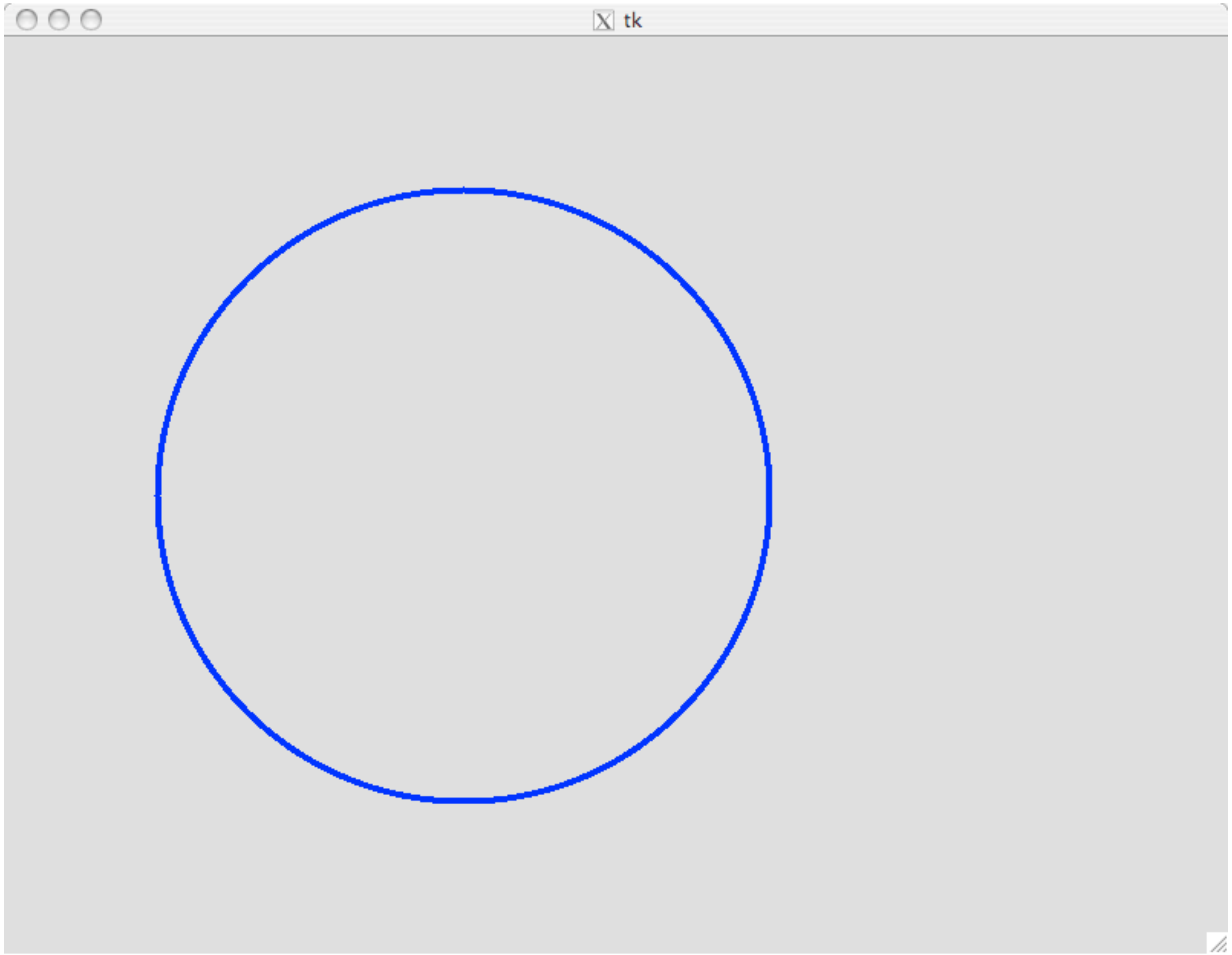


Oval example 4

```
oval = c.create_oval(100, 100, 500, 500,
                     outline='blue',
                     width=4)
```

- (This is a circle)

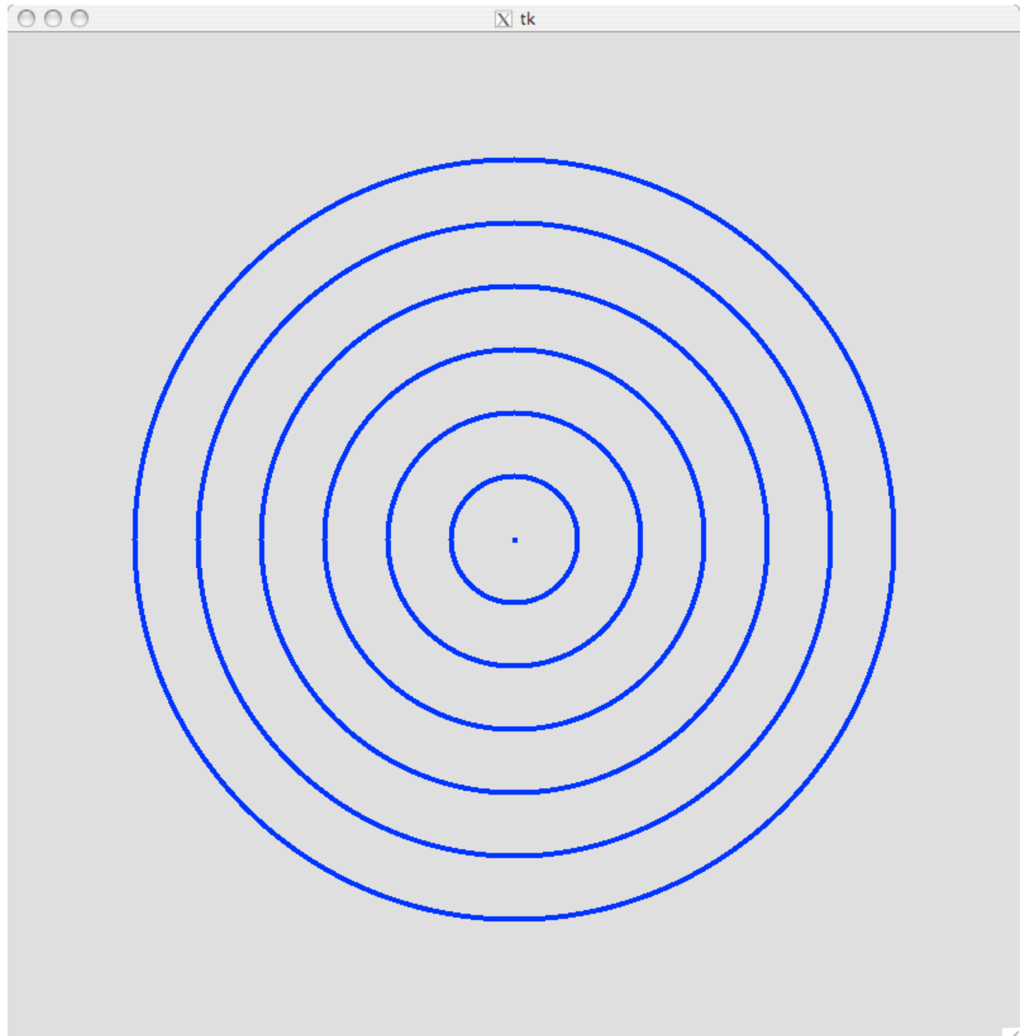
Oval example 4



Oval example 5

- It's easy to put graphics commands inside loops to create interesting images

Oval example 5



Oval example 5

- (Details left as exercise for the student)

Event handling

- So far, at the end of all of our graphics programs we wrote:

```
raw_input('Press <return> to quit')
```

- This is just to make sure the canvas stayed up while we looked at the graphics
- This is not the normal way to use **`Tkinter`**
- And in addition...

Event handling

- Our graphics programs have been totally *static*
- They display an image and that's all
- In reality, many more things can be done:
  - mouse click to exit the program
  - mouse click to create/delete/move graphical objects
  - bind keys to actions ('q' might mean 'quit')
- In other words, we want our graphics programs to be more *dynamic*
  - to respond to user input

Events and the event loop

- `Tkinter` programs are normally structured around an *event loop*
- The event loop is something that waits and "listens" for *events* happening on a graphical object

Events and the event loop

- Events are something you do while the program is running to notify the program that you want some action to occur
- Events may include:
  - key presses
  - mouse clicks
  - mouse movement
  - etc.

Events and the event loop

- Some events you might want:
  - "When I press the 'q' key, I want the program to exit."
  - "When I click the mouse, I want a square to be drawn on the canvas at the location of the mouse cursor."
  - "When I click the mouse, I want all the squares on the screen to be removed from the screen."

Specifying events

- Specifying events requires that you do these things:

1. Decide which graphical object is going to handle the events
   - *e.g.* the canvas, the root window
2. Decide which action will trigger which event
   - *e.g.* a mouse click, a key press
3. Write a function to handle each event
4. *Bind* the action to the event

`mainloop`

- Before we get into the details of this, we introduce the command which starts the event loop
- Assuming the root window is called `root`, we write

```
root.mainloop()
```

- to start the event loop

mainloop

```
root.mainloop()
```

- is an unusual method call
- Unlike most method calls, it doesn't normally return!
  - just loops forever
- Normally, it's a bad thing if a function or method call never terminates
- Here, it's what you want

`mainloop`

- What we need to know:
  - The event loop starts up when `root.mainloop()` executes
  - When events happen, the event loop will catch them and dispatch them to the functions that handle them
  - The details of how this works aren't important to us right now

mainloop

- Let's return to our first example:

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
r = c.create_rectangle(0, 0, 50, 50,
          fill='red', outline='red')
raw_input("Press <return> to quit")
```

mainloop

- Let's return to our first example:

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
r = c.create_rectangle(0, 0, 50, 50,
          fill='red', outline='red')
raw_input("Press <return> to quit")
```

mainloop

- Let's return to our first example:

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
r = c.create_rectangle(0, 0, 50, 50,
          fill='red', outline='red')
root.mainloop()
```

`mainloop`

- We added the

`root.mainloop()`

- line in place of the `raw_input` line
- The drawing doesn't change
- Now, the only way to exit the program is to close the window or to quit Python
- So far, haven't added any code to handle any events

Summary

- We've used the following **`Tkinter`** features:
  - **`Tk()`** function to create the root window
    - **`geometry()`** method
    - **`mainloop()`** method
  - **`Canvas()`** function to create the canvas object
    - **`pack()`** method
    - **`create_rectangle()`** method
    - **`create_line()`** method
    - **`create_oval()`** method

Next lecture

- More on event handling
  - callback functions
  - What's in an event?
- Graphical object *handles*
  - a way to manipulate graphical objects that have been created