Explaining the Quake III Virtual Machine

Ludwig Nussel

SUSE Linux Products GmbH
openSUSE Team

ludwig.nussel@suse.de

FOSDEM 2013

Outline

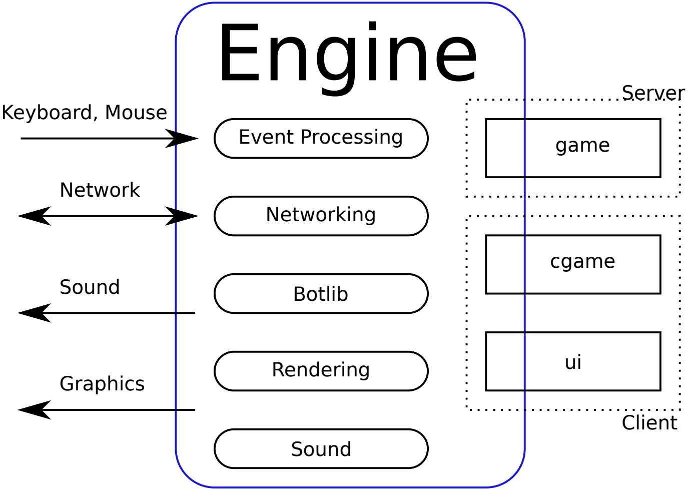
About Quake III Arena

- multiplayer first-person shooter
- developed by id Software
- released in December 1999 for Linux, Windows and Mac
- platform independent Mods
- source released under GPLv2 in August 2005

About ioquake3

- based on id's source release
- compatible to original game
- bug and security fixes
- enhanced portability
  64bit, BSD, Solaris ...
- tons of new features
  IPv6, OpenAL surround sound, VoIP,
  stereo rendering, png image loader ...

Quake III Architecture Overview



Quake III Architecture: the Game Modules

- Game
  - run on server side
  - maintain game state
  - compute bots
  - move objects

- CGame
  - run on client side
  - predict player states
  - tell sound and gfx renderers what to show
- UI
  - run on client side
  - show menus

Typical Ways to Extend a Program

- Native
  - write plugins in C
  - compile as shared library
  - fast
  - no sandbox
  - no memory limit
  - can crash the whole program
- Interpreted
  - use some scripting language
  - slow
  - can be sandboxed
  - can be memory limited
  - main program can catch exceptions

The Quake III Way for Extensions

Allow both, with same code!

Bytecode

- compiled by special compiler (lcc)
- byte code interpreted by a virtual machine
- strict sandbox
- strict memory limit
- main program can catch exceptions and unload VM

Native

- compiled by system C compiler as shared library
- must restrict to embedded libc subset
- must not call malloc()
- must not use external libs

The Virtual Machine Architecture

Overview

- 32bit addressing, little endian
- 32bit floats
- about 60 instructions
- separate memory addressing for code and data
- no dynamic memory allocations
- no GP registers, load/store on 'opstack'

opStack vs Program Stack

- Program stack
  - local variables
  - function parameters
  - saved program counter
- opStack
  - arguments for instructions, results

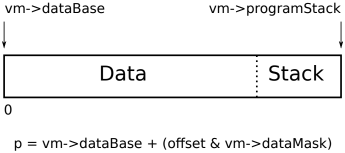
    ```
    OP_CONST  3
    OP_CONST  4
    OP_ADD
    ```

  - function return values

Calling Convention

- similar to x86
- parameters and program counter pushed to stack
- callee responsible for new stack frame
- negative address means syscall
- return value on opstack

VM Memory Layout



- vm memory allocated as one block
- 64k stack at end of image
- code and opstack separate
- memory size rounded to power of two
  - simple access violation checks via mask
- cannot address memory outside that block
  - can't pass pointers

Calling into the VM

- code is one big block
- no symbols
- can only pass int parameters
- start of code has to be dispatcher function
  - first parameter is a command
  - called `vmMain()` for `dlopen()`

```
intptr_t vmMain(int command,
  int arg0, int arg1,
              ..., int arg11) {
  switch (command) {
    case GAME_INIT:
      G_InitGame(arg0, arg1, arg2)
      return 0;
    case GAME_SHUTDOWN:
      G_ShutdownGame(arg0);
      return 0;
...
```

Calling into the Engine

- requests into engine
  - print something
  - open a file
  - play a sound
- expensive operations
  - sinus, cosinus, vector operations
  - memset, memcpy
- pass pointers but not return!

- VM side, C implemenation

```c
typedef enum { CG_PRINT, ... };
void trap_Print(const char *fmt) {
  syscall(CG_PRINT, fmt);
}
```

- host side

```c
intptr_t
CL_CgameSystemCalls(intptr_t *args) {
  switch(args[0]) {
    case CG_PRINT:
    Com_Printf("%s", (char*)VMA(1));
    return 0;
...
```

How the Interpreter Works

Example

```
while (1) {

  r0 = opStack[opStackOfs];
  r1 = opStack[(uint8_t) (opStackOfs - 1)];

nextInstruction2:
  opcode = codeImage[ programCounter++ ];

  switch (opcode) {

    case OP_LOAD4:
          r0 = opStack[opStackOfs] =
               *(int *) &image[r0 & dataMask & ~3 ];
          goto nextInstruction2;
```

More Speed With Native Code

- Byte code interpreter too slow for complex mods
- Translate byte code to native instructions on load
- CPU/OS specific compilers needed
  - currently: x86, x86_64, ppc{,64}, sparc{,64}
  - hard to maintain
- Multiple passes to calculate offsets
- Needs extra memory as native code is bigger
- Need to be careful when optimizing jump targets

VM JIT Compiler

Example

```
case OP_DIVU:
  // mov eax,dword ptr -4[edi + ebx * 4]
  EmitString("8B 44 9F FC");
  // xor edx, edx
  EmitString("33 D2");
  // div dword ptr [edi + ebx * 4]
  EmitString("F7 34 9F");
  // mov dword ptr -4[edi + ebx * 4],eax
  EmitString("89 44 9F FC");
  // sub bl, 1
  EmitCommand(LAST_COMMAND_SUB_BL_1);
```

Future Work

- replace x87 ops with SSE2 in `vm_x86.c`
- write compiler for ARM
- create gcc backend to replace lcc
- use LLVM

Summary

- Quake 3 allows mods as shared library or bytecode
- Bytecode is platform independent
- Game architecture requires strict separation between engine and mod
- CPU/OS specific compilers are a maintenance burden
- VM is simple enough to study and play with it

Links

- http://ioquake3.org/
- http://www.idsoftware.com/
- http://fabiensanglard.net/quake3/
- http://www.icculus.org/~phaethon/q3mc/q3vm_specs.html