

FuBi!

Automatic Function
Exporting for
Networking and
Scripting (!!!)

Scott Bilas
Gas Powered Games



Cell Phones?

Introduction

- Me
 - Scott Bilas
 - Background
- You
 - C++ developers working on boring back-end stuff
 - Visual C++ 6.0 Win32 x86 (but wait!)
 - Developers for other platforms don't leave! Many if not all of these concepts are portable (though *C++ is (mostly) required*). Check the paper for specifics.
- This Talk
 - Glossing over specifics, they are covered in the paper

What Is FuBi?

- First: I'm excited about FuBi!
 - (You should be too)
- What is it?
 - FuBi originally meant “Function Binder”
 - Now it means “foundation of Dungeon Siege”
 - Originally published in Game Programming Gems I
 - By the way, be sure to buy a copy, I hear it's good ;)
 - FuBi is 15 times bigger since then (over a year ago)
 - Kept finding new applications for the system
 - It's the gift that keeps on giving – getting rickety though
 - Wait, let's talk about the problem first

Two Guinea Pigs

- Choose two sample subsystems
 1. Scripting engine
 - Like SiegeSkrit, Java, Python, LUA
 - Say it's a standard p-code VM type thing
 - It's useful (and faster) if it has a parameter stack
 2. Network remote procedure calls (RPC's)
 - All we really need is a network transport for subpackets
 - Compression is always useful

Why Choose These?

- They have something important in common: the need to be able to call arbitrary engine functions
 - Script engine must process CALLFUNC opcode
 - Networking must route an incoming packet to whatever object is expecting it
- Everybody knows how to make them
 - Well treated in books, magazines, and online articles
- Most games need these systems anyway
 - That includes games like Dungeon Siege

The Problem

- How do these systems talk to the game?
 - How does a script call a C++ engine function?
 - How does a subpacket get to the correct C++ object?
- Answer: painfully
 - Often involves nasty packing and unpacking of parameters
 - Other times it's worse: the engine functions must manually parse incoming generic parameters
 - It's 10x too much work, and masks opportunities for some advanced features that we will get to soon

Base Requirements

- Regardless of method, we will always need certain things
- Type database
 - Built-in's: *int, float, const char*...*
 - UDT's: *std::string, Model, Actor, World...*
- Function database
 - Table of function pointers
 - *Model::Animate(), Actor::Think(), World::Sim()*

Base Requirements (cont.)

- Method of filling out the databases
 - Fill the type database – each type has an ID, name, size, “POD-ness”, is-a-handle-ness...
 - Other stuff too, like string versions of enum constants
 - Fill the function database – ID, address, name, calling convention, parameter constraints...
- Method of calling functions
 - Parameter pre- and post-processing
 - Direct function call

Typical Methods

- Databases filled manually
 - *Init* function or other global registration method (tedious)
- Parameter packing/unpacking
 - Each function receives a binary parameter data block
 - Requires serialization to be done by caller and callee
 - Disadvantages are obvious: lack of type safety, overhead in packing and unpacking
- Generic parameters (*argc/argv*)
 - Same disadvantages as above except worse: clients must implement mini-parsers!
- Tools (templates, macros) can help, but it's still too much work, too unsafe, and I'm lazy

Typical Methods (cont.)

- Source code processor (SWIG or homebrew)
 - Best and safest solution so far, but *still* too much work
 - Adds steps to build process, slows down prototyping
 - Requires wrappers, lower performance
 - Exporting is still not *completely* automated
- What gets in the way
 - We do not have access to the compiler's symbol table
 - This is the single greatest problem I have with C++

The FuBi Hack (Uh, Method)

- Use the module's export table
 - *declspec(dllexport)* to export functions to the system
 - Each export entry maps a function name to address
 - Iterate through export data to fill function database
- Take advantage of C++ name mangling
 - Extract class, function name, parameter types, return type, calling convention etc. by de-mangling the C++ name and parsing the results (lex/yacc)
 - Easy to do using functions in *dbghelp.dll*

Sample Usage

- Function we want to export

```
bool PrintMsg( const char* text )  
    { gScreenConsole.Print( text ); }
```

- Exported code

```
#define FUBI_EXPORT __declspec( dllexport )  
  
FUBI_EXPORT PrintMsg( const char* text )  
    { gScreenConsole.Print( text ); }
```

Advantages

- Easy to export any function
 - Just add a tag!
- Works on any kind of function
 - Class method (static or nonstatic), global, vararg, overloads, namespaces, anything!
- Fully automated
 - All engine functions and types can be detected at game startup

Fill The Databases

1. Read EXE export table (MSDN says how)
2. For each function, de-mangle the name
3. Parse the results, extract necessary info
 - a. Function and class name
 - b. Function pointer
 - c. Parameter types and return type
 - d. Calling convention
 - e. Various flags: static, virtual, const
4. Add info to function database, assign ID

Script Compiler

1. Parser detects a function call (say it's like C++)
2. Look up the function entry in the database
3. Verify parameter type match, do necessary conversions
4. Emit CALLFUNC opcode with the function ID as data

Script Virtual Machine

- Assume ordinary p-code based virtual stack machine
 - Store stack in OS format for convenience
 - Grow downward, DWORD-aligned
1. Push parameters on stack left-to-right (going upwards in memory)
 2. Look up function pointer based on ID
 3. Execute blind function call (more on this later)

Network Router

- Say it's just a packet pipe
 - Send chunks of data (subpackets) to it and it will route it to the destination machine(s)
 - Each chunk will be the function ID followed by parameters, followed by indirect data
- Careful to watch for passing pointers over the network
 - Addresses cannot resolve
 - We can handle this!

Network Router (cont.)

1. Networked function automatically wraps up parameters for the transport
 - Use a macro to make it easy
 - Requires some inline assembly
2. Special handling for pass-by-reference parameters
 - Pointers don't work over networks
 - POD types can tack raw data onto end of chunk
 - Non-POD types can be converted to/from a “network cookie”
3. Special handling for *const char**, *std::string*
4. Resolve “this” pointers same as 2(b).

Network Dispatcher

1. Resolve function ID from chunk into pointer
2. Resolve pointers, “this”, and strings (reverse of network router)

Blind Function Caller

- Easiest to write it all in assembly code (see paper)
- Require one per calling convention
 - *cdecl*, *stdcall*, *thiscall*, *thiscall + vararg* (no *fastcall*)
- Each blind caller performs same ops
 1. Receive function address, stack pointer, and size of parameters
 2. Reserve state on thread stack, then copy virtual stack onto thread stack
 3. *CALL* the function
 4. Pull *ST0* off FPU stack if function returns a float/double, otherwise return value is in *eax*

Convenience Is Power!

- Your game has a console (if not, it should)
- Give the console the ability to execute scripts dynamically
 - This should be easy
 - Just wrap up what's entered in the console in a function and execute it on the fly
- Now this is an excellent test tool

Quick Tests and Tweakers

- It takes 30 seconds to add something like this to any .cpp file and instant recompile:

```
FUBI_EXPORT TestCrash( void )
{  *(int*)0 = 0;  }
FUBI_EXPORT TweakMyVar( float f )
{  gMySystem.SetVar( f );  }
```

- Convenience like this is *powerful* – can immediately bring up an entire interactive debug system in seconds.
- Dungeon Siege exports nearly 2000 functions currently. Why? Because we can!



Insert Sexy Demo Here

Porting Notes

- No *dllexport* facility on your platform?
 - Linker can make a .map file that you can parse
 - Or get what you need from debug symbols
 - Will need text file to say which functions are exported (otherwise can't tell the difference)
- No helper function to de-mangle names?
 - If using an open source compiler like *gcc*, figuring out how the mangler works should be easy

Porting Notes (cont.)

- Not x86 instruction set?
 - Shouldn't be difficult to port to another assembly language
- Programming in something other than C++?
 - C++ wrappers that call engine functions
 - All we need is the name mangling to get the type info
 - I use this technique in reverse on Dungeon Siege for calling Skrit functions from C++
 - The function itself is immaterial, we just need the mangled name to get type info!

Final Notes

- Regarding use of *dbghelp.dll*
 - Probably don't want to ship this with the game
 - Also security may be an issue if multiplayer
 - Write a little postprocessor that strips the EXE of its export table and throws it in the resource section in a form that FuBi can read directly (maybe encrypt it too, not that it will stop hackers)
- Regarding *virtual* functions
 - We don't know which entries in the vtbl are for what, so can't call the proper dynamic function (easy to work around though)



The End (?)

Next Steps

- Pretty pedestrian v1.0 stuff so far
 - It took about a week to implement FuBi 1.0
 - Around 1,000 lines of code originally
 - Just the beginning of the really fun stuff
- Why is this all worth the trouble?
- Step back and examine the system from a higher level:

Function database + type database
= *self-awareness*

Self-Awareness

- Self-Awareness: the game knows its own types
 - Then it can do lots of things automatically
 - Game can intelligently debug and report on itself
- Other languages such as C# and Java already have this ability (damn them)
- Apparently most of us use C++
 - We'll probably never get this ability in our language
- FuBi can help bridge the gap
 - Can enhance this through special conventions

Special Conventions

- Tell FuBi about special naming conventions for extra special functionality
- Singleton lookup
- Network cookie transformation
 - Also use for save game!
- Permissions, function sets

Special Conventions (cont.)

- Documentation
- Possibilities are limitless
 - We are defining a functional export protocol
 - This is nothing new, except that FuBi provides the greatly needed automation
 - This removes requirements for `#include` and linking

External Binding

- FuBi can live anywhere, no reason we have to stick it on the game EXE
- It can bind to arbitrary modules, so long as there are exports to be found
- Use it for “native interfaces” useful to the mod community
 - All they have to do is *dllexport* their functions and FuBi can route script calls to their DLL's
- By the way, this makes adding a scripting engine to any app simple – just link it in and it can autodetect everything

External Binding (cont.)

- Use it for dynamic debugging
 - Be able to attach/detach a *devtools.dll* during normal gameplay
 - Need a change during debugging? Detach the DLL, update it, recompile it, then reattach and reinit FuBi
 - New functions are now ready to go, all without exiting and reloading the game

Other Fun Possibilities

- Detect nested RPC's
 - Obviously can be bad for performance
- Execute callbacks to validate params passed over the network

Contact Info

Scott Bilas

<http://scottbilas.com>

Web site contains this presentation in PowerPoint and IE HTML formats, plus sample code for VC++ 6.0 on x86 Win32.