Lecture 3: Telling the computer what to do

Exercise: How do you tell a computer what to do?

Groups of two:
- Programmer
- Computer (Drawer)

Role of Programmer:
- Give instructions so “Computer” draws specified picture

Role of Computer (Drawer):
- Must follow instructions, but can do so in annoying way

What primitives are known?

Basic geometric shapes
- Line, circles, rectangles, octagons, hearts
- Not houses, not smiley faces, not trees

Numbers, sizes, and distances
- Quantitative measurements (inches, cm)
- Qualitative measurements (bigger, smaller)

Coordinates and layout
- Up (above), down (below), top, bottom, left, right, vertical, horizontal, middle, half, divide, center...

Step 1: Create (Secret) Picture

Draw a picture
- You will tell others how to copy

Make sure no one else in room sees!
- Will switch partners 3 times

Pick something interesting, but relatively simple
Step 2: Follow Instructions

Version 1: Full interaction
- Programmer watches computer/drawer
- Computer/Drawer can ask questions and get feedback
- Natural to be cooperative

Version 2: Visual feedback
- Programmer watches drawer and corrects mistakes
- Drawer cannot communicate or ask questions back

Version 3: No Feedback
- Programmer cannot watch drawer
- Drawer does not need to be cooperative but must follow directions (subject to interpretation)

Discussion Questions

Is English a good language for “programming”?  
- Other domains where English is not a good match?

How do different versions impact difficulty?

Which versions correspond to computer programming?

Take-Away Lessons

Programs need set of basic primitives

Multiple programs (drawings, outputs) can be made from those same instructions

Must be precise: English is not always

Versions: Easiest with more feedback and interaction

Traditional programming languages and projects often give no feedback until end: Version 3
- Scratch (very visual) continuously gives feedback: Version 2

Language for Exploring Algorithms

Need a programming language for
- Specifying and defining algorithms
- Evaluating and compare algorithms
- Executing algorithms

Options:
- English: Not precise enough
- Pseudo-code: More precise, but can’t execute it!
- Traditional languages: Assembly, C, Java, ...
Traditional Programming: C

```c
void requestError(int fd, char *cause, char *errnum, char *shortmsg, char *longmsg)
{
    char buf[MAXLINE], body[MAXBUF];
    printf("Request ERROR\n"); /* Create the body of the error message */
    sprintf(body, "<html><title>CS537 Error</title>\n" %s\n<body bgcolor="fffff">\n", body);
    sprintf(body, "%s%s: %s\n\n", body, errnum, shortmsg);
    sprintf(body, "%s\n\n", body);
    sprintf(body, "<hr>CS537 Web Server\n\n", body);

    /* Write out the header information for this response */
    sprintf(buf, "HTTP/1.0 %s %s\n", errnum, shortmsg);
    Rio_writen(fd, buf, strlen(buf));
    printf("%s\n", buf);
    sprintf(buf, "Content-Type: text/html\n\n", body);
    Rio_writen(fd, buf, strlen(buf));
    printf("%s\n", buf);
    sprintf(buf, "Content-Length: %d\n\n", strlen(body));
    Rio_writen(fd, buf, strlen(buf));
    printf("%s\n", buf);

    /* Write out the content */
    Rio_writen(fd, body, strlen(body));
    printf("%s\n", body);
}
```

```
```

Problems with Traditional Languages

High overhead to learning language
- Must get "syntax" just right
  - Keywords, semi-colon placement

Debugging can be frustrating
- Get wrong answer, must figure out why
- Program crashes, must figure out why

Sometimes hard to find motivating problems
- Toy exercises
- Results don’t always look sophisticated

New Introductory Language: Scratch

Low overhead for learning
- Specifically designed for beginners
- No syntax errors (drag and drop building blocks)

Bugs in program not (usually) frustrating
- Bugs are visual, so entertaining
- See bugs right away when problem occurs (Exercise)

Lots of creative projects
- Games, interactive art, music

Simplifies transition to other languages
- Same basic control structures, concepts

What essential features?

Computation: Perform calculations, work of algorithm
- Arithmetic and logical operations

Input/Output: Get data from user; Show result to user
- Input: Keyboard and mouse; Output: Display
- Scratch Limitations: Can’t access disk or network

Expressions: Query values and environment
- Ask questions: mouse clicked? Object touching edge?

Control Structures: Repeat loops, if statements
- Scratch Limitations
  - No procedures (helps structure large projects)
  - Cannot easily share versions of code across Sprites

Variables: Remember data while computing over it
- Store numbers, strings, lists
Scratch Demo

Overview parts of environment
- Stage, Sprites, Blocks, Scripts, Costumes, Sounds

Different categories of blocks
- Motion, Looks, Sound, Pen, Control, Sensing, Operators, Variables

Example Project: Make walking cat
- Each sprite has own code and costumes
- Code within a script runs sequentially (multiple scripts can run concurrently)
- Activate script with “hat” block
- Different backgrounds, different Sprites

Today’s Overview

Today’s Topics
- Motivation: English not precise enough for specifying algorithms
- Introduction to Scratch

Reading:
- “Specializing in Problems that Only Seem Impossible to Solve” [NyTimes, Dec 16 2008]

Announcements
- Assignment 1 Due Wednesday before Class
  - Grades for weekly homeworks: 10 point scale
- Download Scratch as needed from website