Developing Technology for Ratchet and Clank Future: Tools of Destruction

Mike Acton, Engine Director
<macton@insomniacgames.com>
with
Eric Christensen, Principal Programmer
<ec@insomniacgames.com>

Sideline: How is the programming divided?

- Tech Team (Engine)
- Tools Team
- Multiplayer Gameplay
- Singleplayer Gameplay (per Game)

Sideline: What is game code like?

- Gameplay systems
  - Using simple C++
- Engine systems
  - Even simpler C++ and C
  - For C/C++, heavy use of PPU/SPU intrinsics
  - Plenty of hand-optimized SPU assembly
- Only runs on the PS3.
  - Not cross-platform, not-portable.

Our Approach

- “Manual Solution” - “Very optimized codes but at cost of programmability.” -- Marc Gonzales, IBM

- That's us. The “solutions” often cost more than the problem when lack of experience is factored in.
- And with experience, programming becomes easier.
- Remember: 16ms.

What we started with...

- Resistance, Launch title 2006
  - Collision detection on 2 SPUs
  - Physics, SPU managed jobs
  - Special FX, SPU managed jobs
  - Geometry culling, SPU managed jobs
  - Low-level Animation, SPU managed jobs
  - ...and lots more!
- Resistance did take advantage of the Cell, but...
  - We could still do much better.

Today

- The Cell is no longer new
  - The concepts were never really that new, but still...
  - We've had a couple of years with the architecture.
  - We're accustomed to the platform and tools.

Problems developing on Cell?

- Cache and DMA data design?
  - No. Manual design. Make it simple, keep it simple.
- Branch prediction, branch-free coding?
  - No. Target branch-free, add few branches for performance.
- In-order processing?
  - No. Accustomed to it. Easier to optimize anyway.
- No load->store forwarding on PPU?
  - No. Annoying, but focus is on the SPUs anyway.

Problems developing on Cell?

- Decomposing into parallel code?
  - No. Well, sometimes, but we can work it out.
- Compilers?
  - No. GCC compilers for SPU and PPU – Have issues, but manageable. And getting better.
- Debugger?
  - SN debugger (win32). Typical issues, nothing huge.
  - gdb.
- Bandwidth?
  - No. Very good bandwidth for good data.

Problems developing on Cell?

- How about language? Do we need a better one?
  - (Dear Jim, the language is not the issue.)
  - Another language or yet another library to “solve” the Cell “problem”? WTH?
  - The solution is not to hide the issues,
  - The solution is to understand the issues.
  - Understand the data and the code will follow.
  - Pointless anyway. Can't wait.

Problems developing on Cell?

- Performance analysis tools?
  - Sony's tools. Could always be better. But improving all the time.
  - IBM's tools. spu_timing and simulator are handy.
  - Nomatter what, performance is our responsibility tools are just that – tools.
  - There's no “make game fast” button.
  - The best analysis tool by far?
    - Your brain.

Our Real Problems

- Time.
  - Development time and manpower.
    - Training. And more training.
  - Iteration time.
    - Build times. Problem, but we're looking at it.
    - Development kit. Started using PS3/Linux for iteration.

Our Real Problems

- Over-design, unnecessary complexity.
  - Mature code usually gets smaller, faster and simpler.
- Solutions that cause more problems then they solve.
  - For example...

SPU Management, Why, Oh Why?

- Trying to find a single SPU job manager is a lost cause. i.e. SPURS
- Dynamic job management is just like dynamic memory mangement (but time vs. space)
- ... and we wouldn't use malloc/free, why would want to use a job manager/scheduler?

More SPURS?

- SPURS, 1000+ of jobs per 16ms
  - Doing what?
  - Single objects with code?
  - Bad practice in general.
- Real, good data is very manageable.
  - Know inputs and outputs
  - Know what reads when and where
  - Know what writes when and where
  - Write it down. Seriously. Do it. On paper. With a pen.

Away from SPURS?

- Resistance used a mix of job management and manual. We're moving toward much more manual control.
- But what about middleware?
  - Red herring.
  - Use mix of PPU and SPU libraries. Pass whatever data is needed.

What's been our strategy for RCF?

- Put more on the SPUs, less on the PPU.
- Less PPU/SPU synchronization.
- Better dataflow organization.
- Better SPU code design.
- Concentrating on major engine systems

Sideline: Basic Philosophies

- Not porting to the Cell, designing for the Cell.
  - SPUs are the core of the Cell, the PPU is a minor player.
- THE DATA IS EVERYTHING.
  - Good code = Data transformation kernel
    - Small
    - Fast
    - Does nothing more than it needs to,
    - No extra complexity

More on the SPUs, less on the PPU

- The SPUs are not coprocessors.
- The PPU is already (always) a bottleneck.
- We now have in-house expertise, so...
  - Now we're sharing it more among our team.
  - We even have a weekly “SPU Ninja” class
- Targetting SPUs first.
- Building data for the RSX

Less PPU/SPU Synchronization

- On the heels of the collision detection system,
  - More deferred updates
  - Less immediate mode requests
- Grouping data by type
  - Handling similar types together
- Decoupling engine from gameplay systems
- Moving dataflow management to the SPUs

Sideline: Sequential Consistency

- Load/Store ordering is the basis of lock-free coding.
- Yes, the compiler can re-order. So what?
- Common hardware can too. i.e. Not a new issue.
- Hardware: x86 (sfence,lfence); PPC (lwsync)
- Compiler:
  - Single large basic block (asm or inline asm)
  - 3 blocks, with enforced order (external compile units)
  - Link-time optimization screws with this.

Better Dataflow Organization

- One system at a time, we're...
  - Defining the inputs and outputs
  - Defining exactly what memory is read and written to
  - ...and when.
  - Decoupling systems so they can be tested independently.
- We're doing this systematically.
  - There is no silver bullet.
- A well-defined dataflow is necessary for the Cell, and is a skill that we'll use for the next-generation.

Better SPU Code Design

- Fixing dodgy scalar code.
- Small, fast and custom to the SPU.
- SPU intrinsics often get us most of the way.
- But sometimes we need hand-optimized ASM.
  - Some places have a fear of asm, why?
  - This is balanced with flexibility.
- Large transformations split into individual pipeline stages.

Sideline: Dynamic Code Loading

- On SPU, Code is data
- Double buffer / stream same as data
- Very easy to do (No need for special libraries)
  - Compile the code
  - Dump the object as binary
  - Load binary as data
  - Jump to binary location (e.g. Normal function pointer)
  - Pass everything as parameters, the ABI won't change.

Concentrating on Major Systems

- As opposed to unique gameplay code
- The trivial stuff is long done. Now it's about EVERYTHING.
- Understanding that SPUs don't significantly affect the core design:
  - Whatever the PPU can do, the SPUs can do better.
  - Memory impact is (mostly) contained to system
    - i.e. PPU systems impact others via cache
    - But TLB is an issue either way.
  - Code and data size can be managed.

Example: Physics, Before

- Heavy PPU Synchronization
- SPU was used as a co-processor
- SPU “packets” were built on the PPU
  - For small jobs, this often took more time than the processing!
- SPU code was scalar port
- Many stage pipeline with PPU synchronization at each stage.
- Scattered data (No dataflow design)

Physics, Now

- Pipeline well defined, SPU-driven
  - e.g. Code uploading is controlled by SPU
- SPU processing completely asynchronous
- Data well-organized and defined.
- No (or minimal) PPU intervention
- Also, will be:
  - Re-organizing data for better SIMD processing
  - Prefer si_* intrinsics and asm to spu_* intrinsics
  - Better local store management for bigger jobs.

Other Stuff

- Shader-like code
- RSX data generation on SPUs
  - Which also allows for less synchronization
- More deferred mode systems.
  - Programmers are getting more used to the model
  - e.g. Even collision detection, which had both in Resistance.

Special Effects

- Some procedural effects data. Generate the data needed by the RSX.
- Particle effects system:
  - Now, building core kernels that are shared among different types of effects
  - Now, grouping effects by type
  - Now, asynchronous effect processing
  - Now, optimizing code and data for the SPUs
  - Example: Random point on sphere

Almost The End

- The issues that the Cell brings to the fore are not going to go away.
- Teams need practice and experience.
- Modern systems still benefit from heavy optimization.
- Design around asynchronous processing.
- Don't be afraid to learn and change.
- Also see: CellPerformance.com

How do we build a great team?

- Smart, driven people.
- Practice.
- Training
  - Parallel Programming
  - Low-level Optimization
  - Data Design

The Insomniac Tech Team

- Al Hastings
- Mike Day
- Reddy Sambavaran
- Eric Christensen
- Mark Lee
- Jonny Garrett
- Joe Valenzuela
- Carl Glave
- Terry Cohen
- Rob Wyatt
- YOU?

Random Stuff to Fill Time

- Auto-SIMDization. No value e for games.
- Auto-parallelization. Dead-end.
- How much parallelism is in games?
  - A whole lot. Will be able to fill 32 Cells.
- What do we want in Cell v2?
  - Half precision floats (full native math)
  - Bit interleave instruction. Maybe dot product too.
  - Full 128 bit shifts and rotates.
  - Fill out the integer instruction sets.