The Splendors and Miseries of Yade design

Václav Šmilauer

January 13, 2007

Abstract

This article maps software design and implementation deficiencies of Yade, a program for dynamic physical simulations. It does so from the point of a recent newcomer who got involved with the development who now tries to consider and suggests ways to rectify the situation.

Contents

1 Introduction **2**

2 Annoying issues **2**

3 Real future and imaginary vision **4**
3.1 Build system 5
3.2 Refactoring 6
3.2.1 Class renaming 6
3.2.2 Unused and superfluous components 7
3.2.3 File layout 7
3.3 Plugin loader 8
3.4 Parallelization 8
3.4.1 Code vectorization 8
3.4.2 Parallelization with shared memory 9
3.4.3 Parallelization with message passing 10
3.5 Reference benchmarks 11
3.6 Scripting 11

4 Conclusion **12**

1 Introduction

Yade[19] is a program for dynamic physical simulations, initiated by Fredéric Donzé, initially developed by Olivier Galizzi, later jointly with Janek Kozicki and finally by a few other persons from Laboratoire 3S, Grenoble. I joined the development in octobre 2006, originally motivated by modeling concrete damage under extreme loads.

Soon enough, it has become apparent that using Yade requires some understanding of its design, which was only poorly described by a few documents (see below). Since there were several issues with the code itself, I took the necessity of getting acquainted with sources as an opportunity to improve what I saw as deficient. In the following, I tried to list such deficiencies under two categories:

1. what I consider to be bugs that I encountered when routinely developing / using Yade;

2. what seems to me necessary to be addressed in long term — in the sense of issues that will get to surface sonner or later.

To avoid misunderstanding, I *do* think that the overall design of Yade is sound, which is the reason I point out individual deficiencies; if I thought that Yade were rotten from the roots up, I wouldn't bother enumerating blemishes. The positive aspects are present, but only implicitly and/or tacitly. The reader needs to concentrate very well to discern those.

The objection of non-constructive criticism can be easily anticipated, especially in cases when some points I make can be taken personally. Tho that, my answer is as follows. The popular difference of constructive and non-constructive criticism is not fine enough to understand that their difference is not in the content but much rather in the intention of the one who raises it. However, it would be absurd if I were destructive in my intention, since I would turn that destructivity against myself, as I do make part of the development of Yade; and will for several months to come.

2 Annoying issues

The following list is based on my (short) experience or discussions with other users. I tried to list these issues with decreasing importance.

No documentation. Since Yade's design is rather complex, a comprehensive overview of the overall functioning should be provided. Currently, except for two half-finished and half-obsolete articles ([20], [17]), there is no such thing. (Admittedly, Janek Kozicki was very willing to help on Yade's IRC channel or otherwise — and I thank him wholeheartedly — but this is not a substitute for real documentation.)

No documentation. Given that there is, from the design point of view, much emphasis on modularity in order to be able to use different algorithms, the lack of per-class documentation is quite cumbersome as well. Even though formally the Doxygen documentation framework is set up, it is used but very rarely; what should be explained at this level is what is the purpose of this particular class (e.g. `CunallNonViscousDamping` should contain reference to the relevant article by Cundall so that interested user may find rationale of such algorithm or its scope of use), how it interacts with other classes, what are its implementation limitations with regards to the algorithm. Class name clearly is not sufficient to contain all such information.

No documentation. On the most local level, source code is very rarely commented. The traditional maxim "the source code is the documentation"[14, pg. 54] (shouldn't it be "*the* documentation"?) is not applicable, as relatively little code contains algorithms in the proper sense. Vast majority of code is taken by declaration/definition duplication, framework macros for home-brown RTTI, serialization routines, class interactions, header includes. Along with rather low code/file ratio, this made my first encounter with Yade quite frustrating.

Overdesigned file layout. If UNIX were designed like Yade, we would have paths like `/usr/usr-local/usr-local-bin` — cf. `yade/yade-libs/yade-lib-serialization` and the like. In case that seems OK: the actual source code is nevertheless in `yade/yade-libs/yade-lib-serialization/src/yade-lib-serialization/`.

No proper error handling. Robustness is given by the ability to handle non-standard conditions. Functions should check their input values with assertions (these are compiled out for optimized builds, hence do not incur performance penalty) and *not* suppose that the user will validate them (even if validation rules *were* documented). Return values from calls that may fail (notably, system calls) should be checked consistently.

To name a few particular roblems of this kind that I encountered:

1. Unhandled exception (crash) if, within a plugin directory, there is a file with two dots or a dotted file. (Fixed, was a "portability" *feature* (!) of `boost::path`.)
2. Hang if filegenerator attempts to create dispatcher to non-existent class. (Fixed: exceptions do not propagate accross threads.)
3. At many places (file generators, recorders), output file stream is created but never checked; this has disappearing data for consequence.

A great bonus feature would be to install `SIGSEGV` handler that would attach debugger to Yade, permitting inspection of stack trace, open files, etc. This is superior to code forensics, since (a) no manual intervention is necessary and (b) the process still exists as a process.

Bad support for 3rd party modules. When building an out-of-tree module, compiler and linker flags should be consistent. The usual way is to use pkg-config[12] (used by e.g. gtk2 and qt4) or a file `config.h` included by all files. Work is being done on `pkg-config`.

No support for live debugging. Due to complexity of Yade functioning, it is not always feasible to run it from debugger. What seem to be the most important to me is a flexible logging framework (message severity, sources, destinations, filtering). (Experimental support is provided for the `log4cxx` library[7] with backward-compatible macro definition if this library is not present.)

3 Real future and imaginary vision

My opinion is that the following problems will be faced sooner or later. There is one and foremost, however, that I would call *realistic vision of development.* Yade was endowed with maximalism and design perfectionism at its very conception; striving for perfection — even for dubious "reasons" like self-affirmation — provides boldness in design and sometimes also endurance in implementation. But finding the balance between both is not necessarily not easy; resources are limited, because of one's own mortality as well as limited number of people, teams are pressed to have results as soon as possible, to name a few.

Many of my questions or buggy behavior reports were answered by "I already know how to improve that, I *just* (!) need to implement it", as if the genius needed only to think of a solution and the manual labour were secondary or not even worth the genius' time. In this way, less interesting parts (like detailed documentation, which is actually quite boring thing to do) are left alone in favor of planning unit-testing framework. I am sure that the framework itself will eventually be put in place; yeah, then someone *just* needs to write the actual tests. Pretty much in the same way as there is a proof-of-concept implementation of FEM that is otherwise useless, just to prove that the framework can work with FEM! The actual implementation of good FEM code is left as an exercise to the reader, hmm…

The vision of all-capable software necessarily leads to failure in long term, since the available workforce is too dispersed; therefore I think that Yade will have to narrow its scope. The vision is work of intellective prediction faculties tainted with excessive self-esteem: most of the planned work will never be done. Yade needs some real management (with some executive power) to steer the development so that long-term goals are met and evaluation of what has been done is performed.

3.1 Build system

The current qmake/make-based build system suffers has a number of issues.

1. Automatic configuration is not supported. Although it probably has not been an issue until now, using more external libraries and building in less homegeneous environments will enforce library detection, processor features etc. (Autotools have received a lot of justified criticism and are not to be considered at all.)

2. Parallel builds are not supported; major speedup when developing can be gained here. Multicore machine seem to be the future of computing for some years to come. Distcc can be used to distribute compilation accros network. (J. Kozicki once told me that his method for parallel builds is to run `make -j4` several times until it doesn't fail (!).)

3. Bad maintainability, as the files controlling compilation are scattered throughout the whole build tree. For example, recent addition of `Wm3Foundation` to all files to be linked against required a lot of scripting.

I tried to address these with SCons[13], which is extensively used and has an active community around (I dropped waf[15] for being much less mature and almost not used one-man project). Its notable features are:

- All configuration is localized in one `SConstruct` master file, target lists are in relatively few `SConscript` files. Option propagation from master file to individual targets is provided.
- Very good support for parallel builds (including distcc).
- Support for library detection, easy "profile" configuration (debug, profiling, optimized builds).
- SCons uses python for its scripts (and is itself written in Python), which gives virtually unlimited flexibility to the build process.

For now, `SConscript` files are generated automatically from qmake's project files, hence both build systems may be used at this time. However, I hope to be able to drop qmake entirely once SCons support stabilizes. This will pave our way to massive refactoring as described below.

3.2 Refactoring

Refactoring as I understand it here involves more than the file layout mentioned supra: reverting design decisions/side-effects that proved to be detrimental (besides ditching c++). Unfortunately, c++ has very few programs to assist refactoring (due to its syntax, c++ is very difficult to parse correctly) and most of the time, textual replacements with perl/awk/sed/... are used for such tasks.

3.2.1 Class renaming

Longish class names diminish readability of already unreadable code. For example, lexical distance `InteractingGeometry` and `InteractionGeometry` doesn't correspond to its conceptual distance. Table 1 summarizes changes

`BoundingVolume`†	`Bag`
`GeometricalModel`†	`Shape`
`InteractingGeometry`†	`Mold`
`InteractionGeometry`	`Bang`
`MetaEngine`	`Dispatcher`

Table 1: Current and proposed class names. Changes marked by † were aggreed upon by J. Kozicki.

that I proposed, being guided by the principle that root classes should have very short names: they are used very often and sometimes repeated in the names of the derived classes.

3.2.2 Unused and superfluous components

Some Yade libraries are available externally and have been put into the tree. The case of Wm3 geometry library[16] was already resolved by (painfully) reverting changes done by Olivier Galizzi (renaming many methods so that they are not capitalized) and linking against Wm3 library proper — furthermore, this version (now obsoleted by Wm4) has been licensed under the LGPL. Another candidate for cleanup is the Loki library[8].

Some components of Yade are always compiled but never used. `yade-lib-algorithms` was already removed from the tree, with `yade-lib-computational-geometry` following shortly.

Furthermore, the SCons build system now makes it possible to exclude larger parts of Yade from compilation/installation (e.g. dem, fem, lattice, ...).

3.2.3 File layout

Once the migration to SCons is complete, it will be much easier to move files in `[component]/src/[component]/` to `[component]/`. Later, renaming `yade/yade-libs/yade-lib-serialization` to `yade/libs/serialization` etc. is the step to come.

3.3 Plugin loader

The current plugin loader has a few issues that should be addressed earlier or later.

1. It deduces class name from library name. This could be rectified by e.g. inspecting symbols defined in the file (like the utility `nm` does) or by using a conventional symol (like `yadeExportedClasses`) that would contain class names defined in that particular file; the second approach is not automatic, but less difficult to implement.

2. Only one per plugin file, direct consequence of the preceding. That is annoying if one wants to have related classes in *one* implementation file. The workaround I use is to create symlinks with appropriate names to the library in question.

Despie the fact that many c++ programs use plugins, I was unable to find a robust and maintained library for it. The tentative solution I would propose is to

1. See whether the library defines a conventional symbol like `yadeExport-edClasses`. It would be an array of class names that this particular library contains. Every multi=class plugin would have to take care to use appropriate code to define that symbol. This is similar to module's method table in Python, for modules written in c/c++.

2. Otherwise, use the current fragile algorithm of class name inference from filename.

3.4 Parallelization

Parallel computation allows for significant speed gains and can be performed at different levels.

3.4.1 Code vectorization

Recent processors add "Single Instruction Multiple Data" (SIMD) instructions (SSE2 for the x86 platform, AltiVec for PowerPC), primarily perhaps for the reason of faster 3D gaming. These instructions perform e.g. vector addition (codeADDD) for 4 operands in 1 instruction. Vectorization is beneficial even on one processor and is not intrusive from the source point of

	implicit parallelization	explicit parallelization
implicit communication	automatic vectorization	openMP
explicit communication	—	MPI

Table 2: Parallelization techniques by parallelization and communication models, based on [4].

view. To make use of such techniques (reported speedups are in the order of multiples, for some applications), one either has to use specialized fine-tuned libraries (like Atlas[1]) or have compiler that is capable of vectorizing code automatically.

The first option is probably ruled out, perhaps save for a few performance-critical portions of the code. It *could* be beneficial to move to another vector and matrix library instead of Wm3 (`boost::ublas`[3] or Atlas[1]) for this reason.

The GNU compiler advances steadily[2, 6, 5] to make use of SIMD instructions where appropriate; loop vectorization is working (tested with a minimal example), sequential code vectorization is being worked on. C++ iterators seem to be an obstacle to good optimization of loops, though. A quick look into Wm3 sources further reveals, that e.g. vector addition is not performed using a loop but rather sequentially, moreover using `operator[]` instead of direct access to private component array members; therefore, Wm3 probably cannot masively benefit from automatic vetorization.

3.4.2 Parallelization with shared memory

Contrary to the previous case, at this level user is responsible for designating what portions of code should run in parallel. The emerging standard openMP[11], supported by both GNU and Intel compilers, uses #pragmas to insert parallelization directives; therefore, code can be compiled unmodified on compilers that do not support openMP or if it is disabled. Configurable number of threads is created, each one of them calculating some portion of the problem.

Multi-core and/or multi-processor machines may benefit from this kind of parallelization (possibly also Single-System-Image clusters that implement threads scattered accross nodes). Parallelization is still relatively non-intrusive and communication is implicit and quasi-instant (via shared memory).

3.4.3 Parallelization with message passing

The standard for message passing MPI[10] defines protocol that makes easier cross-network parallelization. Both parallelization and communication (with more or less important latencies — see below) are explicit and are very intrusive on the source level.

Rentability criterion. Contrary to FEM code that is computationally very intensive and benefits greatly from parallelization, DEM code tends to have comparatively low computation/communication ratio. Therefore, we should carefully consider possible benefits before laborious implementation.

Let us try to estimate roughly time that can be gained from MPI.[1] Single-node iteration takes t_1, iteration on MPI cluster with n ($n \geq 2$) nodes takes t_n. What we call "rentability criterion" is the condition $t_n < t_1$. If we suppose perfect linear scalability, we may estimate

$$t_n = \frac{t_1 + l_d}{n} + l_l, \tag{1}$$

where l_l is constant per-roundtrip latency given by physical, link, network and transport network layers and constant component of application layer; l_d is linear latency as function of amount of data being communicated (serialization and network transmission), thus is being distributed accross nodes (we simplify by taking $\frac{l_d}{n} \simeq \frac{l_d}{n-1}$); the actual per-node calculation time is $\frac{t_1}{n}$. Substituting (1) into the rentability condition, we obtain

$$\left(1 - \frac{1}{n}\right) t_1 > l_l + \frac{l_d}{n}. \tag{2}$$

If we consider imperfect scalability (with the coefficient $s < 1$) and $t'_n = st_n$, the factor $\left(1 - \frac{1}{n}\right)$ is further multiplied by s. Therefore, is yet more restrictive.

Now, since $\left(1 - \frac{1}{n}\right) s < 1$, we want $t_1 \gg l_l$ if the speedup is to be significant and justified by the effort of implementation. Whether l_d influences heavily this condition is estimated below.

[1]This part is almost literal copy of my message sent to `yade-dev@lits.berlios.de` 25/12/2006.

Rentability estimation. Let us suppose that physical system state needs to be synchronized between nodes after each iteration. We estimate $l_l = 10\,\mathrm{ms}$: [9] reports 5 ms on InfiniBand network, taking double latency on regular 1GBit switched UTP network is perhaps too optimistic. To estimate the order of l_d, let us consider $40\,\mathrm{MBs}^{-1}$ (on a switched Gbit network, provided that NICs are not on 32-bit PCI), we get to $40\,\frac{\mathrm{kB}}{\mathrm{ms}}$. Even if actual simulation requires large data amounts to be transmitted at each iteration, this time can be still reduced by compression, intelligent caching etc. — thusly the ommisionn of l_d above is justified.

Now, for 100 iterations/sec ($t_1 = 10\,\mathrm{ms}$), we have $t_1 \approx l_l$, therefore $t_1 \not\gg l_l$ and parallelization is not rentable. For lower iteration speed, t_1 may increase significantly and parallal computation may yield some speedup.

3.5 Reference benchmarks

After some substantial changes to code (like replacing math library or implementing alternative algorith for a particular task), one would like to measure its impact on overall performance on some set standardized simulations. Providing such a set would make it possible to quantitatively evaluate speedups/slowdowns from a particular change.

Further, frequently one would like to compare performance of Yade against some other software (be it SDEC, Yade's predecessor, or some commercially available program). Having performance data with exhaustive parameters (machine type, processor, memory, bus type, compilation options, library versions, etc.) would perhaps prevent software nacism ("Yade rulez, SDEC sucks") or justify it.

3.6 Scripting

This would be a "killer enhancement", though probably quite difficult to implement:

1. The original design clearly did not take inter-language cooperation into account;

2. c++ is difficult to be interfaced with.

Minimalist scripting would allow user to command Yade (simulation control, but also generating input files, evaluating given expression at given time) using another language. Maximalist scripting would permit coding a class in the other language, while it still would be used by the c++ core.

4 Conclusion

This article may provoke some discussion and I hope it will do so. Any feedback, be it positive or negative, should be sent to Yade developer's mailing list[18].

References

[1] Automatically Tuned Linear Algebra Software, `http://math-atlas.sourceforge.net`

[2] David Monniaux, *Automatic vectorization for the masses* (`http://www.advogato.org/article/871.html`).

[3] Boost::ublas library, `http://boost.org/libs/numeric/ublas/index.html`.

[4] Diego Novillo, *Parallel Programming with GCC*, `http://people.redhat.com/dnovillo/Papers/rhs2006.pdf`.

[5] *Autovect-branch optimizations*, `http://gcc.gnu.org/wiki/AutovectBranchOptimizations`.

[6] *Auto-vectorization in GCC*, `http://gcc.gnu.org/projects/tree-ssa/vectorization.html`.

[7] Log4cxx library, `http://logging.apache.org/log4cxx`.

[8] Loki library, `http://loki-lib.sourceforge.net/`.

[9] *Cluster Price/Performance Trends*, `http://lqcd.fnal.gov/trends.html`.

[10] Message Passing Interface, `http://www.mpi.org`

[11] OpenMP, `http://www.openmp.org`

[12] Pkg-config, `http://pkg-config.freedesktop.org`

[13] SCons, software construction tool, `http://www.scons.org`.

[14] S. Garfinkel, D. Weise and S. Strassmann, *The UNIX-HATERS Handbook*, IDG Books Worldwide, San Mateo, 1994.

[15] The Waf build system, `http://freehackers.org/~tnagy/bksys.html`.

[16] Wild Magic Geometric Tools, `http://www.geometrictool.com`.

[17] O. Galizzi, J. Kozićki, F. Donzé, *Application of modern software design for numerical simulations*, not (yet?) published.

[18] Yade developer's mailing list, `yade-dev@lists.berlios.de`; searchable archive `http://news.gmane.org/gmane.science.physics.yade.devel`.

[19] Yade, Yet Another Dynamic Engine, `yade.berlios.de`.

[20] O. Galizzi, *YADE User's Manual (version 0.5)*, `http://svn.berlios.de/wsvn/yade/trunk/yade-doc/UserManual/YadeUserManual.tex?op=file&rev=0&sc=0`.