

GameMonkey Script FAQ

What is GameMonkey Script?

GameMonkey is a scripting language that is intended for use in game and tool applications. GameMonkey is however suitable for use in any project requiring simple scripting support. GameMonkey Script is usually just referred to as *GameMonkey* and abbreviated to *GM* (gee-em).

GameMonkey borrows concepts from Lua (www.lua.org), but uses syntax similar to C, making it more accessible to game programmers. GameMonkey also natively supports multithreading and the concept of states.

What does the code look like?

Here is some sample script code that calls functions bound from C:

```
OnDoorTriggerEnter = function(door, objEntering)
{
    if(objEntering == player && !door.IsOpen())
    {
        door.Open();
        return true;
    }
    return false;
};
```

Here is some C code that is bound by script:

```
#include "gmThread.h"
#include "gmMachine.h"

//
// float SquareRoot(int/float)
//
int GM_CDECL SquareRoot(gmThread * a_thread)
{
    float fValue;
    GM_CHECK_NUM_PARAMS(1);

    if(a_thread->ParamType(0) == GM_INT)
    {
        fValue = (float)a_thread->Param(0).m_value.m_int;
    }
    else if(a_thread->ParamType(0) == GM_FLOAT)
    {
        fValue = a_thread->Param(0).m_value.m_float;
    }
    else
    {
        return GM_EXCEPTION;
    }

    a_thread->PushFloat(sqrtf(fValue));
    return GM_OK;
}
```

```
// Before using, somewhere in code, register the binding(s):
extern gmMachine machine;    // Virtual Machine instance

gmFunctionEntry mathLibrary[] =
{
    {"SquareRoot", SquareRoot},
};

machine->RegisterLibrary(mathLibrary, sizeof(mathLibrary) / sizeof
(mathLibrary[0]));
```

Here is some C code calling a script function:

```
#include "gmMachine.h"
#include "gmcallsript.h"    // Header contains helper class

extern gmMachine machine;    // Virtual Machine instance

// Assumes a function int Add(int, int) exists in script
int AddTwoIntegers(int valueA, int valueB)
{
    int resultInt = 0;

    gmCallScript::SetMachine(&machine);

    if(gmCallScript::BeginGlobal("Add"))
    {
        gmCallScript::SetReturnInt(resultInt);
        gmCallScript::AddParamInt(valueA);
        gmCallScript::AddParamInt(valueB);
        gmCallScript::End();
    }

    return resultInt;
}
```

What are the key features of GM?

- Small code base. Compiled code may use about 50kb of RAM. Less when tweaking or sharing with application.
- Compile source code at run time, or link to precompiled libs.
- Lightweight, native threading.
- Soft real-time incremental garbage collection. Controllable memory footprint. No painful reference counting.
- Easy to bind C\C++ functions and call script from C\C++.
- Runtime debugging and reflexion support.
- C style syntax.
- Competitive performance when compared to other scripting languages for both CPU and Memory usage. Speed is a trade off for flexibility and simplicity.
- Easily modifiable as it is written in C++ and uses Flex and Bison.

Here are some things that gm does *not* try to be:

- A language for non-programmers. However, when provided with a simple set of bindings, non programmers may be able to configure an application, or define simple behavior.
- A safe language suitable for processes critical to human life. If you are programming the safety controls for a nuclear reactor, an automatic drug administer for medical patients, or a in-flight navigation system, consider other, more fail-safe languages.

What platforms does GameMonkey run on?

Written entirely in C++, it should run on any platform with at most minor modification or configuration. It has been successfully compiled and run on: Windows PC, Apple Mac, Microsoft Xbox, Playstation2.

Where did GameMonkey come from?

During 2002, Matt and Greg investigated scripting languages because they seemed like the 'right thing to do' as far as improving the efficiency of game and tool development. We read postmortems, reviews, users opinions, case studies, GDC notes and evaluated languages like Java, interpreted C, Python and a bunch of wacky scripts used by various games, when source code or examples were present on the Internet. One tiny embedded language stood out, and that was Lua. Matt used it to bind some of his own tools and utilities and quite enjoyed the experience. We considered just using Lua, but even at version 4 (the current release at the time), there were many improvements we thought could be made. Such as native threading, robust parser generated by flex & bison (lex & yacc), the concept of 'states' (which turned out to be a simple binding), as well as things for C programmers like C syntax and base 0 numbers. So in not much time at all, we had a new scripting language, based on the concepts of Lua in a somewhat working form. About that time, a new console project was starting at work and we decided to use the script for both the game code and tools. This meant finishing it off during work time and it thus become part of the company code base. Some months later, the project was canceled (as happens regularly in the game industry), and Matt left to work elsewhere. We requested that the code be released to the community so it could be enjoyed by others, and that development could continue. On June 12th 2003, Auran granted that request on the condition that the source code and related materials be released under a 'free software' or 'open source' style license agreement of our choosing, and that we did not sell or profit from the original code. We would like to thank Auran for releasing this material and for the understanding and support shown by its directors. It can also be noted that Auran has an excellent policy to encourage its employees to self-develop and share knowledge as long as this does not conflict with its business, in relation to trade secrets and intellectual property. You can visit Auran here:

<http://www.auran.com>

Why is it called 'GameMonkey'?

We wanted a unique name that related to game programming, perhaps related to nature, sounded cute, cool or whatever. After tossing a few names around, we decided on 'GameMonkey'. There could be other subconscious, psychological reasons, but that pretty much sums it up.

What do you get with the current release of GameMonkey?

- The GameMonkey language including some useful bindings.
- An example GameMonkey executable that runs scripts from the command line.
- A librarian that display the contents of precompiled libs.
- A example debugger that demonstrates basic stepping, watching and breaking.
- Sample scripts.

What is coming in the future?

- A new virtual machine that is a register stack machine and should be up to two times faster than the current VM.
- An IDE with Debugger.
- More bindings for useful tools and applications.
- Possibly native support for enums or constants.

What can I do to help?

- Use GM in your projects.
- Create bindings for GM and share them.
- Tell others about GM.
- Make improvements to GM or its related materials such as debugger, documents, samples, etc. and share them.
- There may be other things such as server space or download mirrors your could provide, so contact us if you think you can help.

Third party code used:

- Scintilla is used by the debugger. <http://www.scintilla.org>.
- Flex and Bison were used by the compiler. <http://www.gnu.org/software/flex> and <http://www.gnu.org/software/bison/bison>.

Design decision notes:

No increment and decrement operators.

The ++ and -- operators familiar to C programmers were removed for the sake of consistency with the language. GM does not allow assignment in an expression. For example:

```
if ( (a = b + c) > 0 ) {}    // Fine for C/C++, not for GM
a = b + c; if( a > 0) {}    // Fine for both
```

Please note that operators +=, *=, &= etc. are provided, allowing for shorter and simpler statements.

The language is case sensitive.

Although case insensitivity may be desirable, if only to prevent errors where similarly named variables clash, it is impossible to implement in an efficient manner. The problem lies in the very powerful 'table'. Tables are the building blocks for logical objects in the language as well as general purpose containers. A table member may be a string like "Hello World" which requires case preservation, or it may be a variable or function which would not require case preservation. Unfortunately converting or checking for case would waste precious CPU time and is thus not implemented. Of course nothing prevents this functionality from being added. The current source code may even have a #define with the beginnings of such a modification.

Statements end in semicolons.

Since Bison only supports one token look ahead, it is very difficult to remove the need for a terminating token such as the semicolon. If a non 'C' style syntax were to be used, it may work without any semicolons. Here is an example of such code:

```
if (condition) then
    DoSomething()
else
    DoSomethingDifferent()
endif

function FunctionVar(param1, param2)
    SomeCode()
endFunction
```

Experienced C programmers may often forget to put a semicolon on the end of a function, because unlike C, functions in GM are just another variable assignment, thus a statement requiring a semicolon. This example may demonstrate the reason:

```
Add = function( a, b ) { return a + b; };
```

Limited automatic type and conversion casting.

In the interest of speed and flexibility, types like *int* and *float* are not automatically converted by bound functions. Please note that they are automatically upgraded as necessary within script statements. If a bound function were to accept a number, where that number may be an *int* or *float*, it must explicitly check for each type and convert as necessary. Helper functions and macros are provided to simplify this operation.

What is in the download package?

gmsrc\bin	Compiled binaries
\doc	Documents
\EditorHighlighters	Editor syntax highlighters
\scripts	Sample script programs
\src\binds	Sample bindings
\examples	Sample application programs
\gm	GameMonkey Script source code
\gmd	Sample Debugger
\gme	Sample script executable
\gml	GameMonkey librarian
\platform	Platform configuration headers

The source code folders contain *project* and *workspace* files for MS Visual Studio 6.

Your first GameMonkey program

In the tradition of programming, we must present... HelloWorld.

- 1) You have already unzipped the package to you hard disk, so you can just skip step 1.
- 2) Create a text file called 'test.gm'
- 3) Type 'print("HelloWorld");'
- 4) Save it to c:\gmsrc\bin
- 5) Open a Command Prompt in c:\gmsrc\bin
- 6) Type 'gme test.gm'

Getting started

Don't have a favorite editor, or application to embed the language, but want to get started?

Download a free text editor. Here are some good ones:

ConTEXT: <http://fixedsys.com/context/>

CrimsonEditor: <http://www.crimsoneditor.com/>

This example will use CrimsonEditor.

- 1) Copy the appropriate syntax highlighter files into your program directory in the appropriate place. Syntax highlighting makes you think you have a cool development environment.
- 2) Set your 'OpenWith' for '.gm' files. Right click on a .gm file in the \scripts folder and choose 'OpenWith'. Now browse for your editor program and remember to check the 'Always use the selected program to open this kind of file'.
- 3) Set up a tool to run the compiler and execute the program.
For Crimson Editor, you would do this:
Tool(menu)->Preferences(menu)
Tools->UserTools
Scroll the UserTools to find 'Ctrl+0' and fill in
Menu Text: Build GM

Command: C:\gmsrc\bin\gme.exe

Argument: \$(FileName) -d -k -e

InitialDir: \$(FileDir)

HotKey: (leave as None, unless you want a hot key other than Ctrl+0)

4) Now you can use the Tool menu, or short cut key to compile and run the file you are editing.

The gme.exe should output error messages in a standard format so that some editors will let you jump to the error line.

The '-k' command line option will wait for you to press 'Enter' after the program has finished, so the console window will remain and not quickly disappear.

Now for a debugging session.

1) Run 'gmd.exe' and a debugger application will appear.

2) Run your program with the -d command line parameter, and the debugger will attach to your running program. The debugger controls should look familiar. You can stop all threads, step through the code and resume. You can also watch some variables. The current debugger is just an example application and is not fully featured.

GME (GameMonkey Script Executer)

The gme.exe program is just an example application that uses GameMonkey script and binds some functions to write various simple console programs. You could write a console game or a tool with it, as several sample scripts demonstrate. The real power is using it in your own applications to extend, configure, or do most of what general purpose languages can.

Since you will probably start out by using GME to experiment with the language, and maybe even use it to run tool scripts, like a .BAT or .CMD file on steroids, here is a description of the syntax:

`gme.exe <script file> <gme commands> <script file commands>`

<script file> - The script file to execute, most likely ending in '.gm'.

<gme commands> - A combination of:

-k Keypress on exit.

-d Allow debugger to attach.

-e Add Windows environment variables to global table called 'env'.

-ke Keypress on exit, only if there was a compile or run-time error.

<script file commands> - Any parameters you wish to pass to the script. The parameters will be placed in a global table called 'arg'.

Your first embedded virtual machine:

Embedding GameMonkey in your own applications is as simple as compiling the gm source files with your application code, creating a gmMachine and executing some script with it. Please look at the GME source code for example that includes extra bindings.

To embed GameMonkey, you usually follow these steps:

- 1) Add all the **src\gm *.cpp** and ***.h** files to your project except **gmDebugger**. **gmDebugger** is only required to create a specific debugger application.
- 2) Add or create a platform config file such as **src\platform\win32\gmConfig_p.h**.
- 3) Add some ready made bindings and helpers from **src\binds** as you desire.
- 4) Compile these files with your project, or build and link them as a separate lib. You may need to configure preprocessor options to locate files if they reside in different folders, and you may need to configure precompiled headers if you are using them.
- 5) Create your own application specific binds. Use **src\binds** files and available samples as a reference, and refer to the FAQ and ScriptReference for more assistance.

The following examples are win32 console applications.

Here is a minimal example:

```
#include "gmThread.h" // game monkey script

int main(int argc, char* argv[])
{
    gmMachine machine;
    machine.ExecuteString("print(`Hello world`);");
    getchar(); // Keypress before exit
    return 0;
}
```

Here is an interactive example.

```
#include <windows.h>
#include <mmsystem.h> // multimedia timer (may need winmm.lib)
#include "gmThread.h" // game monkey script

int main(int argc, char* argv[])
{
    // Create virtual machine
    gmMachine* machine = new gmMachine;

    // Get a script from stdin. Some examples:
    // print("Hello world");
    // for( i = 0; i < 10; i=i+1 ) { print("i=",i); sleep(1.0); }
    fprintf(stdout, "Please enter one line of script\n>");
    const int MAX_SCRIPT_SIZE = 4096;
    char script[MAX_SCRIPT_SIZE];
    fgets(script, MAX_SCRIPT_SIZE-1, stdin);

    // Compile the script, but don't run it for now
    int errors = machine->ExecuteString(script, NULL, false, NULL);
    // Dump compile time errors to output
    if(errors)
    {
        bool first = true;
        const char * message;

        while((message = machine->GetLog().GetEntry(first)))
        {
            fprintf(stderr, "%s"GM_NL, message);
        }
    }
    else
    {
        int deltaTime = 0;
        int lastTime = timeGetTime();

        // Keep executing script while threads persist
        while(machine->Execute(deltaTime))
        {
            int curTime = timeGetTime();
            deltaTime = curTime - lastTime;
            lastTime = curTime;
        }
    }

    delete machine; // Finished with VM

    fprintf(stdout, "Script complete. Press a key to exit.");
    getchar(); // Keypress before exit

    return 0;
}
```

What's with the strange comments in the source code?

The source code is commented with doxygen compatible comments (<http://www.doxygen.org/> or maybe <http://sourceforge.net/projects/doxygen/>). The C++ style `///
//` comments are favored over the older C style `/*!
*/` comments due to the inability to nest classic C comments.

GameMonkey Website:

<http://www.somedude.net/gamemonkey/>