CMP Game Media Group Presents:

Gamasutra
The Art and Science of Making Games

Toward Programmer Interactivity: Writing Games In Modern Programming Languages

By James Hague
Gamasutra
August 13, 1999

URL: http://www.gamasutra.com/features/19990813/languages_01.htm

What's the most productive way to spend the next month? You could write three or four short stories for *The New Yorker*. You could pen maybe half the songs for your next album. You could devote three hours each morning to updating your popular game-related news site. Or you could continue debugging the unit AI for your up-and-coming real-time strategy game, which will put you close to a third of the way toward getting behavior that won't look blatantly stupid to the average Wal-Mart shopper.

Let's face it: game programming is hard. And it's easy to be jealous of those other guys.

That's not to say that writing songs or fiction is easy, but you're not going to see an author throw up his hands and say, "The technical problems involved in writing a story about a mermaid are going to take me six weeks to solve." Or "I've written this brilliant essay, but I'm having the hardest time keeping the characters from constantly walking into each other."

Writing and debugging code in Visual C++ doesn't have nearly the same easy flow as editing and revising paragraphs in Word. Quick: You have thirty seconds to compute the value of "sin(0.43) * 1.8" without using a calculator! (30 second pause.) Odds are that you're sitting in front of hugely powerful computer, and yet you didn't have an easy way to get at the answer. There's no sine key on the Windows calculator. And could you have created a Visual C++ project and written code to display the result in less than thirty seconds? Something isn't at all right with this picture.

Let's jump off into a caffeine-induced fantasy-land for a minute, and pretend we have a magical system that makes programming be fantastically straightfoward.

First, we dispense with our math problem by typing it in directly:

sin(0.43) * 1.8

and we get an answer back right away:

1.227597

Well, shoot, a Commodore 64 could have done that! Let's write some basic vector functions:

magnitude (x,y,z) = sqrt(x*x + y*y + z*z)

dot (x1,y1,y2) (x2,y2,z2) = x1*x2 + y1*y2 + z1*z2

normalize (x,y,z) = (x*m, y*m, z*m) where m = 1.0 / magnitude(x,y,z)

Now let's test these out:

magnitude(10,20,30)=

37.416574

normalize(10,20,30)=

(0.2672612, 0.5345225, 0.8017637)

normalize(1,0,0)=

(1,0,0)

(1,0,0) ‘dot‘ (0,1,0)=

0

We’re already getting interesting results. Obviously, a unit vector is, by definition, already normalized, but it’s nice to be able to quickly verify that. It also looks like a unit vector dotted with an orthogonal unit vector gives a result of zero. We haven’t proven it, but it’s an interesting theory nonetheless. This is amazing! Three lines of "code" and we’re learning properties of vector math.

Even more amazing is that we aren’t in fantasy land at all. This is real code, written in a real programming language (not just a math package) with existing compilers that generate machine code. You can be doing the same thing on your Windows, Linux, or MacOS machine within a few minutes. For free.

What Are the Benefits, Exactly?

Is Haskell - the language used in the preceding example - the holy grail of game programming? Probably not, but it’s only one of a number of languages that have been the result of much research into language theory and implementation over the last twenty years. In the days of 20 MHz 80286-based machines (which you could still buy only eight years ago), heavier languages like Haskell, ML, and Scheme seemed out of reach for everyday work. Similarly, any graphics text written during the first fifteen years following the invention of the z-buffer dismissed the technique as hopelessly memory intensive. With gigahertz processors dotting the horizon, things have changed.

C and C++ are usually cited as typical high-level languages, but they also have a reputation as "systems programming languages" or "high-level assemblers." What is it that makes so-called modern programming languages higher-level than C?

1. **Basic data structures are easier to work with.**

   Notice in the vector example that there isn’t a need to define a vector type. Grouping three values in parentheses is all that’s necessary. Want to associate a difficulty with a level file? "(’castle.lev’, easy)" will do it. Need a binary tree? "(Left, Right)" describes the general "shape" of a tree, where Left and Right can each either be another tree or a leaf value. Things that would have been a pain are suddenly trivial.

2. **The compiler doesn’t need as much hand-holding.**

   In the expression "x + 1.2" there’s no need to declare the type of x, because it’s obvious from context that it must be a floating point value in order to have 1.2 added to it. If you later use x in an inconsistent manner, the compiler will complain. Notation can be a lot more mathematical, instead of being bulked up with excessive punctuation and typing.

3. **Polymorphism.**

Code that knows about the shape of a binary tree can operate on trees containing vectors, trees containing strings, and trees containing integers. This is much like the template feature of C++, except that you don’t need any extra fussing or syntax. One search routine, for example, can be used in a variety of situations. Bottom line: you write less code.

4. **More advanced concepts built-in.**

Imagine passing a point to a routine and having it return a function that moves a creature one step toward that point. This is pretty bizarre sounding, but that's because it isn't in the usual C++ toolbox. "Programming languages teach you not to want what they cannot provide," says Paul Graham in *ANSI Common Lisp*.

5. **Safety.**

   It isn't possible to have wild pointers or to overrun array bounds. You can't accidentally overwrite memory. In Lisp, it isn't even possible to increment a value and have it overflow; you can add 10 billion to 10 billion - without using floating point arithmetic - and get a valid answer.

6. **Interactivity.**

A tremendous amount of work has gone into writing C debuggers, but the seemingly- unachievable goal is to code up a function and then immediately test it interactively. This is the norm in many other programming languages. As a step in this direction, Visual C++'s "edit and continue" feature is a technical marvel, letting you skip a recompile in certain situations. Lisp systems have been able to do this in an unrestricted manner since the 1960s.

The Tale of An Abused Bandicoot

Crack-dot-com's *Abuse*, released in 1995, was one of the first games to look to a higher-level language for much of the gameplay-related code. Jonathan Clark chose to use Lisp as the scripting language, which although one of the oldest programming languages in existence, still has the features to classify it as modern. Says Clark:

> "Being a C/C++ junkie all my life, I quickly found myself fascinated by Lisp. Here was something totally different: a functional language where data and code are the same thing, an environment where experimentation and incremental development was a way of life."

He offers a comparison of some simple C++ code:

```
typedef void (*draw_function_ptr)();

struct my_struct{

color_class *color;

char *name;

draw_function_ptr ptr;

};

my_struct array[4];

void init()

{

array[0].color = new color_class(255,0,0);

array[0].name = "Blue Guy";

array[0].ptr = blue_draw_function;

// etc.

}
```

and the equivalent code in Lisp, taken directly from the *Abuse* source:

```
(setq my_list ‘(

( ,(new-color 0 0 255) "Blue Guy" blue_draw_function)

( ,(new-color 255 0 0) "Red Guy" red_draw_function)))
```

The current games that show the power of a Lisp-like language are Naughty Dog’s *Crash Bandicoot* titles. Andy Gavin, self-admitted Lisp fanatic, developed his own Lisp-like language that could efficiently be compiled to machine code for the PlayStation’s processor. This may sound Herculean, but the compiler was also written in Lisp (on a PC) and the symbolic manipulation needed for compilation is one of Lisp’s strengths. Jonathan Clark also wrote his own Lisp interpreters for *Abuse* and *Golgotha* ("the language is simple enough that I had a fairly complete system in a week," he says).

The *Crash Bandicoot* graphics engine was coded in C, of course, but the Lispish GOOL was used for the sequencing and data handling that defines the playable aspects of the game. C is brilliant for operating on rigidly structured values in an efficient manner, but isn’t nearly as nice when it comes to slinging around data in the loose manner needed for gluing together bits of rendering into something fun. The scripting in *Crash* is about as complex and slick as ever attempted: Crash can run, fly a plane (and shoot down other planes), drive a boat, use a jetpack, swim underwater, and perform endless specialized actions. Oh, and the series has sold over ten million copies. Chalk one up for Lisp.

Beyond Scripting

Scripting is one thing, but can the benefits of languages like Lisp and Dylan (a Lisp-like language with a more familiar syntax) be used for the tricky bits of a 3D engine? Researcher Francois Pessaux was curious as well, so he set out to write a *Doom*-style engine directly in Objective Caml, an ML dialect that’s been in development for fifteen years. Admittedly, 2.5 dimensional graphics are a thing of the past, but there are still a lot of interesting problems involved, including BSP generation and traversal, and a level-editor.

Objective Caml is an interesting language. It feels a lot like Haskell, but the implementers have given efficient compilation a higher priority than most groups doing language research. So, it’s a good way to see how higher-level languages stack up to old standbys, like C. Francois wrote his code twice, once in Objective Caml, and once in C, for just this reason.

In the end, he had 1000 non-blank lines of C and 700 of Objective Caml. The former code ran at 125 frames per second and the latter at 100, both running on a 333 MHz Pentium II. Yes, C technically "won," but the margin was awfully close. And Francois admitted that the Objective Caml version was much easier to write and debug. As a final comment on this test, note that the Objective Caml compiler was at version 1.0x when Francois wrote his paper. It’s been stepped a full version and greatly improved since then.

The general goal has been to get more elegant languages to within a factor of two of C. Once inside that range, the next processor generation or some general tweaking is enough to close the gap. Being able to interactively diddle with data structures and algorithms might be more than enough to make up for a measured speed hit in toy benchmarks.

Yes, but Are You Really Serious About All This?

The real question is when any of this language research is going to finally make its way into a full-fledged game. The concepts have been proven, the productivity gains are real, but it’s easy to get deluded into thinking that better technology always wins (something that whoever ported *Resident Evil* from the PlayStation to the Game Boy probably thought about on a daily basis). What’s been keeping Objective Caml and Haskell and ML and Dylan and even good ol’ Lisp out of the mainstream?

First, it can take a significant amount of time to unwarp your C-riddled mind enough to be able to wrap it around some very different concepts. Think about how strange some things seemed when you were first exposed to them in Computer Science 101. Even linked lists might have seemed a bit heady back then. When you start reading about currying and lazy lists and finding out that variables now have a reputation similar to that of the GOTO, you’ll be back in that dizzy state of mind.

Then there’s Java. Java has been hyped as a simpler alternative to C++. And it certainly is, except that it’s still the same sort of language with the same general features, style, and shortcomings. It’s close enough to C++ in principle that the performance difference is more striking than that between C and Lisp. In Lisp, you’re willing to trade a few performance points for the interactivity and language benefits. In Java, there's this nagging thought that you should just write in C++ and be done with it. Even so, Java has taken over as the easy and portable language.

Finally, C can live in a vacuum, but most of the languages I've mentioned can't. You can write the game engine in C. You can write the level editor in C. You can use C for scripting. But while ML is great for many things, it doesn't let you do the bit-level mucking about that's needed to write a good LZW compressor or to manipulate a 16-bit frame buffer. You need to write at least small bits of C code that can be hooked into the language of your choice. The bottom line is that you can learn C++ and be done with it, but you can't do the same with most of the languages I've mentioned. You still need to dip into the lower-level bit bucket every once in a while.

Back when most games were programmed entirely in assembly language (and even 1993's hugely popular and oft-ported *NBA Jam* was written in assembly), there were endless debates about how necessary or pointless this was. Over the years, the former point of view has been greatly eroded, leaving only the occasional misguided newbie - Game Boy programmers excepted. Modern languages are still at the beginnings of these debates. "They're too slow! C++ is the only way to go!" might be the current battle cry, but it's sounding awfully familiar?

References

Graham, Paul. *ANSI Common Lisp*. New Jersey: Prentice-Hall, 1996.

Pessaux, Francois. "OCamlDoom: ML for 3D Action Games," 1998.

The Haskell web site: http://www.haskell.org

James Hague (jhague@dadgum.com) became addicted to wacky, alternative viewpoints after starting his own Mac-only software company in 1996. He's currently very happy doing programming for Volition's upcoming Summoner RPG.