# Lecture 18—More A3, More Profiling Tools

## ECE 459: Programming for Performance

March 14, 2013

# Part I

## More A3 Discussion

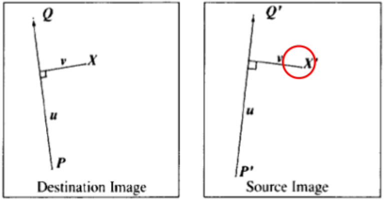
# Morphing = Warping + Cross-Dissolving

# Warping Examples: Single-Line Warp

Next few figures are from the original paper.

Another reference:

`http://www.cs.unc.edu/~lazebnik/research/fall08/qi_mo.pdf`.




$$u = \frac{(X - P) \cdot (Q - P)}{\| Q - P \|^2} \quad (1)$$

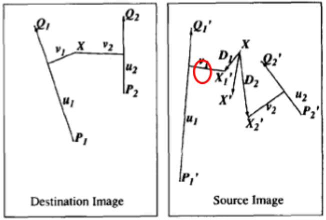
$$v = \frac{(X - P) \cdot Perpendicular(Q - P)}{\| Q - P \|} \quad (2)$$

$$X' = P' + u \cdot (Q' - P') + \frac{v \cdot Perpendicular(Q' - P')}{\| Q' - P' \|} \quad (3)$$

For each point in the destination, use the corresponding transformed point in the source.

# Warping Examples: Multiple-Line Warp

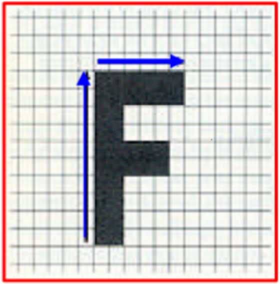
$$D_i = X_i' - X_i$$




$$weight = \left( \frac{length^p}{(a + dist)} \right)^b$$

Generalize the definition of “corresponding point” for multiple lines.

# Warped Image



# Parameters: $a$, $b$, $p$

The warping algorithm takes three parameters:

- $a$: smoothness of warping
  ($a$ near 0 means that lines go to lines)
- $b$: how relative strength of lines falls off with distance.
  large means every pixel only affected by nearest line;
  0 means each pixel affected by all lines equally.
  suggested range: $[0.5, 2]$
- $p$: relationship between length of line and strength.
  0 means all lines have same weight;
  1 means longer lines have greater relative weight;
  suggested range: $[0, 1]$.

# Algorithm Pseudocode

for each pixel $X$ in the destination:
    DSUM $\leftarrow$ (0, 0)
    weightsum $\leftarrow$ 0
    for each line $P_i Q_i$:
        calculate $u, v$ based on $P_i Q_i$
        calculate $X_i'$ based on $u, v$ and $P_i' Q_i'$
        calculate displacement $D_i = X_i' - X_i$ for this line
        dist $\leftarrow$ shortest distance from $X$ to $P_i Q_i$
        weight $\leftarrow$ (length$^p$ / (a+dist))$^b$
        DSUM $+=$ $D_i \times$ weight
        weightsum $+=$ weight
    X' = X + DSUM / weightsum
    destinationImage(X) $\leftarrow$ sourceImage(X$'$)

# Introduction

- The code is more or less a direct translation of the high level functions.
- The language should not be the main hurdle, if there's anything you don't understand about the provided code, feel free to talk to me.
- Code uses standard library functions plus Qt types.

# Built-in Data Structures

- QPoint
- QVector2D
- QImage

This is a fairly straightforward computation.

# Functions

All of the code that you need to deal with is in `model.cpp`.

For your purposes, this file is called from `test_harness.cpp`; it is also called from `window.cpp`, the interactive front-end.

`model` contains three methods:

- `prepStraightLine`: draws the lines and computes auxiliary lines;
- `commonPrep`: draws auxiliary lines;
- `morph`: carries out the actual morphing algorithm.

I refactored `model` out of the initial `window.cpp`, and may have made some mistakes. If you find them, fix them.

# Notes

I plan to add some more test cases and tweak the parameters shortly.

You'll probably want to refactor `morph()` to get useful profiling information from it.

You can remove code if you can prove it useless (using tests and text, which you should have anyway.)

# Part II

## System profiling: oprofile, perf, DTrace, WAIT

# Introduction: oprofile

`http://oprofile.sourceforge.net`

Sampling-based tool.

Uses CPU performance counters.

Tracks currently-running function;
records profiling data for every application run.

Can work system-wide (across processes).

Technology: Linux Kernel Performance Events
(formerly a Linux kernel module).

# Setting up oprofile

Must run as root to use system-wide, otherwise can use per-process.

```
% sudo opcontrol \
    --vmlinux=/usr/src/linux-3.2.7-1-ARCH/vmlinux
% echo 0 | sudo tee /proc/sys/kernel/nmi_watchdog
% sudo opcontrol --start
Using default event: CPU_CLK_UNHALTED:100000:0:1:1
Using 2.6+ OProfile kernel interface.
Reading module info.
Using log file /var/lib/oprofile/samples/oprofiled.log
Daemon started.
Profiler running.
```

Per-process:

```
[plam@lynch nm-morph]$ operf ./test_harness
operf: Profiler started

Profiling done.
```

# oprofile Usage (1)

Pass your executable to opreport.

```
% sudo opreport -l ./test
CPU: Intel Core/i7, speed 1595.78 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not
halted) with a unit mask of 0x00 (No unit mask) count 100000
samples  %        symbol name
7550     26.0749  int_math_helper
5982     20.6596  int_power
5859     20.2348  float_power
3605     12.4504  float_math
3198     11.0447  int_math
2601      8.9829  float_math_helper
160       0.5526  main
```

If you have debug symbols (-g) you could use:

```
% sudo opannotate --source \
--output-dir=/path/to/annotated-source /path/to/mybinary
```

# oprofile Usage (2)

Use opreport by itself for a whole-system view.
You can also reset and stop the profiling.

```
% sudo opcontrol --reset
Signalling daemon... done
% sudo opcontrol --stop
Stopping profiling.
```

# Perf: Introduction

`https://perf.wiki.kernel.org/index.php/Tutorial`

Interface to Linux kernel built-in sampling-based profiling.
Per-process, per-CPU, or system-wide.
Can even report the cost of each line of code.

# Perf: Usage Example

On the Assignment 3 code:

```
[plam@lynch nm-morph]$ perf stat ./test_harness

 Performance counter stats for './test_harness':

       6562.501429 task-clock                #    0.997 CPUs utilized
               666 context-switches          #    0.101 K/sec
                 0 cpu-migrations            #    0.000 K/sec
             3,791 page-faults               #    0.578 K/sec
    24,874,267,078 cycles                    #    3.790 GHz                     [83.32%]
    12,565,457,337 stalled-cycles-frontend   #   50.52% frontend cycles idle    [83.31%]
     5,874,853,028 stalled-cycles-backend    #   23.62% backend  cycles idle    [66.63%]
    33,787,408,650 instructions              #    1.36  insns per cycle
                                             #    0.37  stalled cycles per insn [83.32%]
     5,271,501,213 branches                  #  803.276 M/sec                   [83.38%]
       155,568,356 branch-misses             #    2.95% of all branches         [83.36%]

       6.580225847 seconds time elapsed
```

# Perf: Source-level Analysis

perf can tell you which instructions are taking time, or which lines of code.

Compile with `-ggdb` to enable source code viewing.

```
% perf record ./test_harness
% perf annotate
```

`perf annotate` is interactive. Play around with it.

# DTrace: Introduction

`http://queue.acm.org/detail.cfm?id=1117401`

Intrumentation-based tool.
System-wide.
Meant to be used on production systems. (Eh?)

# DTrace: Introduction

http://queue.acm.org/detail.cfm?id=1117401

Intrumentation-based tool.
System-wide.
Meant to be used on production systems. (Eh?)

(Typical instrumentation can have a slowdown of 100x (Valgrind).)
Design goals:

1. No overhead when not in use;
2. Guarantee safety—must not crash
   (strict limits on expressiveness of probes).

# DTrace: Operation

How does DTrace achieve 0 overhead?

- only when activated, dynamically rewrites code by placing a branch to instrumentation code.

Uninstrumented: runs as if nothing changed.

Most instrumentation: at function entry or exit points. You can also instrument kernel functions, locking, instrument-based on other events.

Can express sampling as instrumentation-based events also.

# DTrace Example

You write this:

```
syscall::read:entry {
    self->t = timestamp;
}

syscall::read:return
/self->t/ {
    printf("%d/%d spent %d nsecs in read\n"
           pid, tid, timestamp - self->t);
}
```

`t` is a thread-local variable.
This code prints how long each call to `read` takes, along with context.

To ensure safety, DTrace limits what you write; e.g. no loops.

- (Hence, no infinite loops!)

# Other Tools

AMD CodeAnalyst—based on oprofile; leverages AMD processor features.

WAIT

- IBM's tool tells you what operations your JVM is waiting on while idle.
- Non-free and not available.

# Other Tools

AMD CodeAnalyst—based on oprofile.

WAIT

- IBM's tool tells you what operations your JVM is waiting on while idle.
- Non-free and not available.

# WAIT: Introduction

Built for production environments.

Specialized for profiling JVMs, uses JVM hooks to analyze idle time.

Sampling-based analysis; infrequent samples (1–2 per minute!)

# WAIT: Operation

At each sample: records each thread's state,

- call stack;
- participation in system locks.

Enables WAIT to compute a "wait state" (using expert-written rules):
what the process is currently doing or waiting on, e.g.

- disk;
- GC;
- network;
- blocked;
- etc.

# WAIT: Workflow

You:

- run your application;
- collect data (using a script or manually); and
- upload the data to the server.

Server provides a report.

- You fix the performance problems.

Report indicates processor utilization (idle, your application, GC, etc); runnable threads; waiting threads (and why they are waiting); thread states; and a stack viewer.

Paper presents 6 case studies where WAIT identified performance problems: deadlocks, server underloads, memory leaks, database bottlenecks, and excess filesystem activity.

# Other Profiling Tools

Profiling: Not limited to C/C++, or even code.

You can profile Python using `cProfile`; standard profiling technology.

Google's Page Speed Tool: profiling for web pages—how can you make your page faster?

- reducing number of DNS lookups;
- leveraging browser caching;
- combining images;
- plus, traditional JavaScript profiling.

# Summary

- More Assignment 3 discussion
- System profiling tools:
  oprofile, DTrace, WAIT, perf

# Future Lectures

- OpenMPI
- OpenCL
- Hadoop MapReduce
- Software Transactional Memory
- Early Phase Termination, Big Data

If there's anything else you want to see in this course, let me know.