Dynamic Execution Tracing of Physical Simulations

Jonathan M. Cohen*
Sony Pictures Imageworks

1 Introduction

Software systems that simulate physical phenomena such as renderers, fluid simulation engines, or cloth dynamics engines, provide special difficulties during debugging. In particular, they often process vast amounts of data, making it necessary to trace not only individual values through an execution, but rather gather higher-level information about how whole classes of data are processed. Furthermore, the data flow paths and code paths through a simulation system can be extremely complex, as well as different for every variable in the system.

Consider, for example, the collision detection phase of a cloth simulator. A bug in the collision culling routine might result in a simulation that looks correct, but runs very slowly because too many triangles are being tested against each other for pair-wise collisions. Since this might only be happening in, say 10,000 out of 1 million cases, it could be difficult to figure out that this is occuring, let alone diagnose and fix the problem using conventional debugging tools such as gdb. Rather, we wish to take a more high-level approach that lets us formulate dynamic queries about overall behavior of the system. Ideally, this debugging code would not have to be embedded in the application itself, but could exist independently.

To address the performance analysis and debugging needs of these types of applications, Imageworks developed a dynamics package called *Sandstorm* which has an embedded tracing language called *sstrace*.

2 *sstrace* Basics

The sstrace language is based on Sun's DTrace language [Sun 2005]. A program consists of a set of function-like constructs called *probes*, which are identified with breakpoints in the simulation engine. When the various breakpoints are hit, the associated code in the probe will be executed. A simple probe might look like this:

```
cloth:ClothSolver@Step:enter {
  print -msg "Entered function " -var $PROBE
  timer -start clothtimer
}
```

This would be triggered when program execution entered the function `ClothSolver::Step` in the `cloth` module, and would print the given message and probe name to a log file, as well as start a time named `clothtimer`. Probes can also be specified via wildcards. For example, a probe `cloth:*:enter` would be triggered when any function in the `cloth` module is entered.

Unlike DTrace, we do not have access to kernel level traps in order to place hooks into our source code. Rather, the software developer links her library against the sstrace module, and explicitly adds hook into the source code via C++ macros. A typical set of such hooks might look like this:

```
int ClothSolver::Step(double time, double dt)
{
  T_PROBE_FUNCTION2(
    "cloth:ClothSolver@Step", time, dt);
  ...

  T_PROBE_FUNCTION_WATCH(errorCode);
  return errorCode;
}
```

The first macro specified two probes, `enter` and `exit`, which will be triggered when the program execution enters and exits the function. Probes can pass program data to the sstrace language via additional arguments to the `T_PROBE` macros. These variables can then be accessed in a sstrace script via built in variables called $ARG1$,$ARG2$,etc. The second probe in the above example acts like a watchpoint, and passes the `returnCode` value to sstrace.

3 Working Examples

Once the hooks are inserted into the C++ code, the sstrace script can be modified to profile, collect statistics, trace complex flow of control, or even print simple stack traces to log files, all without modifying or recompiling the simulation source code. Our current implementation has built-in functionality to start and stop timers, manipulate histograms, set indentation level for printouts, keep track of counters, trace memory usage, and print to log files. Even this basic level of functionality allows for a debugging and performance analysis to be carried out orthogonally to the main application development, similar to the Aspect-Oriented Programming paradigm [Kiczales et al. 1997].

Here is an example that prints an execution trace of the entire cloth module:

```
cloth:*:enter {
  print -var $PROBE
  indentation -increment 2
}
cloth:*:exit{
  indentation -decrement 2
}
```

Here is an example that prints a histogram of the number of collision tests per triangle during the collision phase at each frame:

```
cloth:ClothSolver@Step:enter {
  print -msg "Simulation Time " -var $ARG1
  histogram -clear collisions
}
cloth:Collision@ProcessTriangle:numtests {
  histogram -add collisions -addsample $ARG1
}
system:ClothSolver@Step:exit{
  histogram -print collisions
}
```

References

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of European Conference on Object-Oriented Programming*, vol. 1241, 220–242.

SUN MICROSYSTEMS. 2005. *Solaris Dynamic Tracing Guide*.

*e-mail: jcohen@imageworks.com