Game Scripting in Python

Bruce Dawson
Humongous Entertainment
bruced@humongous.com
http://www.cygnus-software.com/papers/

Introduction

Scripting languages allow rapid development of game behaviour without the pitfalls that await the unwary C++ programmer. Using an existing scripting language saves the time and cost of developing a custom language, and typically gives you a far more powerful language than you could create on your own.

Python is an excellent choice for a game scripting language because it is powerful, easily embedded, can seamlessly be extended with C/C++ code, has most of the advanced features that make scripting languages worthwhile, and can also be used for automating production. Additionally, the books, development tools and libraries available for Python provide great opportunities for benefiting from the work of others.

This session describes our experiences at Humongous Entertainment with integrating Python into our new game engine. It explains why we chose Python, the benefits we've seen, the problems we've encountered, and how we solved them.

Why use a scripting language?

C++ is a powerful language and an enormous improvement over C. However it is not the best choice for all tasks. The emphasis in C++ is run-time performance [Stroustrup94] and no feature will make it into the language if it makes programs run slower. Thus, C++ programmers are burdened with many restrictions and inconveniences.

A few of the restrictions – that C++ programmers are so used to they usually don't notice – include:

- Manual memory management: a huge amount of C++ programmer time is spent making sure that delete gets called at the appropriate time.
- Link phase: C++ modules are linked together so that function addresses don't need to be resolved at run-time. This improves run-time performance, but slows down the edit/test cycle.
- Lack of introspection: C++ code has way of knowing what members are in a class. This makes writing code to load and save objects an enormous task that in some scripting languages is a single call to a built-in function.

C++ is static, scripting languages are dynamic. Grossly oversimplified this means that C++ runs fast, but scripting languages let you code faster.

Thus, C++ should only be used when you need to optimize for run-time performance. On today's fast computers there are huge blocks of code where performance is not an issue. If you develop these in C++ when a scripting language would do then you are optimizing for the wrong thing.

What's wrong with SCUMM?

Humongous has used SCUMM (Script Creation Utility for Maniac Mansion) to create over 50 different games. SCUMM is a powerful adventure game creation language, but it has some limitations. SCUMM was written more than a decade ago and it is missing some of the features found in more modern languages.

Despite continual patching and maintenance, SCUMM will never be as full featured and robust as the many other languages available.

Why Python?

We explored the idea of creating a new, more modern proprietary language, but wisely discarded the idea. We are in the business of making games, not languages.

After spending millions of dollars over the years maintaining a suite of proprietary tools we, really wanted to go with an existing scripting language rather than creating one. Existing languages let us get started faster, have lower maintenance costs, are generally better than a language we would create, and tend to get better over the years, even if we don't work on them.

Once we decided that we wanted to use an existing scripting language we had to choose one. We needed something clean that would support object oriented programming and could be embedded in our games, without technical or licensing problems.

We considered Lua [Lua01] and Python [Python02], both of which had been used in several games.

Lua is smaller and probably easier to embed in an application, and has some nice language constructs. However, at the time we were choosing a language the documentation was a bit sketchy. This is probably because Lua is a newer language than Python.

Python has more extension modules than Lua, more books written about it, and stackless Python [Tismer01], which was a promising way of creating micro-threads for object AI. We ended up not using the stackless version of Python, but we did start using Python for writing automated build scripts and this gave us momentum. We knew the syntax, we liked it, and so we chose Python.

Both languages have been improved since we made our decision – Lua has become stackless and Python has acquired generators, which allow some similar functionality. Either language is a safe choice.

Who is using Python in games

Python has been used in a number of games, including:

- ToonTown - http://www.toontown.com/
- EveOnline - http://www.eve-online.com/
- Blade of Darkness - http://www.codemastersusa.com/blade/

with numerous other sightings that are difficult to verify, including at least one PS2 game.

Python is also used in at least two game engines:

- Game Blender - http://www.blender.nl/gameBlenderDoc/python.html
- PyGame - http://www.pygame.org/

A build script example

As an example of Python code, here is a simple build script that recursively builds all VC++ workspaces. Written in a fairly verbose fashion it still takes just a dozen lines of code:

```
import os

def BuildAllWalker(unused, dirname, nameList):
    for entry in nameList:
        if entry.endswith(".dsw"):
            workspaceName = os.path.join(dirname, entry)
            resultFile = os.popen("msdev %s /make all" % workspaceName)
            resultLines = resultFile.readlines()
            for line in resultLines:
                print line

if __name__ == '__main__':
    os.path.walk(os.curdir, BuildAllWalker, None)
```

With a couple of dozen more lines of code, this script [Dawson02] parses the output and e-mails a summary report to everyone on the team. Unlike some scripting languages, the code is quite readable. Using the same language for build scripts and game scripting will save a lot of learning time.

This build script example shows what is probably the most distressing aspect of Python to new users. Flow control is indicated entirely by indenting. There are no begin/end statements and no braces.

It took me about five minutes to adjust to this idea and I have since found that it is very effective. I've been in more than one heated argument about where the braces in C/C++ should go and I'm sure that Python programmers are 10% more productive just because of the time they don't spend arguing about K&R indenting style versus others. When your code blocks are defined by indenting there is no longer any possibility of the indenting not matching the compilers view of the code's structure.

The one time when indenting for flow control causes problems is when tabs and spaces are mixed. Since most programmers use three or four space tabs and the Python compiler internally uses eight space tabs, mixing tabs and spaces causes confusing syntax errors. If you standardize on either spaces or tabs and use an IDE that warns you when you are mixing them then you will have no problems.

Game script example

The following is a sample of some Python code from our first Python/C++ game. You can see from this small sample that the Python code is running the main loop, calling out to modules that may be written in either language:

```
try:
	# Update the input
	g_InputManager.DispatchInput()

	# Tick the menu code
	g_MenuManager.Tick()

	# Tick the scene code
	g_SceneManager.Tick()
	g_SceneManager.SetScene()

except:
	import traceback
	traceback.print_exc(None,sys.stderr)
	g_Clock.Stop()
#-endtry
```

Thus, our game boots up in Python, and only calls out to C++ as needed.

How does it work?

Python is based on modules. When one source file wants to use functions defined in another source file it imports that file. For instance, if *gameai.py* has a function called UpdateAI then another Python source file can go:

```
import gameai
gameai.UpdateAI()
```

The wonderful thing about this from a game programmer's perspective is that if UpdateAI() runs too slowly it can be rewritten in C++. To do this, the functions and types defined in *gameai.py* are written in C++ and registered with Python with the same module name. The Python code can continue to import and use *gameai* without *any* changes.

Thus, Python modules make it easy to prototype your entire game in Python and then recode as needed in C++.

Glue code

Writing the glue code to make C++ code visible to Python is tedious if you try to do it by hand. A system for generating glue code is essential.

Swig, Boost, CXX and others [Abrahams01] are systems to help generate code to glue together Python and C++. Another option is Fubi [Bilas01] which is a generic system of making C++ functions and classes available to a scripting language.

Most of these glue code systems work by parsing a C++ header file. Thus they are restricted to what a C++ header can expose, and some of them did not support having Python classes derive from C++ classes. They have been improved substantially since then and are worth taking a look at.

We decided to create a custom solution that generates glue code based on an IDL description of the class or function to be exported. Our code name for this is Yaga, which is a recursive acronym that means Yaga is A Game Architecture.

A typical snippet of Yaga IDL code looks like this:

```
module yagagraphics
{
	struct Rect
	{
		int32_t x;
		int32_t y;
		int32_t width;
		int32_t height;
	};
```

which then generates the following glue code, and a few hundred lines more:

```
static int YAGAPYCALL Rect_init(
		YagaPyStruct<Rect>* self,
		PyObject* args,
		PyObject* kwds)
{
	if (!PyArg_ParseTuple(args, "|llll",
			&self->structData.x,
			&self->structData.y,
			&self->structData.width,
```

```
                &self->structData.height))
        {
                return -1;
        }

        return 0;
}
```

This system makes it very easy to export classes and functions, derive Python classes from C++ classes, map C++ arrays and vectors to Python sequences, and much more.

Memory allocations

In Python every thing is an object, and all objects are allocated. Because all objects are reference counted you don't have to worry about freeing the memory, but if you're writing a game – especially a console game – you have to worry about where all that memory is coming from, and how badly those allocations are fragmenting your memory.

To control this problem you need to isolate Python into its own memory arena. You need to redirect all allocations to a custom memory allocator that stays inside a fixed size arena. As long as you leave enough of a buffer above the maximum Python memory footprint you should be able to avoid fragmentation problems.

The other memory problem is unfreed blocks. This is mostly not a problem with Python because every object is reference counted. When variables go out of scope or are explicitly deleted, the reference count of the object is decremented. When it hits zero the object is freed, and life is good.

You can end up with situations where one forgotten variable is holding onto a chain of many other objects, preventing their deletion, so you have to be a little bit careful about cleaning things up. However the more serious problem is circular references. If object A owns object B, but object B has a callback pointer to object A, then neither object will ever get deleted. The Python authors realized this problem and in a recent version of Python they added a garbage collector. This sweeps through all objects, looking for unreachable objects and cleaning them up.

Garbage collectors are a problem for games because they can run at unpredictable times and take long enough to make you miss your frame rate. Thus, game programmers will want to disable the garbage collector – which is easily done – and then explicitly run it at the end of each level. The garbage collector can also give reports on how many unreachable objects were still allocated, which can help you track down your circular references, so you can manually unhook them. This is the Python equivalent of memory leak detection.

Performance

If you write Python code to do some heavy floating point work and then compare the results to C++ you will be disappointed. Python is a slower language. Every variable reference is a hash table lookup, and so is every function call. This will never give performance that can compete against C++, where the locations of variables and functions are decided at compile time.

However this does not mean that Python is not suitable for game programming, it just means that you have to use it appropriately. If you are doing string manipulations or working with STL sets and maps in C++, then Python code may actually be faster. The Python string manipulation functions are written in C, and the reference counted object model for Python avoids some of the string copying that can occur with the C++ string class. Sets and maps are O(log n) for most operations, whereas Python's hash tables are O(1).

You definitely don't want to write scene graph traversal or BSP collision detection code in Python. However if you write that code in C++ and expose it to Python, then you can write your AI code faster than ever.

Remember the old 90/10 rule, and make sure that you aren't sweating about run-time performance in the 90% of the code where expressiveness and fast coding are what really matters.

Consoles

Memory and performance issues are always critical on console machines. Keeping the Python memory allocations contained in arenas is absolutely critical when you don't have virtual memory to save you from careless memory allocations, as is using the garbage collector wisely.

Consoles don't have keyboards, mice and multiple monitors, so running a Python debugger on a console machine does not work well. Remote debugging is essential if you want to be able to tell what is happening in your Python code. Fortunately, remote debugging is easy to set up with the freely available HapDebugger [Josephson02].

Python is written in C and has been ported to many compilers and platform, including PDAs. Thus, it would be surprising if porting Python to console machines took substantial effort.

Python will end up wasting a bit of memory for features that are not relevant on console games, but probably not enough to worry about on the new generation of console machines, with their minimum of 24Mbytes of memory.

Legal wrangling

Building our company's future on a new language is a significant decision and I thought it best to get a blessing from our lawyers before proceeding. This may have been a mistake.

Lawyers know much about the law. However, they generally don't know much about programming. Since most programmers don't ask the company lawyers before incorporating open source code, when you do ask them they tend to think that you are asking for something strange and outlandish. Their immediate reaction was that this was a risky and unwarranted plan.

And it turns out they have a point. If you talk to a good intellectual property lawyer for a few hours they can fill you in on all the ways you can get burned by using open source software. There are precedents for patented or copyrighted material appearing in "freely distributable" code and this has the potential to cause significant legal problems. When you license code from a vendor they can indemnify you against such liability, but with open source software there is no one to license it from.

However the open source community is vigilant about intellectual property law. For example, the Independent JPEG Group removed LZW and arithmetic coding algorithms from their library to avoid patent problems [IJG]. Responsible programmers pay attention to licensing issues and are usually familiar with the GPL and LPGL [FSF01] and their differences.

There are risks associated with incorporating open source code into commercial products. These risks should be taken seriously, but they should not prevent the use of open source. There are many open source libraries used in game development and I know of no reason why Python should not be one of them.

Disadvantages

Multi-language development adds an extra layer of complexity. It's difficult to debug in both languages simultaneously, and time must be spent on maintaining the glue code that ties the languages together.

Dynamic languages like Python have no 'compile time' type checking. This is scary at first, but it really just means that you have a different set of run-time problems to worry about than with C++, and they are generally much easier to deal with.

Different end of line conventions

UNIX, Mac OS and Windows have different conventions for how to terminate lines in text files. This is unfortunate.

C/C++ libraries on Windows routinely do line ending translations so that UNIX files can be read on Windows, by making Windows files look like UNIX files. This seems to be less common on UNIX and Macintosh – presumably with the bizarre assumption that all files on those platforms will have been created on those platforms. This assumption fails miserably in today's networked world, and Python falls victim to it. Until very recently Python code written on Windows would not compile on Macintosh, and vice versa.

Clearly this problem can be fixed by running the files through a filter (written in Python?) prior to running them, or by distributing them in compiled byte code form. However both solutions have problems. It would be nice if the computer industry could standardize on a file format for *text*, or teach all the file IO libraries how to read each other's files.

This problem is particularly amusing on Apple's OS X where, depending on which mode you are in, you are either running UNIX or Macintosh programs. This gives you two line ending conventions on one machine, without even rebooting.

The Macintosh version of Python has recently fixed this problem by checking the line endings when it opens a file and adjusting on a per-file basis. Standardizing on UNIX line endings may therefore be a viable solution that works on all platforms, but be aware of the problem.

Debugger problems

Many Python programmers believe that automated testing and print statements are the only debugging tool they need. They tend to believe that debuggers lead you down counter-productive paths. This may be true for them, but I have grown accustomed to source level debugging and I will not give it up lightly.

PythonWin is (surprise) a Python debugger and IDE running on the Windows platform. It's free, it has some nice features, but it also has some annoying quirks. Looking past the quirks there are the serious problems that it only runs on Windows, and can't debug Python programs that have their own message pumps.

At Humongous Entertainment we develop all of our games for both Macintosh and Windows, and we are moving into console development as well. We really need one debugger that can handle all three platforms. The elegant solution to this need is a remote debugger. It turns out that the Python architecture makes writing a debugger very easy. By leveraging that and some other free components we were able to make a Python debugger that I feel is better than PythonWin for Windows debugging, with remote debugging as an extra bonus. The client that runs on the debuggee is just a few thousand lines of code. The interface is ASCII text over sockets, so we don't anticipate any problems porting the debugger client to many other platforms.

Because we want to concentrate on making games, not language tools, we decide to leverage open source even more. We released the Humongous Addition to Python – the HAP Debugger – to the Python community as an open source project [Josephson02]. It was wonderful to have an opportunity to give back to the community, and we look forward to getting free maintenance out of the deal.

Another debugger problem that we have not yet resolved is performance. Most compiled language debuggers implement breakpoints by replacing regular instructions with instructions that will cause a CPU exception – int 3 on x86 processors. This allows the program to execute at full speed until it hits a breakpoint. Python doesn't support restarting from exceptions, so a

breakpoint exception doesn't work. Instead, Python debuggers handle breakpoints by single stepping through the code, constantly asking themselves "is there a breakpoint on this line?"

The performance implications can be serious. For now we have minimized this problem by making sure our developers have machines that are far faster than our target machines. That, combined with putting all of the heavy lifting inside C++ extensions means that even if the Python code runs a hundred times slower under the debugger, it doesn't affect the overall game speed to a fatal degree. This is a fixable problem, but not one that the main Python developers are currently interested in fixing.

Security and cheating

C++ programmers sometimes joke about optimizing their code by deleting comments or making variable names shorter. In Python it might actually work.

Python is compiled to byte code which is cached for later runs, so deleting comments shouldn't have an effect, but shorter variable names might. As in most scripting languages the variables are looked up by name at run time. This is part of what makes scripting languages so powerful, since it removes a lot of the restrictions imposed by the compile time binding of C++. However it does mean that your variable names ship with your code.

This takes some getting used to. Shipping scatological variable names with children's games could cause a bit of a scandal. However the more serious problem is with multi-player games that use Python for scripting on a PC. Cheaters have already proved themselves willing and able to decompile C++ in order to hack it. Decompiling Python gives you variable and function names, which will make hacking much easier.

Advantages

Programming in Python is fun. It's easy to learn, it makes you more productive, and it makes you think about programming differently. Learning Python has made me a better C++ programmer.

Happy programmers who are learning faster are going to be more productive, and they're going to create better games. The team at Humongous that is using Python for game programming has the highest morale in the company.

Productivity is higher with the Python game programming system, even though development is still being done on it. It is already clear that we will save a lot of money from this switch.

User interfaces, which always seem to take longer than they should in C++, can be done in some creative new ways in Python. It is convenient to define menus with a text file that defines the location of GUI elements and the graphical resources associated with them. With C++ you then have to hook up these GUI elements by writing code to explicitly associate functions and objects with controls. With Python you can put the names of the functions and objects in the

text file, and look these up at run time. The dynamic and introspective nature of Python makes this the natural thing to do.

Many of the limitations of Python that we worried about when we first started working with the language have gone away. The Python developers have continued to improve the language and sometimes it seems like they are working on our list of feature requests first.

Loading and saving

C++ programmers spend a lot of their time solving hard problems that don't exist in scripting languages. For instance, the problem of saving and restoring game state is a messy problem in C++, which usually involves thousands of lines of code. The solutions frequently result in save game files that only work with one version of the program. In Python the cPickle module makes this a non-issue by saving and restoring arbitrarily complex data structures.

In the example below we first declare our mainObject. Normally this would be a user defined class that contained handles to all of the state to be saved, but for simplicity I am using just a list. Initially the list contains the number zero and a string. Then I initialize the first element of the list with another list. This could continue to contain an arbitrarily complicated nest of objects, including circular references.

```
mainObject = [0, "A string"]
mainObject[0] = ["a string", 0.234, 10, 12341234123412341234]
```

Then I save the main object. This requires two lines of code. One line to import the cPickle module, the other to open a file and save the object in binary format. Text format is a useful option for during development – just omit the final parameter to dump().

```
import cPickle
cPickle.dump(mainObject, open("game.sav", "wb"), 1)
```

Then it's time to reload the data. Again this takes two lines of code. One line to import the cPickle module, and the other to recreate the mainObject and all of the objects, lists, member variables and other data inside of it. The third line of code prints the object and you can see the properly restored nested lists.

```
import cPickle
mainObject = cPickle.load(open("game.sav"))
print mainObject
[['a string', 0.23400000000000001, 10, 12341234123412341234L], 'A string']
```

I don't think you can overestimate the importance of this feature, which is fundamentally unavailable in C++.

Generators – micro-threads for game AI

Micro-threads allow greatly simplified AI and object update code [Carter01] by moving much of the object state information into local variables, where it naturally belongs. There are assembly language tricks you can use to hack micro-threads into C++, but this can get messy. In recent versions of Python these micro-threads are built right into the language. They work perfectly.

In Python they are called ‘generators’ because one use of them is to write functions that generate one result and then yield back to the main program. The main program can then resume them later on, and they resume where they left off, with local variables intact. The sample code below shows an object that is created and then moves itself across the screen. While this is a simple case that doesn’t really benefit from micro-threads/generators, it shows the basics of how they can be used to simplify AI and object update code.

```
# Tell the compiler that this code knows about generators and the keyword 'yield'
from __future__ import generators

# Define a class that will define a character's behaviour.
class MovingObject:
      # The constructor creates a handle to the generator
      # function and stores it in a private member variable.
      def __init__(self):
            self.__genHandle = self.__UpdateFunction()
      # Update() is called once per frame and tells the update function to resume.
      def Update(self):
            return self.__genHandle.next()
      # The private object update function is resumed once per frame by Update().
      #It stores all of its state in local variables. Some of it could be stored in
      # member variables if desired.
      def __UpdateFunction(self):
            x, y = 0, 0
            dx = 0.25
            dy = 0.75
            while 1:    # Loop forever
                  # Print the current location.
                  print x, y
                  # Return the current location - in case anybody needs it.
                  yield (x, y)
                  # The function resumes here. Update the location.
                  x += dx
                  y += dy
                  # Update the velocity
                  dx += 0.1
                  dy += 0.5

# This creates an AI object that creates a generator object.
character = MovingObject()
# Run the generator object the first time - local variables
# are initialized, printed, then it yields back to here.
character.Update()
# Run the generator subsequent times. The character moves,
# prints its location, then yields, repeating as needed.
for x in range(10):
      character.Update()
```

Even if you choose not to use generators, implementing an AI update method in Python is cleaner than in C++. This is because if one part of your AI code needs some extra temporary state, it can add it to the object, and then delete it when it is no longer needed. C++'s static nature does not allow adding new member variables at run time, which forces your objects to contain all the state that they will ever need, at all times.

Getting started with Python

If you want to get started with Python the first thing to do is to visit the Python web site at http://www.python.org/ and download the version for your platform.

The documentation is available at http://www.python.org/doc/current/download.html, with the compiled HTML being the easiest version to search.

For Windows development you should get PythonWin and various Win32 extensions from http://users.bigpond.net.au/mhammond/win32all-142.exe

Everybody should get the HapDebugger from https://sourceforge.net/projects/hapdebugger/.

And, of course, you'll need to download and build the Python source, which allows you to build debug and release versions yourself. The Python 2.2 source can be downloaded from: ftp://ftp.python.org/pub/python/2.2/Python-2.2.tgz

Finally, you can read about the inner details of creating Python extensions by hand http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/66509, and then choose one of the glue code packages that will do it for you, from http://www.boost.org/libs/python/doc/comparisons.html.

References

[Stroustrup94] Stroustrup, Bjarne "The Design and Evolution of C++", Addison Wesley, 115

[Lua01] "The Programming Language Lua"
http://www.lua.org/

[Python02] Python Language Website
http://www.python.org/

[Tismer01] Tismer, Christian "Stackless Python"
http://www.stackless.com/

[Dawson02] Dawson, Bruce "Python Scripts"
ftp://ftp.cygnus-software.com/pub/pythonscripts.zip

[Abrahams01] Abrahams, David, "Comparisons with Other Systems"
http://www.boost.org/libs/python/doc/comparisons.html

[Bilas01] Bilas, Scott, "FuBi: Automatic Function Exporting for Scripting and Networking"
http://www.gdconf.com/archives/proceedings/2001/bilas.doc

[IJG] http://www.ijg.org/ - docs\README in the source distribution

[FSF01] “What is Copyleft?”
http://www.gnu.org/copyleft/
[Josephson02] Josephson, Neal “HAP Python Remote Debugger”
http://sourceforge.net/projects/hapdebugger/
[Carter01] Carter, Simon “Managing AI with Micro-Threads”, Game Programming Gems II, Charles River Media, 265-272

About the author

Bruce is the director of technology at Humongous Entertainment, which means he gets to work on all the fun and challenging tasks that nobody else has time to do. He also teaches part-time at DigiPen. Prior to Humongous Entertainment he worked at Cavedog Entertainment, assisting various product teams.

Bruce worked for several years at the Elastic Reality branch of Avid Technology, writing special effects software and video editing plug-ins, and before that worked at Electronic Arts Canada, back when it was called Distinctive Software. There he wrote his first computer games, for the Commodore Amiga, back when thirty-two colours was state of the art.

Bruce presented a paper at GDC 2001 called "What Happened to my Colours?!?" about the quirks of NTSC.

He is currently trying to perfect the ultimate Python script that will automate all of his job duties - so that he can spend more time playing with the console machine he won at a poker game.

Bruce lives with his wonderful wife and two exceptional children near Seattle, Washington where he tries to convince his coworkers that all computer programmers should juggle, unicycle and balance on a tight wire.