



# debugging fun

jonathan garrett

with contributions from the tech team

5/22/07

# introduction

- show some of the things we do to debug
  - some very obvious – you're probably doing a number of them already
- show some debugger features and shortcuts
- not telling anyone how to suck eggs!
- promote open discussion for sharing of strategies and ideas (how nice!)

# our strategies

- if it's obvious, just fix it !
- otherwise, first send out a quick mail to *tech*
  - briefer the better (more chance it'll be read)
  - anyone seen this before ? - anyone think of a likely cause ?
  - keeps others in the loop – it'll be in the back of their mind

# strategies

- if it used to work – when did it break ? -  
what changed before then ?
  - use p4 history / time lapse view
- if not obvious, start breaking down the  
problem (key)

# strategies

- isolate a reproducible test case
  - smaller the better - disable as much code as possible
  - does it happen in a *viewer* ?
  - strip as much away as possible to focus on the bug
  - bisect to further narrow down – happen between after *A* and before *B*
- step and watch what's going on

# strategies

- often have a good idea as to the cause
  - create hypotheses and think them through
- easier when side effect
  - crash / memory trash / visual artifact (object flickering / behaving strangely)
- crash bugs are usually easy to track down
  - what was the actual cause ? – bad load / store ? - how could this happen ? – examine inputs / locals / structs - trace back to see where this data came from – use *break-on-write*
- subtle logic related bugs harder to track down (again reproducible case helps)

# strategies

- known system:
  - think through flow of code
  - what could lead to these symptoms ?
  - be conscious of just taking other people suggestions / assumptions as gospel – at some point need to question whether they're correct

# strategies

- complex system or not known as well:
  - break it down
  - test components separately
  - then build back up
  - can be quite hacky - more work ? – usually saves time in the end though
  - Terry's recent example...

# strategies

- rendering bugs – tools:
  - create small test case (1 or 2 tris) – trace through code / dump output at each stage
  - add helper code to allow breaking on specific problem asset / component (for verts, break if within tolerance)
- rendering bugs – runtime:
  - toggle things on / off to narrow down

# strategies – trashed memory

- trashed memory
  - what's the trashed data ? – does it look familiar ? – does it have a pattern ? – does it have a stride ? – what's the spread ?
  - use break-on-write
- binary dump of memory and offline diff

# strategies – debug vs release

- frequent causes:
  - un-initialized data (could only manifest when data has mutated)
  - aliasing issues
- compile individual files `-O0` to narrow down
  - could also be a compiler bug ☹
    - `IG_NOP` to isolate
- often have issues with original vs optimized versions
  - keep original version on an `if-def`
  - helpful when optimized version first comes online

# strategies

- if still stuck, talk through what's happening with someone (anyone!)
  - often just the act of you explaining will hi-light the problem – the other person never said anything!
- if you can't solve it (or not your bug), get as much info as possible before handing it off
  - can often be solved just from this info

# strategies – preventative / support

- when written new code, step through your own code in debug to make sure it's doing what you think
- set up for easier debugging
  - add asserts / runtime checks (hot topic how far should go with this)
  - use struct padding to stash helpful info - eg. ppu low-level anim system stashes moby ptr in blend tree pad word – useful from spu

# strategies – preventative / support

- in game code for testing – extra work, but worthwhile
  - debug collision prim
  - physics collision buddy
  - shader debug display
  - push-buffer scrub
- write helper code to dump state when bug occurs
  - RC3 dumps state of collision grids
  - I8 / RCF dumps anim system blend-tree
  - RCF dumps z-buffer for occlusion (or will do)

# strategies – preventative / support

- **COMPILE\_ASSERT**

- check alignment
  - check buffer sizes
  - check struct sizes – asm may be tuned to particular size
- test apps to help debug isolated processes (eg. vert pack / unpack) – go back to test when something breaks (Mark)

# strategies – preventative / support

- when fix a bug, think whether any other similar cases ? – fix them too
- after fixed the bug, is there anything that could be done to prevent it next time, or help track it down quicker ?
- mail out to *tech* to let them know the cause and resolution

# strategies - spu

- issues from asynchronous nature
  - don't know when our module will run
  - ppu doing something different
- everything data – code + stack = easy to trash and really break things
- due to 256k limit, some systems can only be built in release – ugh!
- old favorites – `IG_STOP`, `printf`

# strategies - spu

- *timeouts* - ppu watchdog ensuring spu module completed
  - can be an issue if your module didn't even start
- “exception” handler
  - detects error condition
    - dmas helper structs back to ppu
    - print to tty
- our new dynamic code system
  - no debugger support (yet)
  - nightmare for another discussion

# strategies

- Terry's recent fun...



# debugger

- **ctrl+g** is your friend
- hardware break is also your friend
  - break on read / write (8 byte granularity)
  - can't break on DMA
  - if break in **memcpy** / **memset** run to **blr** then continue
  - will be set-able from code in future SDK update
- conditional break: **((u32)moby\_ == 0x6462fd00)**

# debugger

- watch window is another of your friends (so many friends!)
  - keypad +, - and cursors are all you need to navigate
  - cast ptr to sized array: `(MyType*)my_ptr, 10`  
(keyborad +, - change the range start)
  - **enter** to edit current value
  - **ctrl+enter** to edit current watch
  - **tab** steps all through display format (decimal, hex, float, octal (!) etc.) - **ctrl+h** steps all open watches

# debugger

– **>** and **<** increment / decrement a selected variable (ptrs increment as expected)

– examine variables in a caller's stack frame:

**{ stack:previous\_function } variable\_name**

eg:

COLL::DebugDrawGeomTriMesh(...)
COLL::DebugDrawTriMeshObjects(...)
DEBUG::DrawCollDebug()
DEBUG::DebugDisplayData()
DrawUsualObjects()
16

– watch **{ stack:DrawCollDebug } pb\_usage\_threshold** shows value of **pb\_usage\_threshold** in **DrawCollDebug**

# debugger

- auto-expansion of known types (and floating tooltips):
  - `autoexp.dat` (`c:\program files\sn systems\ps3\bin`)
  - same (or very similar) format to dev-studio
    - eg. `vec4f` display:
      - `vec4f=<x, f>|<y, f>|<z, f>|<w, f>`
  - would be nice to have a standard set of these for common Insomniac types

# debugger - scripting

- made use of this on PS2 - PS3 support is very similar
- powerful - evaluation, examine / modify memory, start / stop, file I/O, GDI type “rendering”
- reasonable documentation and examples:
  - `c:\program files\sn systems\ps3\help\ProDG_PS3_Debugger-E.chm`  
`\examples\debuggerscripting`
- Shaun’s script displays a moby’s class name:
  - `IGG::g_MobyCon.m_class_debug_info[my_moby_instance.m_iClass].m_class_name`

```
void MobyInstanceHandler(sn_uint32 processId, sn_uint32 ThreadIDWord0, sn_uint32
    ThreadIDWord1, const char* pVariable, char * pOutput, sn_uint32 uSize)
{
    int        result;
    char        expression[256];
    sn_uint64  u64ThreadId;
    sn_val      iclass;
    sn_val      class_name_addr;

    u64ThreadId.word[0] = ThreadIDWord0;
    u64ThreadId.word[1] = ThreadIDWord1;

1  sprintf(expression, "%s.m_iclass", pVariable);
2  result = PS3EvaluateExpression(processId, u64ThreadId, &iclass, expression);
3  sprintf(expression, "IGG::g_MobyCon.m_class_debug_info[%d].m_class_name",
    iclass.val.u16);
4  result = PS3EvaluateExpression(processId, u64ThreadId, &class_name_addr,
    expression);

    if (!SN_FAILED(result))
    {
        if (class_name_addr.type == snval_uint64)
        {
5         PS3GetMemory(processId, u64ThreadId, (void*)pOutput,
            class_name_addr.val.Address, uSize);
        }
    }
}
```

# debugger

- **nop** asserts / traps
  - branches to remove certain code paths
  - if you're feeling adventurous can hand assemble instructions
- edit-and-continue
  - build in some helper globals and compile
  - can then direct code using those globals on the fly

# debugger

- crashed ps3
  - connect from the comfort of your own devkit
  - just need the IP of the crashed machine and (ideally) access to the self (nicer if you also have source in sync - elf only contains references to the code)
  - add new target, select in debugger, load just the symbols for the relevant elf
  - Shaun's step-by-step is on the tech wiki

# forced stack trace

- trigger stack trace from code

```
void IGG::StackTrace(u32 max_depth_,  
                    const char* filename_)
```

- useful to see output from testers without resorting to asserts
- collision does it to help identify offending calls from gamecode

# release / asm level debugging

- frequently need to debug at the asm level
- very useful – try not to be intimidated
- the more asm you know, the more sense it'll make
  - PPC ISA has instruction listing:  
`\\locutus\Research\Docs\PPC\PPC_Vers202_Book1_public.pdf`
- debugger can sometimes give us the correct data
  - always for globals - sometimes for locals (often shows wrong value – but might look correct!)
- registers never lie though

# abi

- abi defines register usage (inc. how parameters are passed between functions)

r1	stack ptr
----	-----------

r3 - r10	int input args (first 8)
r3 - r4	int return args

f1 - f8	float input args (first 8)
f1	float return arg

r0	volatile (caller save)
r3 - r12	volatile (caller save)
r13	
r14 - r31	non-volatile (callee save)

f0	volatile (caller save)
f9 - f13	volatile (caller save)
f14 - f31	non-volatile (callee save)

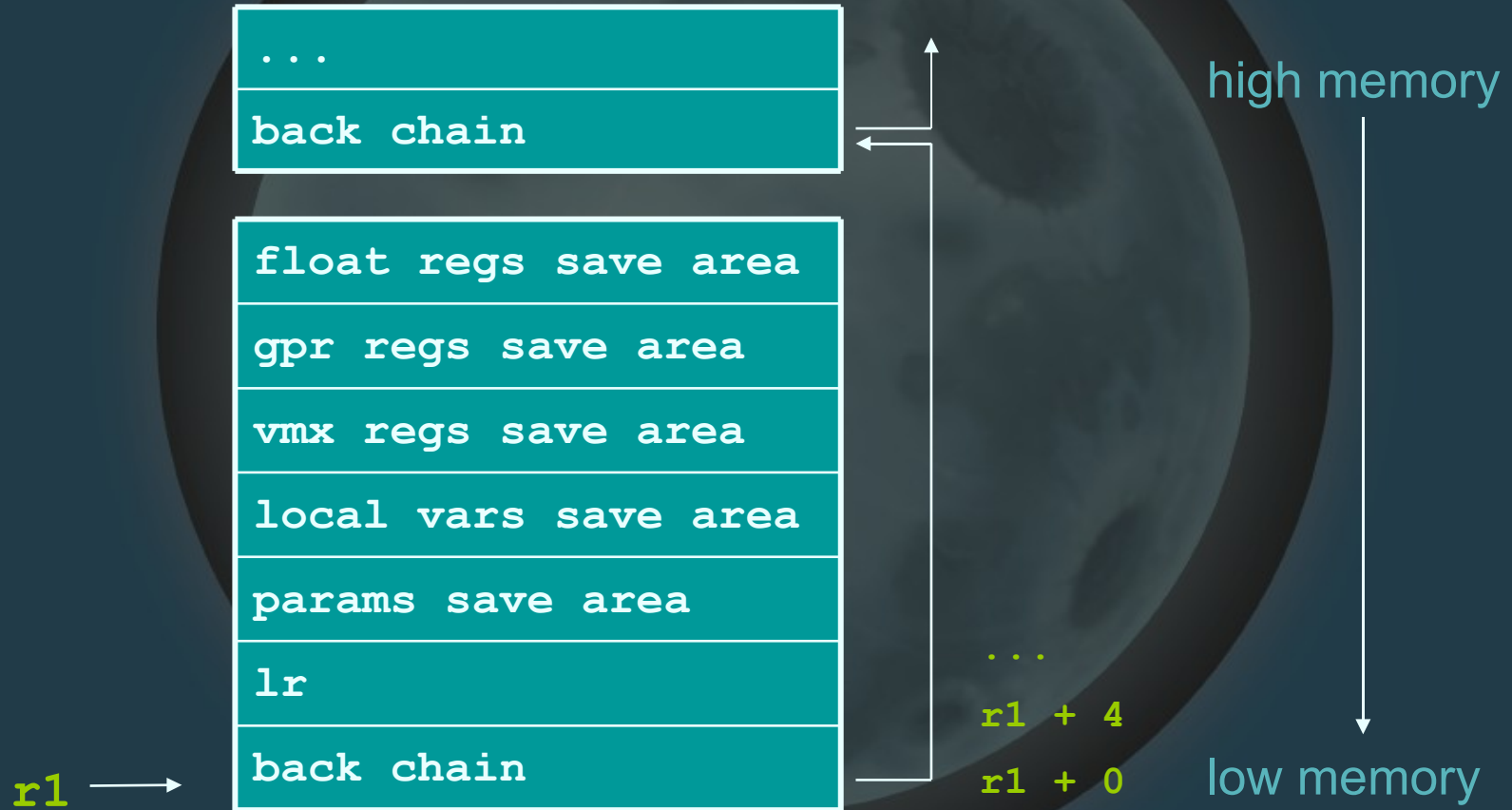
# abi

- arguments which don't fit in registers are passed through the stack - **r3** = ptr to caller storage area
- if calling a class member **r3 = this**
- **(MyStruct\*) \$5** (3<sup>rd</sup> parameter to my function)
- note: watch **(u32) \$3** or **(f32) \$f1**

# abi

- also defines stack usage
  - stack grows down from high addresses to low
  - stack addresses have the top nybble set to **0xd**
  - stack starts at a little above **0xd0010000** and defaults to 64k

# abi - stack frame (simplified)



# abi

- before calling a function, current function stashes any volatile registers in its stack-frame
- branches to function (**lr** = return address)
- on entry, a function (generally) creates a stack-frame
  - updates stack ptr
  - stashes link register in caller's frame
  - stashes any non-volatile registers it uses in its frame
- on exit, it restores any non-volatile registers, loads the stashed **lr** and returns

```
struct t_MyStruct
```

```
{
```

```
    u32 m_a;
```

```
    u32 m_b;
```

```
    u32 m_c;
```

```
};
```

```
u32 func_b(u32 val_, t_MyStruct* s_)
```

```
{
```

```
    printf("hello %d - %d\n", s_>m_b, val_);
```

```
    return s_>m_c;
```

```
}
```

```
u32 func_a(t_MyStruct* s_)
```

```
{
```

```
    u32 ret = func_b(123, s_);
```

```
    return ret;
```

```
}
```

```
u32 func_b(u32 val_, t_MyStruct* s_)
```

```
00000000 <_Z6func_bjP10t_MyStruct>: <- r3 = u32, r4 = MyStruct*
   0:    mflr    r0                <- r0 = lr
   4:    stwu    r1,-32(r1)         <- alloc stack frame and stash sp
   8:    mr      r5,r3             <- r5 = r3 (setup for arg 3 to printf)
  c:    lis     r3,0x1234          <- r3 = 0x1234<<32 = hi16("hello %d - %d\n")
 10:    addi    r3,r3,0x5678        <- r3 += 0x5678 = lo16("hello %d - %d\n")
 14:    stw     r0,36(r1)          <- stash lr (into caller's stack frame)
 18:    stw     r29,20(r1)         <- stash r29
 1c:    mr      r29,r4             <- r29 = r4
 20:    lwz     r4,4(r4)           <- r4 = s_->m_b
 24:    crclr   4*cr1+eq          <- clear eq bit of cr1 (ie. no float args)
 28:    bl      printf            <- r3 = string, r4 = s_->m_b, r5 = val_
 2c:    lwz     r0,36(r1)          <- load lr (from caller's stack frame)
 30:    lwz     r3,8(r29)          <- r3 = s_->c
 34:    lwz     r29,20(r1)         <- restore r29
 38:    addi    r1,r1,32           <- free stack frame
 3c:    mtlr    r0                <- lr = r0
 40:    blr                                <- return
```

```
u32 func_a(t_MyStruct* s_)
```

```
00000050 <_Z6func_aP10t_MyStruct>: <- r3 = MyStruct*
  50:    mr      r4,r3             <- r4 = r3 = arg 2
  54:    li      r3,123            <- r3 = 123 = arg 1
  58:    b       func_b            <- call func_b
```

# desperation debugging

- **printf** – surprisingly effective
  - show how values change up to problem point
  - **TRACE** macros which compile out
  - but does change the code (timing could be affected)
- trace through a global (volatile)
  - or mini ring-buffer of previous ops

# desperation debugging

- sprinkled validates to narrow down erroneous event (eg. we know something storing `0x3f800000` to our ptr – assert top byte not `0x3f`)
  - start sparse then bisect and narrow down
- does the trashed data look familiar ?

# trashed / misaligned data

- knowing how familiar values look can help track down trashed data / bad packing
- identifying known values can be useful
  - $0xffffffff = 4294967295 = -1$
  - $0x3f800000 = 1065353216 = 1.0f$
  - $0x40000000 = 1073741824 = 2.0f$
- float values are usually pretty easy to spot
- vectors / matrices usually pretty easy too

# trashed / misaligned data

- recent bug - struct contained something like this:

```
my_struct      0x1063A560
  .m_word1     1346454576      u32
  .m_word2     1346454577      u32
  .m_float1    12969887744.0   f32
  . . .
```

# trashed / misaligned data

- memory view:

1063A560	50414430 50414431 50414432 50414433	PAD0PAD1PAD2PAD3
1063A564	50414434 50414435 50414436 50414437	PAD4PAD5PAD6PAD7
1063A568	...	

- tools packing issue which was easily fixed

# misaligned data

- unlike the PC, can only load / store on boundary of the native type:

byte	1
short	2
word	4
float	4
dword	8
vector	16

- vector load reads from address `&~0xf` (i.e. ignores low bits) so don't get the data you expect
- misaligned load of other types results in alignment exception

# summary

- difficult to describe how to debug
- hopefully shared something
- hopefully opened up forum for discussion
- debugger is powerful
- how do you debug ? – send us your tricks and we'll add them to the wiki

end!

- questions ?

