**Line Between Scripting and Compiled languages:**

* At some point, with so many readily available, easy to use (and inexpensive or free!) high-level scripting languages to choose from, why would anyone still resort to C, C++, or Java? We can think of several reasons, but execution speed and the desire to keep algorithms private are probably the dominant ones. Both of these reasons are going away, however, with increases in processor speed and the advent of compilers for traditionally interpreted scripting languages. \_\_http://www.mactech.com/articles/mactech/Vol.15/15.09/ScriptingLanguages/index.html
* *“The ACM flagship, COMMUNICATIONS OF THE ACM for example, has never published a paper recognizing the scripting philosophy, and the references throughout the ACM Digital Library to scripting are not encouraging.” –* by Ronald P. Loui  
  + \_\_In Praise of Scripting: Real Programming Pragmatism \_\_http://www.cse.wustl.edu/~loui/praiseieee.html
* Scripting languages have yet to achieve the amount of users and their respect/reputation for them, especially in undergraduate users, as the compiled languages, given the major barriers that keeps users on their back foot.

**Disadvantages:**

* As useful as they are, conventional scripting languages have syntactic and other limitations, however, which tend to limit their efficiency and expressive power. For instance, even "advanced" Unix shells (e.g., bash and ksh) have very limited support for concurrency, data structuring, information hiding, object-oriented programming, regular expressions, etc.   
  \_\_http://www.mactech.com/articles/mactech/Vol.15/15.09/ScriptingLanguages/index.html
  + **Turning Point:** Responding to these deficiencies, language developers have created extended scripting languages such as Perl, Python, and Tcl/Tk, among others. These languages are still able to invoke and glue together other programs, manipulate files, and set values on-the-fly, but they are also appropriate for writing much larger applications.  
    \_\_http://www.mactech.com/articles/mactech/Vol.15/15.09/ScriptingLanguages/index.html

**Advantages:**

(flexibility of typelessness, rapid turnaround of interpretation, higher level semantics, development speed, appropriateness for gluing components and internet programming, ease of learning and increase in amount of casual programming)

**Scripting in Games:**

Scripts are usually **compiled at run time**, while the host language will be compiled at compile time. This means that we don't need to recompile if the script changes. Recompiling a full game can take minutes to hours, which implies a big productivity hit.

Usually, the critical code or backend code will not be scripted. This code should run fast and often memory management is crucial.

In games, game logic and configuration are typically contained in script files. These **scripts can easily be updated by non-programmers** (like the designer) to tweak the gameplay. Script languages are easy and act in a forgiving manner for that purpose.

Often, a script language is also used to do **scripting at real time**. This comes in handy for tweaking some gameplay elements or even for debugging. Many games provide a console for this (mostly in-house) purpose.

It is very well possible that you create a game using an existing game engine, just by scripting. The **game engine layer is thus fully decouple from the game logic layer**. Modern engines can usually be used to create FPS or RTS games easily like this, but it is not possible for any genre. An MMO would probably require another type of engine.

So the bottom line is decoupling. The benefits listed above often outweigh the extra work to create or integrate a scripting language.

\_\_http://gamedev.stackexchange.com/questions/2913/why-do-we-use-scripts-in-development

**Introduction**

Given the choice made in the practical classes of DJCO of the theme for the game development assignment, scripting languages will be the major subject written on this report.

Scripting languages have been around for a few years and have now been gaining a bit more popularity amongst the more casual programmers but in games they have, for a while now, been there above the core games functions.

This report will cover the basics of scripting languages and their context in game development issuing some relevance to their advantages and disadvantages facing other types programming languages, mainly compiled languages.

Part of the problem is that scripting has risen in the shadow of

object-oriented programming and highly publicized corporate battles

between Sun, Netscape, and Microsoft with their competing software

practices. Scripting has been appearing language by language,

including object-oriented scripting languages now. Another part of the

problem is that scripting is only now mature enough to stand up against

its legitimate detractors. Today, there are answers to many of the

persistent questions about scripting: is there a scripting language

appropriate for the teaching of CS1 (the first programming course for

majors in the undergraduate computing curriculum)? Is there a scripting

language for enterprise or real-time applications? Is there a way for

scripting practices to scale to larger software engineering projects?

I intend to review the recent history briefly for those who have not yet

joined the debate, then present some of the answers that scripting

advocates now give to those nagging questions. Finally, I will describe

how a real pragmatism of academic interest in programming languages would

have better prepared the academic computing community to see the changes

that have been afoot.

1996-1998 are perhaps the most interesting years in the phylogeny of

scripting. In those years, perl "held the web together", and together

with a new POSIX awk and GNU gawk, was shipping with every new Linux.

Meanwhile javascript was being deployed furiously (javascript bearing no

important relation to java, having been renamed from "livescript" for

purely corporate purposes, apparently a sign of Netscape's solidarity with

Sun, and even renamed "jscript" under Microsoft). Also, a handoff from

tcl/tk to python was taking place as the language of choice for GUI

developers who would not yield to Microsoft's VisualBasic. Php appeared

in those years, though it would take another round of development before

it would start displacing server-side perl, cold fusion, and asp. Every

one of these languages is now considered a classic, even prototypical,

scripting language.

Already by mid-decade, the shift from scheme to java as the dominant CS1

language was complete, and the superiority of c++ over c was unquestioned

in industry. But java applets were not well supported in browsers, so the

appeal of "write once, run everywhere" quickly became derided as "write

once, debug everywhere." Web page forms, which used the common gateway

interface (cgi) were proliferating, and systems programming languages like

c became recognized as overkill for server-side programming. Developers

quickly discovered the main advantage of perl for cgi forms processing,

especially in the dot-com setting: it minimized the programmer's

write-time. What about performance? The algorithms were simple, network

latency masked small delays, and database performance was built into the

database software. It turned out that the bottleneck was the

programming. Even at run-time, the network and disk properties were the

problems, not the cpu processing. What about maintenance? The developers

and management were both happy to rewrite code for redesigned services

rather than deal with legacy code. Scripting, it turns out, was so

powerful and programmer-friendly that it was easier to create new scripts

from scratch than to modify old programs. What about user interface?

After all, by 1990, most of the programming effort had become the writing

of the GUI, and the object-oriented paradigm had much of its momentum in

the inheritance of interface widget behaviors. Surprisingly, the

interface that most programmers needed could be had in a browser. The

html/javascript/cgi trio became the GUI, and if more was needed, then

ambitious client-side javascript was more reliable than the browser's java

virtual machine. Moreover, the server-side program was simply a better

way to distribute automation in a heterogeneous internet than the

downloadable client-side program, regardless of whether the download was

in binary or bytecode.

Although there was not agreement on what exact necessary and sufficient

properties characterized scripting and distinguished it from "more

serious" programming, several things were clear:

scripting permitted rapid development, often regarded as merely

"rapid prototyping," but subsequently recognized as a kind of agile

programming;

scripting was the kind of high-level programming that had always

been envisioned, in the ascent from low-level assembly language

programming to higher levels of abstraction: it was concise,

and it removed programmers from concerning themselves with

many performance and memory management details;

scripting was well suited to the majority of a programming task,

usually the accumulation, extraction, and transformation of data,

followed eventually by its presentation, so that only the

performance-critical portion of a project had to be written in a

more cumbersome, high-performance language;

it was easier to get things right when source code was short, when

behavior was determined by code that fit on a page, all types were

easily coerced into strings for trace-printing, code fragments

could be interpreted, identifiers were short, and when the

programmer could turn ideas into code quickly without losing

focus.

This last point was extremely counterintuitive. Strong typing, naming

regimen, and verbosity were motivated mainly by a desire to help the

programmer avoid errors. But the programmer who had to generate too many

keystrokes and consult too many pages, who had to search through many

different files to discover semantics, and who had to follow too many

rules, who had to sustain motivation and concentration over a long period

of time, was a distracted and consequently inefficient programmer. Just

as vast libraries did not deliver the promise of greater reusability, and

virtual machines did not deliver the promise of platform-independence, the

language's promise to discipline the programmer quite simply did not

reduce the tendency of humans to err. It exchanged one kind of frequent

error for another.

Scripting languages became the favorite tools of the independent-minded

programmers: the "hackers" yes, but also the gifted and genius

programmers who tended to drive a project's design and development. As

Paul Graham noted (in a column reprinted in ["Hackers and Painters"](http://www.paulgraham.com/gh.html) or [this](http://www.paulgraham.com/javacover.html)), one

of the lasting and legitimate benefits of java is that it permits managers

to level the playing field and extract considerable productivity from the

less talented and less motivated programmers (hence, more disposable).

There was a corollary to this difference between the mundane and the

liberating:

scripting was not enervating but was actually renewing:

programmers who viewed code generation as tedious and tiresome in

contrast viewed scripting as rewarding self-expression or

recreation.

The distinct features of scripting languages that produce these effects

are usually enumerated as semantic features, starting with low I/O

specification costs, the use of implicit coercion and weak typing,

automatic variable initialization with optional declaration, predominant

use of associative arrays for storage and regular expressions for pattern

matching, reduced syntax, and powerful control structures. But the main

reason for the productivity gains may be found in the name "scripting"

itself. To script an environment is to be powerfully embedded in that

environment. In the same way that the dolphin reigns over the open ocean,

lisp is a powerful language for those who would customize their emacs,

javascript is feral among browsers, and gawk and perl rule the linux

jungle.

There is even a hint of AI in the idea of scripting: the scripting

language is the way to get high level control, to automate by capturing

the intentions and routines normally provided by the human. If recording

and replaying macros is a kind of autopilot, then scripting is a kind of

proxy for human decisionmaking. Nowhere is this clearer than in simple

server-side php, or in sysadmin shell scripting.

So where do we stand now? While it may have been risky for Ousterhout to

proclaim scripting on the rise in 1998, it would be folly to dismiss the

success of scripting today. It is even possible that java will yield its

position of dominance in the near future. (By the time this essay is

printed, LAMP and AJAX might be the new darlings of the tech press;

see recent articles in Business Week, this IEEE COMPUTER, and even James

Gosling's blog where he concedes he was wanting to write a scripting

language when he was handed the java project. Java is very much in full

retreat.) Is scripting ready to fill the huge vacuum that would be

produced?

I personally believe that CS1 java is the greatest single mistake in the

history of computing curricula. I believe this because of the empirical

evidence, not because I have an a priori preference (I too voted to shift

from scheme to java in our CS1, over a decade ago, so I am complicit in

the java debacle). I reported in [SIGPLAN 1996 ("Why gawk for AI?")](http://www.softpanorama.org/Scripting/Scripting_wars/classic_papers_on_scripting.shtml) that

only the scripting programmers could generate code fast enough to keep up

with the demands of the artificial intelligence laboratory class. Even

though students were allowed to choose any language they wanted, and many

had to unlearn the java ways of doing things in order to benefit from

scripting, there were few who could develop ideas into code effectively

and rapidly without scripting. In the intervening decade, little has

changed. We actually see more scripting, as students are happy to

compress images so that they can script their computer vision projects

rather than stumble around in c and c++. In fact, students who learn to

script early are empowered throughout their college years, especially in

the crucial UNIX and web environments. Those who learn only java are

stifled by enterprise-sized correctness and the chimerae of just-in-time

compilation, swing, JRE, JINI, etc. Young programmers need to practice

and produce, and to learn through mistakes why discipline is needed. They

need to learn design patterns by solving problems, not by reading

interfaces to someone else's black box code. It is imperative that

programmers learn to be creative and inventive, and they need programming

tools that support code exploration rather than code production.

What scripting language could be used for CS1? My personal preferences

are gawk, javascript, php, and asp, mainly because of their very gentle

learning curves. I don't think perl would be a disaster; its imperfection

would create many teaching moments. But there is emerging consensus in

the scripting community that python is the right choice for freshman

programming. Ruby would also be a defensible choice. Python and ruby

have the enviable properties that almost no one dislikes them, and almost

everyone respects them. Both languages support a wide variety of

programming styles and paradigms and satisfy practitioners and

theoreticians equally. Both languages are carefully enough designed that

"correct" programming practices can be demonstrated and high standards of

code quality can be enforced. The fact that Google stands by python is an

added motivation for undergraduate majors.

But do scripting solutions scale? What about the performance gap when the

polynomial, or worse the exponential, algorithm faces large n, and the

algorithm is written in an interpreted or weakly compiled language? What

about software engineering in the large, on big projects? There has been

a lot of discussion about scalability of scripts recently. In the past,

debates have simply ended with the concession that large systems would

have to be rewritten in c++, or a similar language, once the scripting had

served its prototyping duty.

The enterprise question is the easier of the two. Just as the individual

programmer reaps benefits from a division of labor among tools, writing

most of the code in scripts, and writing all bottleneck code in a highly

optimizable language, the group of programmers benefits from the use of

multiple paradigms and multiple languages. In a recent large project, we

used vhdl for fpga's with a lot of gawk to configure the vhdl. We

used python and php to generate dynamic html with svg and javascript for

the interfaces. We used c and c++ for high performance communications

wrappers, which communicated xml to higher level scripts that managed

databases and processes. We saw sysadmin and report-generation in perl,

ruby, and gawk, data scrubbing in perl and gawk, user scripting in bash,

tcl, and gawk, and prototyping in perl and gawk. Only one module was

written in java (because that programmer loved java): it was late, it was

slow, it failed, and it was eventually rewritten in c++. In retrospect,

neither the high performance components nor the lightweight code

components were appropriate for the java language. Does scripting scale

to enterprise software? I would not manage a project that did not include

a lot of scripting, to minimize the amount of "hard" programming, to

increase flexibility and reduce delivery time at all stages, to take

testing to a higher level, and to free development resources for

components where performance is actually critical. I nearly weep when I

think about the text processing that was written in c under my managerial

watch, because the programmer did not know perl. We write five hundred

line scripts in gawk that would be ten thousand line modules in java or

c++. In view of the fact that there are much better scripting tools for

most of what gets programmed in java and c++, perhaps the question is

whether java and c++ scale.

How about algorithmic complexity? Don't scripting languages take too long

to perform nested loops? The answer here is that a cpu-bound tight loop

such as a matrix multiplication is indeed faster in a language like c.

But such bottlenecks are easy to identify and indeed easy to rewrite in

c. True system bottlenecks are things like paging, chasing pointers on

disk, process initialization, garbage collection, fragmentation, cache

mismanagement, and poor data organization. Often, we see that better data

organization was unimplemented because it would have required more code,

code that would have been attempted in an "easier" programming language

like a scripting language, but which was too difficult to attempt in a

"harder" programming language. We saw this in the AI class with heuristic

search and computer vision, where brute force is better in c, but complex

heuristics are better than brute force, and scripting is better for

complex heuristics. When algorithms are exponential, it usually doesn't

matter what language you use because most practical n will incur too great

a cost. Again, the solution is to write heuristics, and scripting is the

top dog in that house. Cpu's are so much faster than disks these days

that a single extra disk read can erase the CPU advantage of using

compiled c instead of interpreted gawk. In any case, java is hardly the

first choice for those who have algorithmic bottlenecks.

The real reason why academics were blindsided by scripting is their lack

of practicality. Academic computing was generally late to adopt Wintel

architectures, late to embrace cgi programming, and late to accept Linux

in the same decade that brought scripting's rise. Academia understandably

holds industry at a distance. Still, there is a purely intellectual

reason why programming language courses are only now warming to

scripting. The historical concerns of programming language theory have

been syntax and semantics. Java's amazing contribution to computer

science is that it raised so many old-fashioned questions that tickled the

talents of existing programming language experts: e.g., how can it be

compiled? But there are new questions that can be asked, too, such as

what a particular language is well-suited to achieve inexpensively,

quickly, or elegantly, especially with the new mix of platforms. The

proliferation of scripting languages represents a new age of innovation in

programming practice.

Linguists recognize something above syntax and semantics, and they call it

"pragmatics". Pragmatics has to do with more abstract social and

cognitive functions of language: situations, speakers and hearers,

discourse, plans and actions, and performance. We are entering an era of

comparative programming language study when the issues are higher-level,

social, and cognitive too.

My old friend, Michael Scott, has a popular textbook called PROGRAMMING

LANGUAGE PRAGMATICS. But it is a fairly traditional tome concerned with

parameter passing, types, and bindings (it's hard to see why it merits

"pragmatics" in its title, even as it goes to second edition with a

chapter on scripting added!). A real programming pragmatics would ask

questions like:

how well does each language mate to the other UNIX tools?

what is the propensity in each language for programmers

at various expertise levels to produce a memory leak?

what is the likelihood in each language that unmodified

code will still function in five years?

what is the demand of a programmer's concentration, what is the

load on her short-term memory of ontology, and what is the support

for visual metaphor in each language?

There have been programming language "shootouts" and "scriptometers" on

the internet that have sought to address some of the questions that are

relevant to the choice of scripting language, but they have been just

first steps. For example, one site reports on the shortest script in each

scripting language that can perform a simple task. But absolute brevity

for trivial tasks, such as "print hello world" is not as illuminating as

typical brevity for real tasks, such as xml parsing.

Pragmatic questions are not the easiest questions for

mathematically-inclined computer scientists to address. They refer by

their nature to people, their habits, their sociology, and the

technological demands of the day. But it is the importance of such

questions that makes programmers choose scripting languages. Ousterhout

declared scripting on the rise, but perhaps so too are programming

language pragmatics.

ACKNOWLEDGEMENTS

I have to thank Charlie Comstock for contributing many ideas and

references over the past two years that have shaped my views,

especially the commitment to the idea of pragmatics.

Prof. Dr. Loui and his students are the usual winners of the department

programming contest and have contributed to current gnu releases of gawk

and malloc. He has lectured on AI for two decades on five continents,

taught AI programming for two decades, and is currently funded on a

project delivering hardware and software on U.S. government contracts.

REFERENCES

Graham, P., Hackers and Painters, O'Reilly, 2004.

Levinson, S., Pragmatics, Cambridge University Press, 1983.

Loui, R.P., Why Gawk for AI? SIGPLAN Notices 31(8): 8-9, 1996.

Ousterhout, John K., Scripting: Higher Level Programming for the

21st Century, IEEE Computer, March 1998.

Shannon, Christine, Another breadth-first approach to CS I using Python,

Proceedings of the 34th SIGCSE technical symposium on

Computer science education, Reno, 2003.

Scott, Michael L., Programming Language Pragmatics, Morgan Kaufman 2000.

Spinellis, Diomidis. Java makes scripting languages irrelevant? IEEE

Software, 22(3):70-71, May/June 2005.

Zelle, J.M., Python as a First Language, 13th Annual Midwest Computer

Conference, 1999.

The Great Win32 Computer Language Shootout

<http://dada.perl.it/shootout/>

Scriptometer: measuring the ease of SOP (Script-Oriented Programming)

<http://merd.sourceforge.net/pixel/language-study/scripting-language/>

A Slightly Skeptical View on Scripting Languages, Dr. Nikolai Bezroukov

<http://www.softpanorama.org/Articles/a_slightly_skeptical_view_on_scripting_languages.shtml>

WIKIPEDIA: Scripting Language

<http://en.wikipedia.org/wiki/Scripting_language>

WIKIPEDIA: Java Programming Language/Criticism

<http://en.wikipedia.org/wiki/Java_programming_language#Criticism>

Java? It's So Nineties

<http://www.businessweek.com/technology/content/dec2005/tc20051213_042973.htm>

Quotes about Java - Smalltalk

<http://www.sysprog.net/quotjava.html>

The PHP Scalability Myth

<http://www.onjava.com/pub/a/onjava/2003/10/15/php_scalability.html>

Gosling: RADlab, scripting and scale

<http://blogs.sun.com/roller/page/jag?entry=radlab_scripting_and_scale>

More Gosling, Jan 9

<http://news.com.com/Is+Java+getting+better+with+age/2008-7345_3-6022062.html>

"Java Is Dead, Long Live Java!" The Future of Java By: Bryan W. Taylor

(check out the very last line... hardly a defense of java!)

<http://au.sys-con.com/read/169595.htm>

Open-source LAMP a beacon to developers

<http://news.com.com/Open-source+LAMP+a+beacon+to+developers/2100-7344_3-5744767.html>

The Perils of JavaSchools

<http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html>

Java's Demise

<http://dsonline.computer.org/portal/site/dsonline/menuitem.9ed3d9924aeb0dcd82ccc6716bbe36ec/index.jsp?&pName=dso_level1&path=dsonline/2006/02&file=o2004.xml&xsl=article.xsl&>

Ruby on Rails

<http://csdl2.computer.org/comp/mags/co/2006/02/r2018.pdf>

Beyond Java

<http://books.slashdot.org/article.pl?sid=06/02/01/1455213&from=rss>

Stevey on Python culture

<http://www.cabochon.com/~stevey/blog-rants/anti-anti-hype.html>

Learn to talk awk (see end comment about 12,000 lines -- we've done 2k-5k a lot)

<http://www.linux.com/article.pl?sid=06/01/10/2211211>

Not in the Script: News of Java's Demise Is Premature Greg Goth

<http://dsonline.computer.org/portal/site/dsonline/menuitem.9ed3d9924aeb0dcd82ccc6716bbe36ec/index.jsp?&pName=dso_level1&path=dsonline/2006/02&file=o2004.xml&xsl=article.xsl&>

Older prechelt ieee sofwtare study on seven languages

<http://page.mi.fu-berlin.de/~prechelt/Biblio/jccpprtTR.pdf>

which reminded me of kernighan's '98 thing

<http://netlib.bell-labs.com/cm/cs/who/bwk/interps/pap.html>

http://www.cs.up.ac.za/cs/jbishop/Homepage/Pubs/Tech-reports/Sacla99.pdf