Lecture 18 — March 14, 2013

Patrick Lam *version 1*

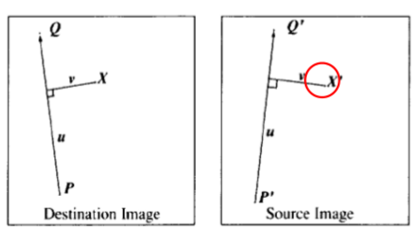
Notes on Assignment 3

The assignment is about morphing, which I've illustrated below.

The interesting part is the warp, which makes cross-dissolving look smooth. Your task is to optimize the warping algorithm. We'll talk about the warping algorithm first (see L17 for the reference), then the general code structure. The next few figures are from the original paper.

General Idea. For each point in the destination, use the corresponding point in the source, as determined by the warping transformation.

Single Warp. First, let's look at the single-line transformation. The idea is to find orthogonal (u, v) corresponding to each point in the destination, and work out what that is in the source.



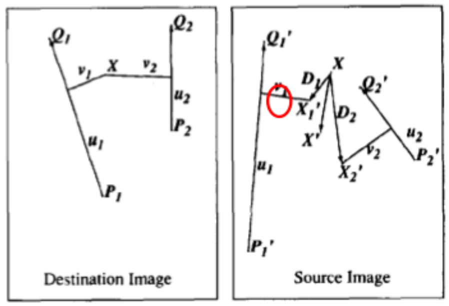
$$u = \frac{(X-P)\cdot(Q-P)}{\|Q-P\|^2} \qquad (1)$$

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

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

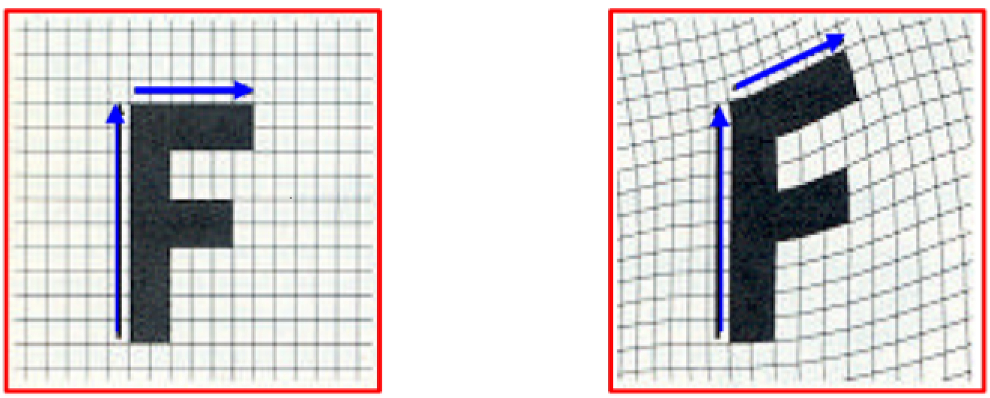
Multiple-Line Warp. A more typical case is that we have a number of features which we want to warp between. We therefore need to generalize the definition of "corresponding point", as seen in the following figure.

$$D_i = X_i' - X_i$$



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

Example. Warping on two axes can look like this.



Warp 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: $[0, 1]$.

Pseudocode. Here's some pseudocode for the warp.

for each pixel X in the destination:
  DSUM $\leftarrow$ (0, 0)
  weightsum $\leftarrow$ 0
  for each line P_iQ_i:
    calculate u, v based on P_iQ_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_iQ_i
    weight $\leftarrow$ (length^p / (a+dist))^b
    DSUM $+=$ $D_i\times$ weight
    weightsum $+=$ weight
  X' = X + DSUM / weightsum
  destinationImage(X) $\leftarrow$ sourceImage(X$'$)

Implementation Details

The code is more or less a direct translation of the high level functions. It uses standard library functions plus Qt types. 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.

Built-in Data Structures . The computation is fairly straightforward and uses arrays and Qt types:

- QPoint
- QVector2D
- QImage

Code structure. 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.

The `Model` class 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 and let me know. Thanks!

Notes. I plan to add some more test cases and tweak the parameters shortly. You may refactor `morph()` to get useful profiling information from it, although you don't have to if you use the right profiling tool. You can remove code if you can prove it useless (using tests and with a textual description, which you should provide anyway.)

System-level profiling

Most profiling tools interrogate the CPU in more detail than `gprof` and friends. These tools are typically aware of the whole system, but may focus on one application, and may have both per-process and system-wide modes. We'll discuss a couple of these tools here, highlighting conceptual differences between these applications.

Solaris Studio Performance Analyzer. (Did not discuss in lecture.) This tool[1] supports `gprof`-style profiling ("clock-based profiling") as well as kernel-level profiling through DTrace (described later). At process level, it collects more process-level data than `gprof`, including page fault times and wait times. It also can read CPU performance counters (e.g. the number of executed floating point adds and multiplies). As a Sun application, it also works with Java programs.

Since locks and concurrency are important, modern tools, including the Studio Performance Analyzer, can track the amount of time spent waiting for locks, as well as statistics about MPI message passing. More on lock waits below, when we talk about WAIT.

VTune. (Did not discuss in lecture). Intel and AMD both provide profiling tools; Intel's VTune tool costs money, while AMD's CodeAnalyst tool is free software.

Intel uses the term "event-based sampling" to refer to sampling which fires after a certain number of CPU events occur, and "time-based sampling" to refer to the `gprof`-style sampling (e.g. every 100ms). VTune can also correlate the behaviour of the counters with other system events (like disk workload). Both of these sampling modes also include the behaviour of the operating system and I/O in their counts.

VTune also supports an instrumentation-based profiling approach, which measures time spent in each procedure (same type of data as `gprof`, but using a different collection scheme).

VTune will also tell you what it thinks the top problems with your software are. However, if you want to understand what it's saying, you do actually need to understand the architecture.

CodeAnalyst. (Discussed oprofile in lecture, but not CodeAnalyst specifics). AMD also provides a profiling tool. Unlike Intel's tool, AMD's tool is free software (the Linux version is released under the GPL), so that, for instance, Mozilla suggests that people include CodeAnalyst profiling data when reporting Firefox performance problems [2].

[1] You can find a high-level description at `http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/oss-performance-tools-183986.pdf`

[2] `https://developer.mozilla.org/Profiling_with_AMD_CodeAnalyst`

CodeAnalyst is a system-wide profiler. It supports drilling down into particular programs and libraries; the only disadvantage of being system-wide is that the process you're interested in has to execute often enough to show up in the profile. It also uses debug symbols to provide meaningful names; these symbols are potentially supplied over the Internet.

Like all profilers, it includes a sampling mode, which it calls "Time-based profiling" (TBP). This mode works on all processors. The other modes are "Event-based profiling" (EBP) and "Instruction-based sampling" (IBS); these modes use hardware performance counters.

AMD's CodeAnalyst documentation points out that your sampling interval needs to be sufficiently high to capture useful data, and that you need to take samples for enough time. The default sampling rate is once every millisecond, and they suggest that programs should run for at least 15 seconds to get meaningful data.

The EBP mode works like VTune's event-based sampling: after a certain number of CPU events occur, the profiler records the system state. That way, it knows where e.g. all the cache misses are occuring. A caveat, though, is that EBP can't exactly identify the guilty statement, because of "skid": in the presence of out-of-order execution, guilt gets spread to the adjacent instructions.

To improve the accuracy of the profile information, CodeAnalyst uses AMD hardware features to watch specific x86 instructions and "ops", their associated backend instructions. This is the IBS mode[3] of CodeAnalyst. AMD provides an example[4] where IBS tracks down the exact instruction responsible for data translation lookaside buffer (DTLB) misses, while EBP indicates four potential guilty instructions.

oprofile. This free software is a sampling-based tool which uses the Linux Kernel Performance Events API to access CPU performance counters. It tracks the currently-running function (or even the line of code) and can, in system-wide mode, work across processes, recording data for every active application.

Webpage: `http://oprofile.sourceforge.net`.

You can run oprofile either in system-wide mode (as root) or per-process. To run it in system-wide mode:

```
% 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.
```

[3]Available on AMD processors as of the K10 family—typically manufactured in 2007+; see `http://developer.amd.com/assets/AMD_IBS_paper_EN.pdf`. Thanks to Jonathan Thomas for pointing this out.

[4]`http://developer.amd.com/cpu/CodeAnalyst/assets/ISPASS2010_IBS_CA_abstract.pdf`

Or, per-process:

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

 Profiling done.
```

Both of these invocations produce profiling output. You can read the profiling output by running `opreport` and giving it your executable.

```
% 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 can also get better data:

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

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. This uses the same base data as oprofile, but provides a better (git-like) interface. Once again, it is an interface to the Linux kernel's built-in sample-based profiling using CPU counters. It works per-process, per-CPU, or system-wide. It can report the cost of each line of code.

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

Here's a 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 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. DTrace[5][CSL04] is an instrumentation-based system-wide profiling tool designed to be used on production systems. It supports custom queries about system behaviour: when you are debugging system performance, you can collect all sorts of data about what the system is doing. The two primary design goals were in support of use in production: 1) avoid overhead when not tracing and 2) guarantee safety (i.e. DTrace can never cause crashes).

DTrace runs on Solaris and some BSDs. There is a Linux port, which may be usable. I'll try to install it on `ece459-1`.

Probe effect. "Wait! Don't 'instrumentation-based' and 'production systems' not go together[6]?"

Nope! DTrace was designed to have zero overhead when inactive. It does this by dynamically rewriting the code to insert instrumentation when requested. So, if you want to instrument all calls to the `open` system call, then DTrace is going to replace the instruction at the beginning of `open` with an unconditional branch to the instrumentation code, execute the profiling code, then return to your code. Otherwise, the code runs exactly as if you weren't looking.

Safety. As I've mentioned before, crashing a production system is a big no-no. DTrace is therefore designed to never cause a system crash. How? The instrumentation you write for DTrace must conform to fairly strict constraints.

DTrace system design. The DTrace framework supports instrumentation *providers*, which make *probes* (i.e. instrumentation points) available; and *consumers*, which enable probes as appropriate. Examples of probes include system calls, arbitrary kernel functions, and locking actions. Typically, probes apply at function entry or exit points. DTrace also supports typical sampling-based profiling in the form of timer-based probes; that is, it executes instrumentation every 100ms. This is tantamount to sampling.

You can specify a DTrace clause using probes, predicates, and a set of action statements. The action statements execute when the condition specified by the probe holds and the predicate evaluates to true. D programs consist of a sequence of clauses.

Example. Here's an example of a DTrace query from [CSL04].

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

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

[6]For instance, Valgrind incurs a 100× slowdown.

```
}

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

The first clause instruments all entries to the system call `read` and sets a thread-local variable `t` to the current time. The second clause instruments returns from `read` where the thread-local variable `t` is non-zero, calling `printf` to print out the relevant data.

The D (DTrace clause language) design ensures that clauses cannot loop indefinitely (since they can't loop at all), nor can they execute unsafe code; providers are responsible for providing safety guarantees. Probes might be unsafe because they might interrupt the system at a critical time. Or, action statements could perform illegal writes. DTrace won't execute unsafe code.

Workflow. Both the USENIX article [CSL04] and the ACM Queue article referenced above contain example usages of DTrace. In high-level terms: first identify a problem; then, use standard system monitoring tools, plus custom DTrace queries, to collect data about the problem (and resolve it).

WAIT

Another approach which recently appeared in the research literature is the WAIT tool out of IBM. Unfortunately, this tool is not free and not generally available. Let's talk about it anyways.

Like DTrace, WAIT is suitable for use in production environments. It uses hooks built into modern Java Virtual Machines (JVMs) to analyze their idle time. It performs a sampling-based analysis of the behaviour of the Java VM. Note that its samples are quite infrequent; they suggest that taking samples once or twice a minute is enough. At each sample, WAIT records the state of each of the threads, which includes its call stack and participation in system locks. This data enables WAIT to compute (using expert rules) an abstract "wait state". The wait state indicates what the process is currently doing or waiting on, e.g. "disk", "GC", "network", or "blocked".

Workflow. You run your application, collect data (using a script or manually), and upload the data to the server. The server provides a report which you use to fix the performance problems. The report indicates processor utilization (idle, your application, GC, etc); runnable threads; waiting threads (and why they are waiting); thread states; and a stack viewer.

The paper presents six case studies where WAIT helped solve performance problems, including deadlocks, server underloads, memory leaks, database bottlenecks, and excess filesystem activity.

Other Applications of Profiling.

Profiling applies to languages beyond C/C++ and Java, of course. If you are profiling an interpreted language, you'll need a specific tool to get useful results. For Python, you can use `cProfile`; it is a standard implementation of profiling, from what I can see.

Here's a short tangent. Many of the concepts that we've seen for code also apply to web pages. Google's Page Speed tool[7], in conjunction with Firebug, helps profile web pages, and provides suggestions on how to make your web pages faster. Note that Page Speed includes improvements for the web page's design, e.g. not requiring multiple DNS lookups; leveraging browser caching; or combining images; as well as traditional profiling for the JavaScript on your pages.

References

[CSL04] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 15–28, Berkeley, CA, USA, 2004. USENIX Association.

[7] `http://code.google.com/speed/page-speed/`