Principles of Programming and Computer Science

What is Computer Science?

- the study of the principles and use of computers
- on the theoretical side
  - models of computation
  - what problems can be solved with a given model of computation
  - what problems can be solved efficiently with a given model of computation
- on the applied side
  - designing algorithms to solve problems in a variety of domains
  - producing software

- representation
- decomposition
- modularity
- abstraction

- efficiency
- complexity
- computability

- common programming elements – variables, assignment statements, functions, conditionals, loops, arrays
- patterns – recognizing when to use particular elements and common ways of using them
- algorithms – putting together the patterns to create a whole sketch
- syntax and semantics of Processing

Constructing Programs in Processing

1. What do you want to do?
   (determines program structure)

- draw a fixed picture
  → static (just a list of instructions) or active mode (`setup()` and `draw()`)
- have something that is different (or might be different) over time
  → active mode (`setup()` and `draw()`)

Constructing Programs in Processing

2. What kinds of elements do you have?
   (determines patterns and structure of sections of code)

- things which respond to user actions
  → interaction
  - mouse position → `mouseX`, `mouseY` system variables
  - mouse clicks → `mouseClicked()` event handler function
  - key presses → `keyPressed()` event handler function
- things which change over time
  → animation
  - individual things → animation variable(s)
  - many similar things animated similarly → arrays

Constructing Programs in Processing

2. What kinds of elements do you have?
   (determines patterns and structure of sections of code)

- a complex thing to draw (3+ shapes or many steps)
  → drawing function
  - what's different for different copies? → parameters
- a self-similar thing to draw (fractals)
  → recursive drawing function
  - each level adds more – additive pattern
  - each level replaces what was there – replacement pattern
  - L-system

Constructing Programs in Processing

2. What kinds of elements do you have?
   (determines patterns and structure of sections of code)

- different things at different times
  → conditional
  - can determine what to do based only on the current state (animation and system variables) → on-the-spot decision
  - need more information e.g. about past events → state machine
- repetition within a frame
  → loop
  - one or more properties changing predictably with each repetition → loop with one or more loop variables
  - repeat a known number of times → counting loop
  - two different patterns of repetition → one loop after another
  - several properties change but not all at the same time (grid pattern) → nested loops

Constructing Programs in Processing

2. What kinds of elements do you have?
   (determines patterns and structure of sections of code)

- things changing
  → variables (over time – animation variables, over repetition – loop variables)
  - four steps: declare, initialize, use, update

Constructing Programs in Processing

2. What kinds of elements do you have?
   (determines patterns and structure of sections of code)

- images
  - display an image
  - use an image as a source of colors
  - generate an image
  - generate an image whose colors are based on another (image filter)

- basic ingredients
  - load image
  - add image to sketch
- other ingredients
  - create a new image
  - load pixels to access
  - update pixels if changed
  - computing image (row,col) for on-screen (x,y)
  - computing pixel array loc for (row,col)

Constructing Programs

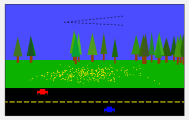
- some of these things are specific to Processing
  - static vs active mode, `setup()`, `draw()`
  - system variables like `mouseX`, `mouseY`
  - specific operations for drawing and working with images
  - specific event-handler functions like `mouseClicked()`
- some of these things exist in other languages
  - variables and assignment statements
  - functions
  - conditionals
  - loops
  - arrays
  - the notion of event-driven programming and event handlers

the patterns of usage we studied are not limited to Processing (and can be generalized beyond just creating pictures)

Principles and Key Concepts

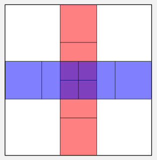
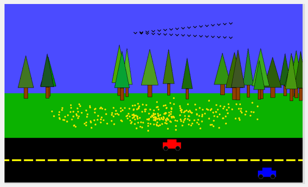
Constructing programs –

- an **algorithm** is a series of steps for solving a problem
  - algorithms are expressed in terms of constructs like variables, assignment statements, conditionals, loops
  - a program is an algorithm written in a particular programming language
- must identify the key elements and decide on how to **represent** them in the program
  - which elements to convey the scene's content
  - which shapes to depict scene elements
  - which variables (and what type) for animation and loops
  - which parameters (and what type) for functions

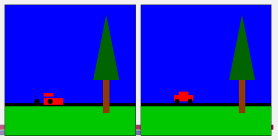


Principles and Key Concepts

- **decomposing** a task into smaller pieces is a key problem-solving strategy

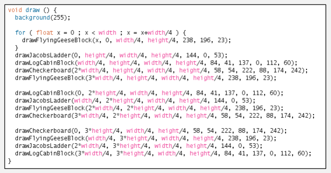
- **logical thinking** is important for problem solving and for debugging



Principles and Key Concepts

- organizing a program into self-contained **modules** makes it easier to understand and supports reuse
  - drawing functions
  - generator and filter functions for images

modules in programs include functions, classes, and packages

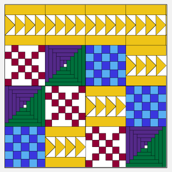


```
void draw () {
  background(255);

  for ( float x = 0 ; x < width ; x = x+width/4 ) {
    drawFlyingGeeseBlock(x, 0, width/4, height/4, 238, 196, 23);
  }
  drawJacobsLadder(0, height/4, width/4, height/4, 144, 0, 53);
  drawLogCabinBlock(width/4, height/4, width/4, height/4, 84, 41, 137, 0, 112, 60);
  drawCheckerboard(2*width/4, height/4, width/4, height/4, 58, 54, 222, 88, 174, 242);
  drawFlyingGeeseBlock(3*width/4, height/4, width/4, height/4, 238, 196, 23);

  drawLogCabinBlock(0, 2*height/4, width/4, height/4, 84, 41, 137, 0, 112, 60);
  drawJacobsLadder(width/4, 2*height/4, width/4, height/4, 144, 0, 53);
  drawFlyingGeeseBlock(2*width/4, 2*height/4, width/4, height/4, 238, 196, 23);
  drawCheckerboard(3*width/4, 2*height/4, width/4, height/4, 58, 54, 222, 88, 174, 242);

  drawCheckerboard(0, 3*height/4, width/4, height/4, 58, 54, 222, 88, 174, 242);
  drawFlyingGeeseBlock(width/4, 3*height/4, width/4, height/4, 238, 196, 23);
  drawJacobsLadder(2*width/4, 3*height/4, width/4, height/4, 144, 0, 53);
  drawLogCabinBlock(3*width/4, 3*height/4, width/4, height/4, 84, 41, 137, 0, 112, 60);
}
```



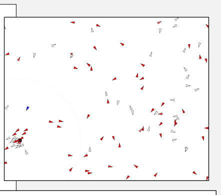
Principles and Key Concepts

- **abstraction** makes complex systems possible by replacing details with key ideas
  - do not need to know internal details of a module in order to use it
  - for functions, only need the function header and comments

```
// compute the evade steering vector
//  pos, vel - position and velocity of boid
//  maxspeed - boid's max speed
//  quarrypos, quarryvel - position and velocity of quarry
PVector computeEvade ( PVector pos, PVector vel, float maxspeed,
                       PVector quarrypos, PVector quarryvel ) { ... }

// compute the arrive steering vector
//  pos, vel - position and velocity of boid
//  maxspeed - boid's max speed
//  target - position of the target
//  threshold - distance from target at which the boid starts slowing
PVector computeArrive ( PVector pos, PVector vel, float maxspeed,
                        PVector target, float threshold ) { ... }

// compute the wander steeri
//  pos, vel - position and
//  maxspeed - boid's max sp
PVector computeWander ( PVec
```

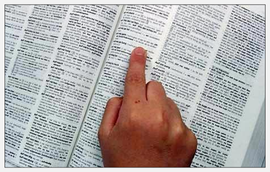


```
// compute steering force
PVector preySteer;
if ( dist(preyPos.x, preyPos.y, predatorPos.x, predatorPos.y) < 200 ) {
  preySteer = computeEvade(preyPos, preyVel, predatorPos, predatorVel, preyMaxspeed);
} else if ( hunger <= 0 && dist(foodx, foody, preyPos.x, preyPos.y) < 300 ) {
  preySteer = computeArrive(preyPos, preyVel, new PVector(foodx, foody), 100, preyMaxspeed);
} else {
  preySteer = computeWander(preyPos, preyVel, preyMaxspeed);
}
preySteer.limit(preyMaxforce);
```

Principles and Key Concepts

- **efficiency** deals with how long it takes to carry out an algorithm

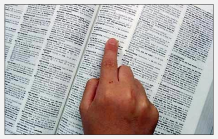
To look up a word in the dictionary –



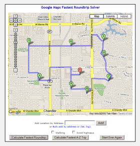
- start with the first word on the first page and keep reading until you find the word you are looking for
  - → the time required depends on the number of words — O(n)
- open to the middle page, see whether the first word on the page is before or after the word you are looking for, open to the middle of the appropriate half, repeat
  - → the time required depends on how many times the number of words is divided in half — O(log n)

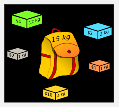
Principles and Key Concepts

- **complexity** addresses the inherent difficulty of a problem
  - considers how long the fastest solution will take to compute (not how challenging it might be to figure out an algorithm)
- "easy" problems
  - e.g. looking up a word in the dictionary
  - time required is at most a polynomial function of the size of the problem
    - don't have to look at every element
    - look at every element a fixed number of times
    - look at every pair or triple of elements



- "hard" problems
  - e.g. traveling salesman problem, knapsack problem
  - time required is exponential in the size of the problem
    - must look at all combinations of elements

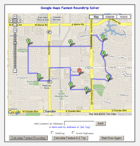


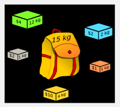


Principles and Key Concepts

- **computability** concerns whether or not a problem can be solved by a computer in any amount of time

- easy to come up with an efficient algorithm





- easy to come up with an algorithm (just try all possibilities)
- more difficult to find an algorithm fast enough to be practical for anything but the smallest problem sizes

- what about problems that are difficult to find an algorithm for?

Challenging Problems

- there are many challenging problems where there has been great progress (though lots of computing power is still required)
  - achieving realism in computer graphics and animation
    - photorealistic appearance
    - modeling and rendering of hair, fur, feathers, cloth, muscles, and skin
    - modeling and rendering of plants, trees, and terrain
    - complex group behavior
  - mimicking intelligence
    - expert knowledge
    - conversation (chatbots)
    - game playing
    - learning
    - image recognition and understanding
  - self-driving cars

...to name a few highly-visible areas

Impossible Problems

Some problems cannot be solved by a computer, no matter how clever the programmer.

- halting problem
  - assume you can write a program H which tells you if a program ever finishes running (halts) or not
  - create program K which takes another program P as input
    - K runs H on P to determine if P halts
    - if H says "yes, P stops", K goes into an infinite loop
    - if H says "no, P never ends", K terminates
  - what happens if you run K on itself?
    - if H says that K halts, K goes into an infinite loop
    - if H says that K doesn't halt, K halts
  - this paradox means that H cannot exist

Where To Go From Here

Want to learn more about Processing?

- there's more in the book that we didn't cover

Want to learn more about programming the natural world? (physics, particle systems, autonomous agents, cellular automata, fractals, genetic algorithms, neural networks)

- free (*) online book *The Nature of Code* by the author of *Learning Processing* – `http://natureofcode.com`
  - prerequisites: chapters 1-12 of *Learning Processing* (the main new material is objects, chapter 8)
  - (*) donation encouraged

Want to learn more about programming in general?

- take CPSC 124
- free online Java textbook *Introduction to Programming Using Java* – `http://math.hws.edu/javanotes/`